Advanced Algorithms Analysis and Design (CS702)

https://studyc.info/

Contents

Lecture 01 What, Why and Where Algorithms 1

Lecture 02 Mathematical Tools for Design and Analysis of Algorithms 5

Lecture 03 Logic and Proving Techniques 10

Lecture 04 Mathematical Induction 16

Lecture 05 Strong Mathematical Induction 22

Lecture 06 Fibonacci Sequences 27

Lecture 07 Recurrence Relations 37

Lecture 08 Recurrence Relations 45

Lecture 09 Further Techniques Solving Recurrence Relations 52

Lecture 10 Time Complexity of Algorithms (Asymptotic Notations) 64

Lecture 11 Relations Over Asymptotic Notations 73

Lecture 12 Design of Algorithms using Brute Force Approach 80

Lecture 13 Designing Algorithms using Brute Force and Divide & Conquer Approaches 88

Lecture 14 Designing Algorithms using Divide & Conquer Approach 92

Lecture 15 Dynamic Programming for Solving Optimization Problems 103

Lecture 16 Chain Matrix Multiplication Problem using Dynamic Programming 110

Lecture 17 Assembly-Line Scheduling Problem 116

Lecture 18 2-Line Assembly Scheduling Problem 123

Lecture 19 0-1 Knapsack Problem using Dynamic Programming 127

Lecture 20 0-1 Knapsack Problem's Algorithm (using Dynamic Programming) and Optimal Weight Triangulation 135

Lecture 21 Optimal Weight Triangulation 146

Lecture 22 Review of Lectures 01-21 149

Lecture 23 Longest Common Subsequence (Dynamic Algorithm) & Optimal Binary Search Trees 150

Lecture 24 Optimal Binary Search Trees (Constructing Dynamic Programming) 160

Lecture 25 Greedy Algorithms 166

Lecture 26 Huffman Coding 176

Lecture 27 Huffman Coding Problem and Graph Theory 186

Lecture 28 Breadth First Search 197

Lecture 29 Proof (Breadth First Search Algorithm) and Depth First Search 205

Lecture 30 Proof (White Path Theorem) & Applications of Depth First Search 216

Lecture 31 Backtracking and Branch & Bound Algorithms 221

https://studyc.info/

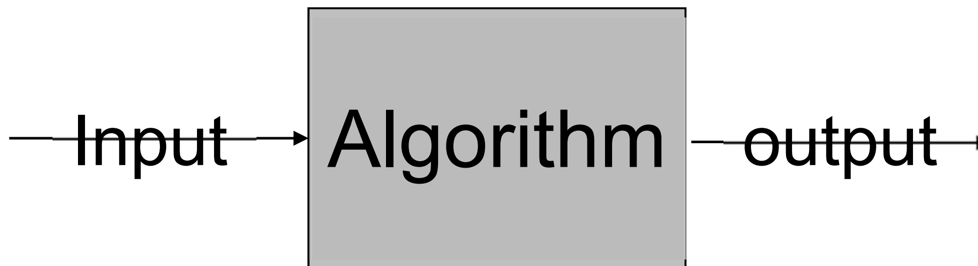
Lecture 32 Minimal Spanning Tree Problem ... 227
Lecture 33 Single-Source Shortest Path ... 242
Lecture 34 Proof: Bellman-Ford Algorithm & Shortest Paths in Directed Acyclic Graphs ... 252
Lecture 35 Dijkstra's Algorithm ... 257
Lecture 36 All Pairs Shortest Paths ... 266
Lecture 37 The Floyd-Warshall Algorithm and Johnson's Algorithm ... 272
Lecture 38 Number Theoretic Algorithms ... 283
Lecture 39 Number Theoretic Algorithms ... 293
Lecture 40 Chinese Remainder Theorem and RSA Cryptosystem ... 302
Lecture 41 RSA Cryptosystem and String Matching ... 309
Lecture 42 String Matching ... 314
Lecture 43 Polynomials and Fast Fourier Transform ... 323
Lecture 44 NP Completeness ... 333
Lecture 45 Review of Lectures 01-44 ... 345

https://studyc.info/

Lecture 01 What, Why and Where Algorithms . . .

What is algorithm?

- A computer algorithm is a detailed step-by-step method for solving a problem by using a computer.
- An algorithm is a sequence of unambiguous instructions for solving a problem in a finite amount of time.
- An Algorithm is well defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output.
- More generally, an Algorithm is any well defined computational procedure that takes collection of elements as input and produces a collection of elements as output.



Popular Algorithms, Factors of Dependence

Most basic and popular algorithms are *sorting algorithms* & *Searching algorithms*

Which algorithm is best?

- Mainly, it depends upon various factors, for example in case of sorting
- The number of items to be sorted
- The extent to which the items are already sorted
- Possible restrictions on the item values
- The kind of storage device to be used etc.

One Problem, Many Algorithms

Problem: The statement of the problem specifies, in general terms, the desired input/output relationship.
Algorithm: The algorithm describes a specific computational procedure for achieving input/output relationship.
Example: One might need to sort a sequence of numbers into non-decreasing order.
Algorithms: Various algorithms e.g. merge sort, quick sort, heap sorts etc.

Important Designing Techniques

- **Brute Force:** Straightforward, naive approach, mostly expensive
- **Divide-and-Conquer:** Divide into smaller sub-problems
- **Iterative Improvement:** Improve one change at a time

https://studyc.info/

- **Decrease-and-Conquer:** Decrease instance size
- **Transform-and-Conquer:** Modify problem first and then solve it
- **Space and Time Tradeoffs:** Use more space now to save time later

Greedy Approach

- Locally optimal decisions; cannot change once made.
- Efficient
- Easy to implement
- The solution is expected to be optimal
- Every problem may not have greedy solution

Dynamic programming

- Decompose into sub-problems like divide and conquer
- Sub-problems are dependant
- Record results of smaller sub-problems
- Re-use it for further occurrence
- Mostly reduces complexity exponential to polynomial

Problem Solving Phases

Analysis

- How does system work?
- Breaking a system down to known components
- How components (processes) relate to each other
- Breaking a process down to known functions

Synthesis

- Building tools
- Building functions with supporting tools
- Composing functions to form a process
- How components should be put together?
- Final solution

Problem Solving Process

- Problem
- Strategy
- Algorithm
  - Input
  - Output
  - Steps
- Analysis
  - Correctness
  - Time & Space
  - Optimality

https://studyc.info/

- Implementation
- Verification

Model of Computation (Assumptions)

- Design assumption
  - Level of abstraction which meets our requirements
  - Neither more nor less e.g. [0, 1] infinite continuous interval
- Analysis independent of the variations in
  - Machine
  - Operating system
  - Programming languages
  - Compiler etc.
- Low-level details will not be considered
- Our model will be an abstraction of a standard generic single-processor machine, called a random access machine or RAM.
- A RAM is assumed to be an idealized machine
  - Infinitely large random-access memory
  - Instructions execute sequentially
- Every instruction is in fact a basic operation on two values in the machines memory which takes unit time.
- These might be characters or integers.
- Example of basic operations include
  - Assigning a value to a variable
  - Arithmetic operation (+, - , × , /) on integers
  - Performing any comparison e.g. $a < b$
  - Boolean operations
  - Accessing an element of an array.
- In theoretical analysis, computational complexity
  - Estimated in asymptotic sense, i.e.
  - Estimating for large inputs
- Big O, Omega, Theta etc. notations are used to compute the complexity
- Asymptotic notations are used because different implementations of algorithm may differ in efficiency
- Efficiencies of two given algorithm are related
  - By a constant multiplicative factor
  - Called hidden constant.

Drawbacks in Model of Computation

Poor assumptions

- We assumed that each basic operation takes constant time, i.e. model allows adding, multiplying, comparing etc. any two numbers of any length in constant time
- Addition of two numbers takes a unit time!
  - Not good because numbers may be arbitrarily
- Addition and multiplication both take unit time!
  - Again very bad assumption

Model of Computation not so Bad

Finally what about Our Model?

- But with all these weaknesses, our model is not so bad because we have to give the

https://studyc.info/

  - Comparison not the absolute analysis of any algorithm.
  - We have to deal with large inputs not with the small size
- Model seems to work well describing computational power of modern nonparallel machines

Can we do Exact Measure of Efficiency?

- Exact, not asymptotic, measure of efficiency can be sometimes computed but it usually requires certain assumptions concerning implementation

Summary: Computational Model

- Analysis will be performed with respect to this computational model for comparison of algorithms
- We will give asymptotic analysis not detailed comparison i.e. for large inputs
- We will use generic uni-processor random-access machine (RAM) in analysis
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time, except, of course, function calls

https://studyc.info/

Lecture 02 Mathematical Tools for Design and Analysis of Algorithms

Set: A set is well defined collection of objects, which are unordered, distinct, have same type and possess with common properties

Notation:

Elements of set are listed between a pair of curly braces

S1 = {R, R, R, B, G} = {R, B, G} = {B, G, R}

Empty Set

S3 = { } = ∅, has not elements, called empty set

Representation of Sets

Three ways to represent a set

- Descriptive Form
- Tabular form
- Set Builder Form (Set Comprehension)

Example

- Descriptive Form: S = set of all prime numbers
- Tabular form: {2, 3, 5, ...}
- Set Builder Form: $\{x : N \mid (\forall i \in \{2, 3, \ldots, x-1\} \bullet \neg (i / x)) \bullet x\}$

Set Comprehension

Some More Examples

$\{x : s \mid p \bullet x\} = \{x : s \mid p\}$ = all x in s that satisfy p

1. $\{x : Z \mid x^2 = x \bullet x\} = \{0, 1\}$
2. $\{x : N \mid x \equiv 0 \bmod 2 \bullet x\} = \{0, 2, 4, \ldots\}$
3. $\{x : N \mid x \equiv 1 \bmod 2 \bullet x\} = \{1, 3, 5, \ldots\}$
4. $\{x : Z \mid x \geq 0 \wedge x \leq 6 \bullet x\} = \{0, 1, 2, 3, 4, 5, 6\}$
5. $\{x : Z \mid x \geq 0 \wedge x \leq 6 \bullet x^2\} = \{0, 1, 4, \ldots, 25, 36\}$
6. $\{x : N \mid x \equiv 1 \bmod 2 \bullet x^3\} = \{1, 27, 125, \ldots\}$

All collections are not sets

- The prime numbers

  Primes == {2, 3, 5, 7, . . . }
- The four oceans of the world

  Oceans == {Atlantic, Arctic, Indian, Pacific}
- The passwords that may be generated using eight lower-case letters, when repetition is allowed
- Hard working students in MSCS class session 2007-09 at Virtual University
- Intelligent students in your class
- Kind teachers at VU

Operators Over Sets

Membership Operator

https://studyc.info/

- If an element e is a member of set S then it is denoted as $e \in S$ and read e is in S. Let S is sub-collection of X

  X = a set

  $S \subseteq X$

Now

$\in : X \times P\, X \rightarrow Bool$

$\in (x, S) == 1$ if x is in S

0 if x is not in S

Subset:

If each element of A is also in B, then A is said to be a subset of B, $A \subseteq B$ and B is superset of A, $B \supseteq A$.

Let X = a universal set, $A \subseteq X, B \subseteq X$, Now

$\subseteq : PX \times PX \rightarrow Bool$

$\subseteq (A, B) == 1$, if $\forall\, x : X, x \in A \Rightarrow x \in B$

0 if $\exists\, x : X, x \in A \Rightarrow x \notin B$

Operators Over Sets

Intersection

$\cap : PX \times PX \rightarrow PX$

$\cap (A, B) == \{x : X \mid x \in A \text{ and } x \in B \bullet x\}$

Union

$\cup : PX \times PX \rightarrow PX$

$\cup (A, B) == \{x : X \mid x \in A \text{ or } x \in B \bullet x\}$

Set Difference

$\setminus : PX \times PX \rightarrow PX$

$\setminus (A, B) == \{x : X \mid x \in A \text{ but } x \notin B \bullet x\}$

Cardinality and Power Set of a given Set

- A set, S, is *finite* if there is an integer n such that the elements of S can be placed in a one-to-one correspondence with {1, 2, 3, …, n}, and we say that cardinality is n. We write as: |S| = n

Power Set

- How many distinct subsets does a finite set on n elements have? There are 2^n subsets.
- How many distinct subsets of cardinality *k* does a finite set of n elements have?

There are

$$C(n,k) = {}^nC_k = \binom{n}{k} = \frac{n!}{(n-k)!k!}$$

Partitioning of a set

A partition of a set A is a set of non-empty subsets of A such that every element *x* in A exactly belong to one of these subsets.

https://studyc.info/

Equivalently, set P of subsets of A, is a partition of A

1. If no element of P is empty
2. The union of the elements of P is equal to A. We say the elements of P cover A.
3. The intersection of any two elements of P is empty. That means the elements of P are pair wise disjoints

Partitioning of a set

A partition of a set A is a set of non-empty subsets of A such that every element *x* in A exactly belong to one of these subsets.

Equivalently, set P of subsets of A, is a partition of A

4. If no element of P is empty
5. The union of the elements of P is equal to A. We say the elements of P cover A.
6. The intersection of any two elements of P is empty. That means the elements of P are pair wise disjoints

Mathematical Way of Defining Partitioning

A partition P of a set A is a collection $\{A_1, A_2, \ldots, A_n\}$ such that following are satisfied

1. $\forall A_i \in P, A_i \neq \emptyset$,
2. $A_i \cap A_j = \emptyset, \forall i, j \in \{1, 2, \ldots, n\}$ and $i \neq j$
3. $A = A_1 \cup A_2 \cup \ldots \cup A_n$

Example: Partitioning of Z Modulo 4

$\{x : Z \mid x \equiv 0 \bmod 4\} = \{\ldots -8, -4, 0, 4, 8, \ldots\} = [0]$

$\{x : Z \mid x \equiv 1 \bmod 4\} = \{\ldots -7, -3, 1, 5, 9, \ldots\} = [1]$

$\{x : Z \mid x \equiv 2 \bmod 4\} = \{\ldots -6, -2, 2, 6, 10, \ldots\} = [2]$

$\{x : Z \mid x \equiv 3 \bmod 4\} = \{\ldots -5, -1, 3, 7, 11, \ldots\} = [3]$

$\{x : Z \mid x \equiv 4 \bmod 4\} = \{\ldots, -8, -4, 0, 4, 8, \ldots\} = [4]$

Sequence

- It is sometimes necessary to record the order in which objects are arranged, e.g.,
  - Data may be indexed by an ordered collection of keys
  - Messages may be stored in order of arrival
  - Tasks may be performed in order of importance.
  - Names can be sorted in order of alphabets etc.

Definition

- A group of elements in a specified order is called a sequence.
- A sequence can have repeated elements.
- Notation: Sequence is defined by listing elements in order, enclosed in parentheses. e.g.

  S = (a, b, c), T = (b, c, a), U = (a, a, b, c)
- Sequence is a set

  S = {(1, a), (2, b), (3, c)} = {(3, c), (2, b), (1, a)}
- Permutation: If all elements of a finite sequence are distinct, that sequence is said to be a permutation of the finite set consisting of the same elements.

https://studyc.info/

- No. of Permutations: If a set has n elements, then there are n! distinct permutations over it.

Operators over Sequences

- Operators are for manipulations

Concatenation

- (a, b, c) ⌢ (d, a) = (a, b, c, d, a)

Other operators

- Extraction of information: (a, b, c, d, a)(2) = b
- Filter Operator: {a, d} ⨡ (a, b, c, d, a) = (a, d, a)

Note:

- We can think how resulting theory of sequences falls within our existing theory of sets
- And how operators in set theory can be used in case of sequences

Tuples and Cross Product

- A tuple is a finite sequence.
    - Ordered pair (x, y), triple (x, y, z), quintuple
    - A k-tuple is a tuple of k elements.

Construction to ordered pairs

- The cross product of two sets, say A and B, is
  $A \times B = \{(x, y) \mid x \in A, y \in B\}$

  $| A \times B | = |A| \, |B|$
- Some times, A and B are of same set, e.g.,
  $Z \times Z$, where Z denotes set of Integers

Binary Relations

Definition: If X and Y are two non-empty sets, then

$X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$

Example: If X = {a, b}, Y = {0,1} Then

$X \times Y = \{(a, 0), (a, 1), (b, 0), (b, 1)\}$

Definition: A subset of X x Y is a relation over X x Y

Example: Compute all relations over X × Y, where

X = {a, b}, Y = {0,1}

$R_1 = 0$, $R_2 = \{(a, 0)\}$, $R_3 = \{(a, 1)\}$

$R_4 = \{(b, 0)\}$, $R_5 = \{(b, 1)\}$, $R_6 = \{(a, 0), (b, 0)\}, \ldots$

There will be $2^4 = 16$ number of relations

Equivalence Relation

A relation $R \subseteq X \times X$, is

Reflexive: $(x, x) \in R, \forall x \in X$

Symmetric: $(x, y) \in R \Rightarrow (y, x) \in R, \forall x, y \in X$

Transitive: $(x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$

Equivalence: If reflexive, symmetric and transitive

https://studyc.info/

Applications : Order Pairs in Terms of String

Definition

- A relation R over X, Y, Z is some subset of X x Y x Z and so on

Example 1

- If Σ = {0, 1}, then construct set of all strings of length 2 and 3
- Set of length 2 = Σ x Σ = {0,1} x {0,1} = {(0,0), (0,1), (1,0), (1,1)} = {00, 01, 10, 11}
- Set of length 3 = Σ x Σ x Σ = {0, 1} x {0, 1} x {0,1}

= {(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)}

= {000, 010, 100, 110, 001, 011, 101, 111}

Example 2

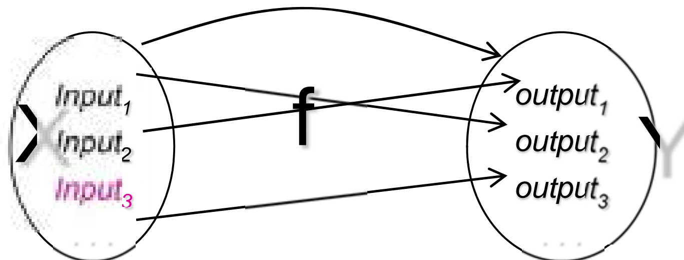
- If Σ = {0, 1}, then construct set of all strings of length ≤ 3
- Construction = {} ∪ Σ ∪ Σ x Σ ∪ Σ x Σ x Σ

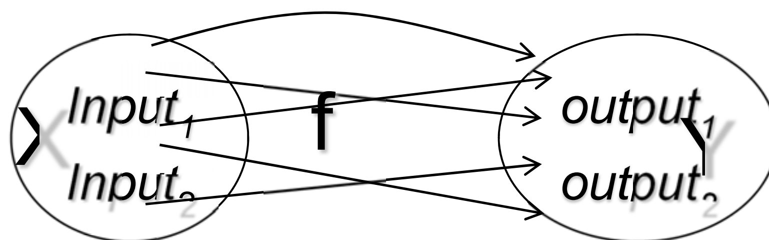
Similarly we can construct collection of all sets of length n

Partial Function: A partial function is a relation that maps each element of X to at most one element of Y. X↛Y denotes the set of all partial functions.

Function: If each element of X is related to unique element of Y then partial function is a total function denoted by X ↛ Y.

Following are not good algorithms.





Lecture 03 Logic and Proving Techniques

Tools used for proving algorithms

- Propositional Logic
- Predicate Logic
- Proofs using
    - Truth Tables
    - Logical Equivalences
    - Counter Example
    - Contradiction
    - Rule of Inference
- Probability as Analysis Tool
- Series and Summation etc.

Propositional and Predicate Logic

Logical Connectives

- **Proposition:** Simplest statements also called atomic formula
- Propositions may be connected based on atomic formula.
- Logical connectives, in descending order of operator precedence

Symbol	Name	Pronunciation
$\neg$	negation	not
$\wedge$	conjunction	and
$\vee$	disjunction	or
$\Rightarrow$	implication	implies
$\Leftrightarrow$	equivalence	if and only if

Negation: The negation of p is true if p is false and vice versa.

Conjunction

- The conjunction $p \wedge q$ is true only if p and q both are true otherwise false
- The conjunction follows the commutative property, i.e. $p \wedge q = q \wedge p$

Disjunction

- The disjunction $p \vee q$ is false if both p and q are false otherwise true
- The disjunction follows the commutative property as well, i.e. $p \vee q = q \vee p$

Implication

- The p is antecedent and q is consequent
- The antecedent is stronger than consequent.
- Commutative property does not hold, i.e. $(p \Rightarrow q) \neq (q \Rightarrow p)$

https://studyc.info/

p	**q**	**p ⇒ q**	**q ⇒ p**	**¬p ∨ q**
t	t	t	t	t
t	f	f	t	f
f	t	t	f	t
f	f	t	t	t

Bi-implication

The equivalence $p \Leftrightarrow q$ means $p \Rightarrow q$ & $q \Rightarrow p$

Commutative property does hold, i.e. $(p \Leftrightarrow q) = (q \Leftrightarrow p)$

p	**q**	**p ⇒ q**	**q ⇒ p**	**p ⇒ q & q ⇒ p**
t	t	t	t	t
t	f	f	t	f
f	t	t	f	f
f	f	t	t	t

Predicates and Quantifiers

Predicate: $P(x) \equiv x < 5$

Example: $\forall\ x : N \mid x^2 = x \bullet x < 2$

For all quantifier

- $\forall$ x, P(x) is true $\Leftrightarrow$ P(x) is true for all x.

Existential Quantifier

- $\exists$ x, P(x) is true $\Leftrightarrow$ P(x) is true for some value of x.

Logical Equivalences

- $\forall$ x, P(x) is logically equivalent to $\neg(\exists x, \neg P(x))$
- $\exists$ x, P(x) is logically equivalent to $\neg(\forall x, \neg P(x))$
- $\forall x, (P(x) \Rightarrow Q(x))$ means $\forall x, \neg P(x) \vee Q(x)$

Proving Techniques

Proof using Truth Table: $(p \wedge q \Rightarrow r) \Rightarrow (p \Rightarrow (q \Rightarrow r))$

https://studyc.info/

p	q	r	$(p \wedge q \Rightarrow r) \Rightarrow (p \Rightarrow (q \Rightarrow r))$
t	t	t	t t t t t t
t	t	f	t f t f f t
t	f	t	f t t t t t
t	f	f	f t t t t t
f	t	t	f t t t t t
f	t	f	f t t t f t
f	f	t	f t t t t t
f	f	f	f t t t t t

De Morgan's Laws

1. $\neg(p \wedge q) = \neg p \vee \neg q$

p	q	$p \wedge q$	$\neg(p \wedge q)$	$\neg p$	$\neg q$	$\neg p \vee \neg q$
t	t	t	f	f	f	f
t	f	f	t	f	t	t
f	t	f	t	t	f	t
f	f	f	t	t	t	t

2. $\neg(p \vee q) = \neg p \wedge \neg q$

p	q	$p \vee q$	$\neg(p \vee q)$	$\neg p$	$\neg q$	$\neg p \wedge \neg q$
t	t	t	f	f	f	f
t	f	t	f	f	t	f
f	t	t	f	t	f	f
f	f	f	t	t	t	t

Proof using Counter Example, Contraposition

Counter Example

To prove $\forall x\, (A(x) \Rightarrow B(x))$ is false, we show some object x for which A(x) is true and B(x) is false.

Proof

$\neg (\forall x\, (A(x) \Rightarrow B(x))) \Leftrightarrow$

$\exists x, \neg(A(x) \Rightarrow B(x))) \Leftrightarrow$

$\exists x, \neg(\neg A(x) \vee B(x)) \Leftrightarrow$

$\exists x, A(x) \wedge \neg B(x))$

https://studyc.info/

Contraposition

To prove $A \Rightarrow B$, we show $(\neg B) \Rightarrow (\neg A)$

- x is divisible by 4 $\Rightarrow$ x is divisible by 2 $\Leftrightarrow$

x is not divisible by 2 $\Rightarrow$ x is not divisible by 4 $\Leftrightarrow$

Proof by Contradiction

Contradiction

To prove $A \Rightarrow B$,

Steps in Proof

- We assume A and to prove that B
- On contrary suppose that $\neg$ B and
- Then prove B, it will be contradiction

Further analysis

- $A \Rightarrow B \;\; \Leftrightarrow \;\; (A \wedge \neg B) \Rightarrow B$ Contradiction
- $A \Rightarrow B \;\; \Leftrightarrow \;\; (A \wedge \neg B)$ is false
- Assuming $(A \wedge \neg B)$ is true,
  and discover a contradiction (such as $A \wedge \neg A$),
  then conclude $(A \wedge \neg B)$ is false, and so $A \Rightarrow B$.

Problem: Proof by Contradiction

Prove:

$[B \wedge (B \Rightarrow C)] \Rightarrow C$, by contradiction

Proof:

Suppose $[B \wedge (B \Rightarrow C)]$, to prove C

On contrary, assume $\neg C$

$\neg C \wedge [B \wedge (B \Rightarrow C)]$ must be true

$\Rightarrow \neg C \wedge [B \wedge (\neg B \vee C)]$

$\Rightarrow \neg C \wedge [(B \wedge \neg B) \vee (B \wedge C)]$

$\Rightarrow \neg C \wedge [f \vee (B \wedge C)]$

$\Rightarrow \neg C \wedge B \wedge C = \neg C \wedge C \wedge B = f \wedge B = f$

$\Rightarrow$ False, Contradiction $\Rightarrow$ C

Rules of Inference

A rule of inference is a general pattern that allows us to draw some new conclusion from a set of given statements. If we know P then we can conclude Q.

Modus ponens

If $\{B \wedge (B \Rightarrow C)\}$ then $\{C\}$

Proof:

Suppose $B \wedge (B \Rightarrow C)$ then

B

https://studyc.info/

$B \Rightarrow C$

Syllogism

If $\{A \Rightarrow B \wedge B \Rightarrow C\}$ then $\{A \Rightarrow C\}$

Proof

- Suppose $A \Rightarrow B \wedge B \Rightarrow C$, To prove $A \Rightarrow C$
- B
- C

Rule of cases

If $\{B \Rightarrow C \wedge \neg B \Rightarrow C\}$ then $\{C\}$

B, true, implies C true

$\neg$ B, true, implies C true

Two Valued Boolean Logic

1. Boolean values = B = {0, 1}, there are two binary operations:

- $+$ = or = $\vee$
- $\cdot$ = and = $\wedge$

2. Closure properties:

- $\forall x, y \in B, x + y \in B$
- $\forall x, y \in B, x . y \in B$

3. Identity element:

- $x + 0 = 0 + x = x$
- $x \cdot 1 = 1 . x = x$

4. Commutative:

- $x + y = y + x$
- $x \cdot y = y \cdot x$

5. Distributive:

- $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
- $x + (y \cdot z) = (x + y) \cdot (x + z)$

6. Complement:

- $\forall x \in B, \exists x' \in B$ such that

$x + x' = 1, x \cdot x' = 0$

Tautologies and Truth Table

Tautology: Any statement which is always true is called a tautology

Example

Show $[B \wedge (B \Rightarrow C)] \Rightarrow C$ is a tautology:

Proof

B	C	$(B \Rightarrow C)$	$(B \wedge (B \Rightarrow C))$	$(B \wedge (B \Rightarrow C)) \Rightarrow C$
0	0	1	0	1
0	1	1	0	1

https://studyc.info/

1	0	0	0	1
1	1	1	1	

For every assignment for B and C, the statement is True, hence the above statement is a tautology.

Probability as Analysis Tool

Elementary events

- Suppose that in a given situation an event, or an experiment, may have any one, and only one, of k outcomes, $s_1, s_2, \ldots, s_k$. Assume that all these outcomes are mutually exclusive.

Universe

The set of all elementary events is called the universe of discourse and is denoted

$U = \{s_1, s_2, \ldots, s_k\}$.

Probability of an outcome s_i

- Associate a real number $Pr(s_i)$, such that

$0 \le Pr(s_i) \le 1$ for $1 \le i \le k$;

$Pr(s_1) + Pr(s_2) + \ldots + Pr(s_k) = 1$

Event

- Let $S \subseteq U$. Then S is called an event, and $\Pr(S) = \sum_{s_i \in S} \Pr(s_i)$

Sure event

- $U = \{s_1, s_2, \ldots, s_k\}$, if S = U $\Pr(S) = \sum_{s_i \in S} \Pr(s_i) = 1$

Impossible event

- $S = \varnothing$, $Pr(\varnothing) = 0$

Arithmetic and Geometric Series

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{2n^3 + 3n^2 + n}{6}$$

$$\sum_{k=1}^{n} i^k = i + i^2 + \ldots + i^n = \frac{i(i^n - 1)}{i - 1}$$

$$\sum_{i=1}^{n} 2^i = 2^n - 1$$

$$\sum_{i=1}^{k} i2^i = \left(k-1\right)2^{k+1} + 2$$

https://studyc.info/

Lecture 04 Mathematical Induction

What is Mathematical Induction?

- Mathematical induction is a powerful, yet straight-forward method of proving statements whose domain is a subset of the set of integers.
- Usually, a statement that is proven by induction is based on the set of natural numbers.
- This statement can often be thought of as a function of a number n, where n = 1, 2, 3,. . .
- Proof by induction involves three main steps
  - Proving the base of induction
  - Forming the induction hypothesis
  - Proving that the induction hypothesis holds true for all numbers in the domain.

Let P(n) be the predicate defined for any positive integers n, and let n_0 be a fixed integer. Suppose the following two statements are true

1. $P(n_0)$ is true.
2. For any positive integers k, $k \geq n_0$, if P(k) is true then P(k+1)is true.

If both of the above statements are true then the statement:

$\forall$ n $\in$ N, such that $n \geq n_0$, P(n) is also true

Steps in Proving by Induction

Claim: P(n) is true for all $n \in Z^+$, for $n \geq n_0$

1. Basis
   - Show formula is true when $n = n_0$
2. Inductive hypothesis
   - Assume formula is true for an arbitrary n = k

     where, $k \in Z_+$ and $k \geq n_0$
3. To Prove Claim
   - Show that formula is then true for k+1

Note: In fact we have to prove

1) $P(n_0)$ and
2) $P(k) \Rightarrow P(k+1)$

Mathematical Way of Expressing Induction

- *Basis step.*

  Show that proposition $P(1)$ is true.
- *Inductive step.*

  Show that for every positive integer n, the implication $P(n) \to P(n+1)$ is true.

  $P(n)$ for a fixed n is called **inductive hypothesis.**
- $[P(1) \wedge \forall n, (P(n) \to P(n+1))] \to \forall n, P(n)$

Definition (Well-Ordering Principle)

https://studyc.info/

- The Well-ordering Principle is the following statement
  "*every nonempty set of positive integers contains a least element*"
- In a mathematical way we can define this Principle as:

  there is a in S such that $a \leq b$ for all b in S i.e.

  $\exists\ a \in S$, such that $a \leq b,\ \forall\ b \in S$
- And we say that set S is *well-ordered* with respect to $\leq$.

Modus Ponens Principal

$$p \Rightarrow q$$

$$p$$

Hence, q

Why Mathematical Induction is Valid?

- Let us suppose that P(1) is true, and that

  $\forall n\ (P(n) \rightarrow P(n+1))$ is also true.
- Claim: $\forall n\ P(n)$ is true
  - Assume proposition $\forall\ n,\ P(n)$ is false, i. e, there are some positive integers for which $P(n)$ is false.
  - Let S be the set of those n's. By well-ordering property, S has a least element, suppose, k.
  - As $1 \notin S$, *so* $1 < k$, *s*o k-1 is a positive
  - Since $k-1 < k$, hence $k-1 \notin S$. So $P(k-1)$ is true.
  - By modus ponens, $P((k-1) + 1) = P(k)$ is true.
  - Contradiction, hence $\forall n,\ P(n)$

Another Reason for Validity?

Basis Step

First suppose that we have a proof of P(0).

Inductive Hypothesis

$\forall\ k > 0, \quad P(k) \Rightarrow P(k + 1)$

How it is proved $\forall\ n > 0$?

$P(0) \Rightarrow P(1)$

$P(1) \Rightarrow P(2)$

$P(2) \Rightarrow P(3)$

. . .

Iterating gives a proof of $\forall$ n, P(n). This is another way of proving validity of mathematical Induction.

Example 1:

Prove that $n^2 \geq n + 100 \qquad \forall\ n \geq 11$

Solution

https://studyc.info/

Initially, base case

Solution set = {11}

By, P(k) $\Rightarrow$ P(K+1) - P(11) $\Rightarrow$ P(12), taking k = 11

Solution set = {11, 12}

Similarly, P(12) $\Rightarrow$ P(13), taking k = 12

Solution set = {11, 12, 13}

And, P(13) $\Rightarrow$ P(14), taking k = 13

Solution set = {11, 12, 13, 14}

And so on

Example 2:

Use Mathematical Induction to prove that sum of the first n odd positive integers is n^2

Proof:

Let $P(n)$ denote the proposition that $\sum_{i=1}^{n}(2i-1) = n^2$

Basis step : $P(1)$ is true , since $1 = 1^2$

Inductive step : Let $P(k)$ is true for a positive integer k, i.e., $1+3+5+\ldots+(2k-1) = k^2$

- Note that: $1+3+5+\ldots+(2k-1)+(2k+1) = k^2+2k+1= (k+1)^2$

$\therefore P(k+1)$ true, by induction, $P(n)$ is true for all $n \in Z^+$

Another Proof: $\sum_{i=1}^{n}(2i-1) = 2\sum_{i=1}^{n} i - n = n(n+1) - n = n^2$

Example 3:

Use mathematical Induction to prove that the inequality $n < 2^n$ for all $n \in Z^+$

Proof:

Let $P(n)$ be the proposition that $n < 2^n$

Basis step : $P(1)$ is true since $1 < 2^1$.

Inductive step :

Assume that $P(n)$ is true for a positive integer $n = k$,
i.e., $k < 2^k$.

Now consider for $P(k+1)$:

Since, $k + 1 < 2^k + 1 \leq 2^k + 2^k = 2.2^k = 2^{k+1}$

$\therefore P(k+1)$ is true.

It proves that $P(n)$ is true for all $n \in Z^+$.

https://studyc.info/

Harmonic Numbers:

Now consider

$$H_{2^{k+1}} = 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{2^k}+\frac{1}{2^k+1}+\frac{1}{2^k+2}+\cdots+\frac{1}{2^{k+1}=2^k+2^k}$$

$$= H_{2^k}+\frac{1}{2^k+1}+\frac{1}{2^k+2}+\cdots+\frac{1}{2^{k+1}}$$

$$\geq (1+\frac{k}{2})+\frac{1}{2^k+1}+\frac{1}{2^k+2}+\cdots+\frac{1}{2^{k+1}}$$

$$\geq (1+\frac{k}{2})+\frac{1}{2^k+2^k}+\frac{1}{2^k+2^k}+\cdots+\frac{1}{2^k+2^k}$$

$$= (1+\frac{k}{2})+\frac{2^k}{2^k+2^k} = 1+\frac{k}{2}+\frac{1}{2} = 1+\frac{k+1}{2}$$

$\therefore P(k+1)$ is true.
Hence the statement is true for all $n \in Z^+$.

Strong Mathematical Induction

Let P(n) be a predicate defined for integers n, and a and b are fixed integers with $a \leq b$.
Suppose the following statements are true:

1. P(a), P(a + 1), … , P(b) are all true

(basis step)

2. For any integer k > b,

if P(i) is true for all integers i with a ≤ i < k,

then P(k) is true. (inductive step)

Then P(n) is true for all integers n ≥ a.

Example 1: Divisibility by a Prime

Theorem:

- For any integer n ≥ 2, n is divisible by a prime.

Proof

(by strong mathematical induction):

- Basis step:

The statement is true for n = 2. This is because 2 | 2 and 2 is a prime number.

Inductive step:

Assume the statement is true for all i with 2 ≤ i <k (inductive hypothesis) ;

To show that it is true for k .

- We know that ∀ i ∈ **Z,** with 2 ≤ i < k, P(i), i.e. i is divisible by a prime number. **(1)**
- Now we show P(k), i.e., k is divisible by a prime.

Take two cases:

Case 1: k is prime.

https://studyc.info/

Then k is divisible by itself. And nothing to prove

Case 2: k is composite.

Then $k = a \cdot b$, *where* $2 \le a < k$ and $2 \le b < k$

Based on (1), p|a for some prime p. (2)

Based on Case 2, a|k (3)

By transitivity, p|a and a|k $\Rightarrow$ p|k

Thus, P(n) is true by strong induction.

Example 2: Another Example in Number Theory

If $n \in Z$, $n > 1$, then n can be written as product of primes.

Proof :

Let P(n) ≡ n can be written as the product of primes.

Basis : $P(2)$ is true, since 2 is the first prime number

Inductive : Assume that the statement is true for n = k, i.e.

$P(2), P(3), \ldots, P(k)$ can be written as product of primes.

Prove that: true for n = k, i.e. $P(k + 1)$ is product of primes.

Case 1 : $k + 1$ is prime, then nothing to prove

Case 2 : $k + 1$ is composite, then

$k + 1 = xy$, where $2 \le x \le y < k+1$

Inductive hypothesis, a and b are product of primes.

Hence P(k+1) can be written as product of primes.

Any Amount Limited Coins: More Steps in Basis

Statement

Show that any amount in cents ≥ 8 cents can be obtained using 3 cents and 5 cents coins only.

Proof

We have to prove that, amount = 3.m + 5.n, m ≥ 0, n ≥ 0

Basis Step

This time check for a five particular values:

8 = 1.3 + 1.5

9 = 3.3

10 = 2.5

11 = 2.3 + 1.5

12 = 4.3

Now we generalize it?

Let P(n) be the statement that:

"n cents can be obtained using 3 and 5 cents".

Inductive Hypothesis

We want to show that

P(k) is true $\Rightarrow$ P(k+1), $\forall$ k ≥ 8

There are two cases now

Case 1

P(k) is true and k cents contain at least one 5 coin.

Case 2

P(k) true, k cents do not contain any coin of 5 cent.

https://studyc.info/

Case 1

P(k) is true and k cents contain at least one 5 coin.

Since P(k) is true $\quad k \geq 8$

Hence k can be expressed as

$k = 3.m + 5.n \quad m \geq 0$ and $n \geq 1$

$k + 1= 3.m + 5.n + 1$

$k + 1= 3.m + 5.(n - 1) + 1 + 5$

$k + 1= 3.(m + 2) + 5.(n - 1), \quad m \geq 2$ and $n \geq 0$

Hence the statement is true for $n = k + 1$

Case 2

- P(k) is true and k cents do not contain any coin of 5 cent. for $k \geq 8$

Hence k can be expressed as

$k = 3.m \quad m \geq 3$

$k + 1= 3.(m – 3) + 9 + 1$

$k + 1= 3.(m – 3) + 2.5$

$k + 1= 3.m' + 5.n \quad m' \geq 0$ and $n = 2$

Hence the statement is true for $n = k + 1$

Hence $P(k + 1)$ is true

https://studyc.info/

Lecture 05 Strong Mathematical Induction

Generalized Demargon's Law by Induction

Prove $\overline{\bigcap_{j=1}^{n} A_j} = \bigcup_{j=1}^{n} \overline{A_j}$ when $n \geq 2$, i.e., $\left(\overline{A_1 \cap A_2 \cap \ldots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \ldots \cup \overline{A_n}\right)$

Proof:

Basis step: Since, $\left(\overline{A_1 \cap A_2} = \overline{A_1} \cup \overline{A_2}\right)$ true for n = 2

Induction Step: Assume the result is true n = k and then prove for n = k+1

$$\overline{\bigcap_{j=1}^{k+1} A_j} = \overline{\bigcap_{j=1}^{k} A_j \cap A_{k+1}}$$

$$= \overline{\bigcap_{j=1}^{k} A_j} \cup \overline{A_{k+1}}$$

$$= \bigcup_{j=1}^{k} \overline{A_j} \cup \overline{A_{k+1}} \quad \text{(by induction hypothesis)}$$

$$= \bigcup_{j=1}^{k+1} \overline{A_j}$$

Postage Ticket: Again More Steps in Basis

Prove that postage ticket of amount ≥ 12 cents can be formed using only 4 cent and 5 cent stamps.

Proof

Let $P(n) \equiv n$ cents can be formed using only 4 and 5 cent

$P(n) \equiv n = 4s + 5t \quad s \geq 0$, and $t \geq 0 \forall\ n \geq 12$

Basis : $P(12)$ is true, since $12 = 4 \times 3$;

$P(13)$ is true, since $13 = 4 \times 2 + 5 \times 1$;

$P(14)$ is true, since $14 = 4 \times 1 + 5 \times 2$;

$P(15)$ is true, since $15 = 5 \times 3$;

Inductive : Assume $P(12), P(13), \ldots, P(k)$ are true.

Now prove for $P(k + 1)$

Suppose $k\text{-}3 = 4 \times s + 5 \times t$.

Then $k + 1 = 4 \times (s + 1) + 5 \times t$. true for n = k + 1.

By Strong Induction, $P(n)$ is true if $n \in \mathbb{Z}$ and $n \geq 12$.

Proving a Property of a Sequence

Proposition:

Suppose $a_0, a_1, a_2, \ldots$ is defined as follows:

$a_0 = 1,\ a_1 = 2,\ a_2 = 3,$

$a_k = a_{k-1} + a_{k-2} + a_{k-3}$ for all integers $k \geq 3$.

Then $a_n \leq 2_n$ for all integers $n \geq 0$. P(n)

https://studyc.info/

Proof (*by strong induction*)

Basis step:

The statement is true

for n = 0: $a_0 = 1 \le 1 = 2_0$ P(0)

for n = 1: $a_1 = 2 \le 2 = 2_1$ P(1)

for n = 2: $a_2 = 3 \le 4 = 2_2$ P(2)

Inductive step:

For any $k > 2$, assume P(i) is true for all i with $0 \le i < k$, i.e., $a_i \le 2^i$ for all $0 \le i < k$ **(1)**

Show that

P(k) is true: $a_k \le 2_k$ **(2)**

Now consider

$a_k = a_{k-1} + a_{k-2} + a_{k-3}$

$\le 2^{k-1} + 2^{k-2} + 2^{k-3}$ *based on (1)*

$\le 2^0 + 2^1 + \ldots + 2^{k-3} + 2^{k-2} + 2^{k-1}$

$= 2^k - 1 \le 2^k$

Thus, P(n) is true by strong mathematical induction.

Hence it proves the result

Existence of Binary Integer Representation

Theorem

Given any positive integer n, there exists a unique representation of n in the form:

$$n = c_r \cdot 2^r + c_{r-1} \cdot 2^{r-1} + \ldots + c_1 \cdot 2^1 + c_0$$

where r is non-negative integer, $c_r = 1$, and $c_j = 0$ or 1, $\forall$ j = 0, 1, 2, . . . , r-1

Proof (*by strong induction*)

Let P(n) be the statement that n can be written in the form

$$n = c_r \cdot 2^r + c_{r-1} \cdot 2^{r-1} + \ldots + c_1 \cdot 2^1 + c_0$$

Basis step:

If n = 1, then $n = c_r.2_r = c_0$, where r = 0, and $c_0 = 1$

Hence the statement is true for n = 1, i.e. P(1) is true

Inductive Hypothesis:

Let us suppose that statement is true for all i, $1 \le i < k$,

$i = c_k.2^k + c_{k-1}.2^{k-1} + \ldots + c_1.2^1 + c_0$

$c_r .= 1$, and $c_i = 0$ or 1, $\forall$ j = 0, 1, 2, . . . , r-1

Show that

https://studyc.info/

Now we prove that statement is true for k

Case 1

Suppose k is even, k/2 is an integer and k/2 < k, hence

$k/2 = c_r.2^r + c_{r-1}.2^{r-1} + \ldots + c_1.2^1 + c_0$

where r is non-negative integer and

$c_r = 1$, and $c_i = 0$ or 1, $\forall$ j = 0, 1, 2, . . . , r-1

$k = 2.c_r.2^r + 2.c_{r-1}.2^{r-1} + \ldots + 2.c_1.2^1 + 2.c_0$

$k = cr.2^{r+1} + c_{r-1}.2_r + \ldots + c_1.2_2 + c_0.2^1$, true

which is the required form

Case 2

Let k ≥ 3, is odd, (k-1)/2 is an integer and 1 ≤ (k-1)/2 < k,

$(k-1)/2 = c_r.2_r + c_{r-1}.2_{r-1} + \ldots + c_1.2_1 + c_0$

where r is non-negative integer and

$c_r = 1$, and $c_j = 0$ or 1, $\forall$ j = 0, 1, 2, . . . , r-1

Now, $k - 1 = c_r.2^{r+1} + c_{r-1}.2^r + \ldots + c_1.2^2 + c_0.2^1$

And, $k = c_r.2_{r+1} + c_{r-1}.2^r + \ldots + c_1.2^2 + c_0.2^1 + 1$, true

Hence by strong mathematical induction, P(n) is true

Uniqueness

Uniqueness

Now we prove that n has a unique representation

$n = c_r \cdot 2^r + c_{r-1} \cdot 2^{r-1} + \ldots + c_1 \cdot 2^1 + c_0$

where r is non-negative integer, $c_r = 1$, and $c_j = 0$ or 1, $\forall$ j = 0, 1, 2, . . . , r-1

On contrary, suppose that n has two different representations, i.e.

$n = c_r \cdot 2^r + c_{r-1} \cdot 2^{r-1} + \ldots + c_1 \cdot 2^1 + c_0$ (1) and

$n = c_r \cdot 2^r + c_{r-1} \cdot 2^{r-1} + \ldots + c_1 \cdot 2^1 + c_0$ (2)

Now subtract (2) from (1) we get

$0 = (b_r - c_r)2^r + (b_{r-1} - c_{r-1})2^{r-1} + \ldots + (b_1 - c_1)2^1 + (b_0 - c_0) \Leftrightarrow$

$b_r = c_r$, $b_{r-1} = c_{r-1}$, ... , $\mathrm{b}_1 = c_1$, $b_0 = c_0$ proved

More Complicated Example

Problem Let $\mathrm{f}_0(x) = \frac{1}{2-x}$, and $\mathrm{f}_{n+1} = \mathrm{f}_0 \circ \mathrm{f}_n, n \geq 0.$

Find an expression for f_n and prove it by induction

https://studyc.info/

Solution

Since $f_0 = \dfrac{1}{2-x}$ and $f_{n+1} = f_o \circ f_0$ therefore

$$f_1(x) = f_0 \circ f_0(x) = f_0(\frac{1}{2-x}) = \frac{1}{2-\frac{1}{2-x}} = \frac{2-x}{3-2x}$$

And, $f_2(x) = f_0 \circ f_1(x) = f_0(\dfrac{2-x}{3-2x}) = \dfrac{1}{2-\frac{2-x}{3-2x}} = \dfrac{3-2x}{4-3x}$

And, $f_3(x) = f_0 \circ f_2(x) = f_0(\dfrac{3-2x}{4-3x})$

$$= \frac{1}{2-\frac{3-2x}{4-3x}} = \frac{4-3x}{5-4x}$$

And so on

$f_n(x) = f_0 \circ f_{n-1}(x) = f_0(\dfrac{n-(n-1)x}{(n+1)-nx})$

$$= \frac{1}{2-\frac{n-(n-1)x}{(n+1)-nx}} = \frac{(n+1)-nx}{(n+2)-(n+1)x}$$

Now generalized function is

$$f_n(x) = \frac{(n+1)-nx}{(n+2)-(n+1)x}$$

Now we prove this guess by mathematical Induction

Basis case: take n = 0

$f_0 = \dfrac{1}{2-x}$, which is true

Inductive hypothesis: assume that statement is true n = k

$$f_k(x) = \frac{(k+1)-kx}{(k+2)-(k+1)x}$$

Claim: Claim: Now we have to prove that statement is true n = k + 1

$$f_{k+1}(x) = \frac{(k+1+1)-(k+1)x}{(k+1+2)-(k+1+1)x} = \frac{(k+2)-(k+1)x}{(k+3)-(k+2)x}$$

https://studyc.info/

By definition: $f_{n+1} = f_0 \circ f_n \Rightarrow f_{k+1} = f_0 \circ f_k \quad \forall \text{ k} \geq 0$

$$\mathrm{f}_{\mathrm{k+1}}(x) = \mathrm{f}_0\left(\frac{(k+1)-kx}{(k+2)-(k+1)x}\right) = \frac{1}{2-\dfrac{(k+1)-kx}{(k+2)-(k+1)x}}$$

After simplification, $\mathrm{f}_{\mathrm{k+1}}(x) = \dfrac{(k+2)-(k+1)x}{(k+3)-(k+2)x}$, proved.

https://studyc.info/

Lecture 06 Fibonacci Sequences

Fibonacci Sequence

By studying Fibonacci numbers and constructing Fibonacci sequence we can imagine how mathematics is connected to apparently unrelated things in this universe. Even though these numbers were introduced in 1202 in Fibonacci's book *Liber abaci*, but these numbers and sequence are still fascinating and mysterious to people of today. Fibonacci, who was born Leonardo da Pisa gave a problem in his book whose solution was the Fibonacci sequence as we will discuss it today.

Statement:

Start with a pair of rabbits, one male and one female, born on January 1.

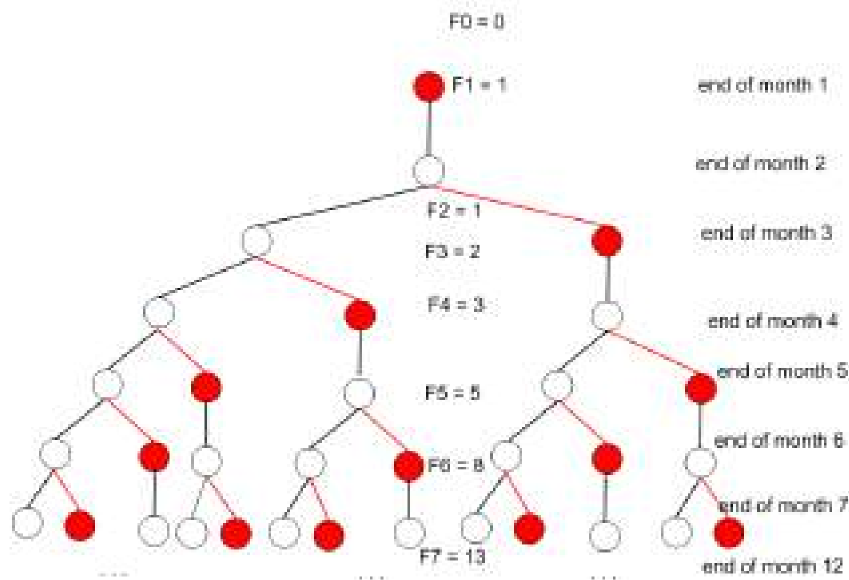
Assume that all months are of equal length and that rabbits begin to produce two months after their own birth. After reaching age of two months, each pair produces another mixed pair, one male and one female, and then another mixed pair each month, and no rabbit dies.

How many pairs of rabbits will there be after one year?

Answer: The Fibonacci Sequence!

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . .

Construction of Mathematical Model



Total pairs at level k = Total pairs at level k-1 + Total pairs born at level k (1)

Since

Total pairs born at level k = Total pairs at level k-2 (2)
Hence by equation (1) and (2)

https://studyc.info/

Total pairs at level k = Total pairs at level k-1 + Total pairs at level k-2
Now let us denote

F_k = Total pairs at level k

Now our recursive mathematical model will become $F_k = F_{k-1} + F_{k-2}$

Computing Values using Mathematical Model

Since $F_k = F_{k-1} + F_{k-2}$ $F_0 = 0, F_1 = 1$
$F_2 = F_1 + F_0 = 1 + 0 = 1$
$F_3 = F_2 + F_1 = 1 + 1 = 2$
$F_4 = F3 + F2 = 2 + 1 = 3$
$F_5 = F_4 + F3 = 3 + 2 = 5$
$F_6 = F_5 + F4 = 5 + 3 = 8$
$F_7 = F_6 + F_5 = 8 + 5 = 13$
$F_8 = F_7 + F_6 = 13 + 8 = 21$
$F_9 = F_8 + F_7 = 21 + 13 = 34$
$F_{10} = F_9 + F_8 = 34 + 21 = 55$
$F_{11} = F_{10} + F_9 = 55 + 34 = 89$
$F_{12} = F_{11} + F_{10} = 89 + 55 = 144 \ldots$

Explicit Formula Computing Fibonacci Numbers

Theorem:
The Fibonacci sequence $F_0, F_1, F_2, \ldots$. Satisfies the recurrence relation

$$F_k = F_{k-1} + F_{k-2} \qquad \forall\ k \geq 2$$

with initial condition $F_0 = F_1 = 1$

Find the explicit formula for this sequence.
Solution:
The given Fibonacci sequence

$$F_k = F_{k-1} + F_{k-2}$$

Let t^k is solution to this, then characteristic equation

$$t^2 - t - 1 = 0$$

$$t = \frac{1 \pm \sqrt{1+4}}{2}$$

$$t_1 = \frac{1+\sqrt{5}}{2}, \quad t_2 = \frac{1-\sqrt{5}}{2}$$

For some real C and D Fibonacci sequence satisfies the relation

https://studyc.info/

$$F_n = C\left(\frac{1+\sqrt{5}}{2}\right)^n + D\left(\frac{1-\sqrt{5}}{2}\right)^n \quad \forall\, n \geq 0$$

$n = 0$

$$F_0 = C\left(\frac{1+\sqrt{5}}{2}\right)^0 + D\left(\frac{1-\sqrt{5}}{2}\right)^0$$

$\Rightarrow \mathrm{F}_0 = C + D$

$\Rightarrow C + D = 0$ ....(1) $\qquad \because F_0 = 0$

Now $n = 1$

$$\mathrm{F}_1 = C\left(\frac{1+\sqrt{5}}{2}\right) + D\left(\frac{1-\sqrt{5}}{2}\right)$$

$\Rightarrow \frac{1+\sqrt{5}}{2}\cdot C + \frac{1-\sqrt{5}}{2}\cdot D = 1$ ....(2) $\qquad \because F_1 = 1$

Solving (1) and (2) simultaneously we get

$\mathrm{C} = \frac{1}{\sqrt{5}}, D = -\frac{1}{\sqrt{5}}$

Hence

$$\mathrm{F_n} = \left(\frac{1}{\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1}{\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2\sqrt{5}}\right)^n$$

After simplifying we get

$$\mathrm{F_n} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

which is called the explicit formula for the Fibonacci sequence recurrence relation.

Let $\Phi = \left(\frac{1+\sqrt{5}}{2}\right)$ and $\ddot{\Phi} = \left(\frac{1-\sqrt{5}}{2}\right)$ then

$$\mathrm{F_n} = \frac{1}{\sqrt{5}}\Phi^{\mathrm{n}} - \frac{1}{\sqrt{5}}\Phi^{\mathrm{n}}$$

Verification of the Explicit Formula

Example: Compute F3.

Since $\mathrm{F_n} = \frac{1}{\sqrt{5}}\Phi^{\mathrm{n}} - \frac{1}{\sqrt{5}}\Phi^{\mathrm{n}}$ where $\Phi = \left(\frac{1+\sqrt{5}}{2}\right)$ and $\ddot{\Phi} = \left(\frac{1-\sqrt{5}}{2}\right)$ then

$$\mathrm{F}_3 = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^3 - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^3$$

Now, $\mathrm{F}_3 = \frac{1}{\sqrt{5}}\left(\frac{1+3.1^2.\sqrt{5}+3.1.5+5\sqrt{5}}{8}\right) - \frac{1}{\sqrt{5}}\left(\frac{1-3.1^2.\sqrt{5}+3.1.5-5\sqrt{5}}{8}\right)$

$$F_3 = \frac{1}{8.\sqrt{5}}\left(1+3.1.\sqrt{5}+3.1.5+5\sqrt{5}\right) - \frac{1}{8.\sqrt{5}}\left(1-3.1.\sqrt{5}+3.1.5-5\sqrt{5}\right)$$

$$F_3 = \frac{1}{8.\sqrt{5}}\left(1+3.1.\sqrt{5}+3.1.5+5\sqrt{5}-1+3.1.\sqrt{5}-3.1.5+5\sqrt{5}\right)$$

$$F_3 = 2$$

Recursive Algorithm Computing Fibonacci Numbers

Fibo-R(n)

if n=0
then 0
if n=1
then 1 } Terminating conditions

else Fibo-R(n – 1)+Fibo-R(n – 2)   Recursive calls

Running Time of Recursive Fibonacci Algorithm

Least Cost: To find an asymptotic bound of computational cost of this algorithm, we can use a simple trick to solve this recurrence containing big oh expressions

Simply drop the big O from the recurrence, solve the recurrence, and put the O back. Our recurrence

$$T(n) = \begin{cases} O(1) & \text{if } n < 2 \\ T(n-1)+T(n-2) & n \geq 2 \end{cases}$$

will be refined to

$$T(n) = \begin{cases} 1 & \text{if } n < 2 \\ T(n-1)+T(n-2) & n \geq 2 \end{cases}$$

Guess that F_{n+1} is the least cost to solve this recurrence. Why this guess?

$\forall\ n \geq 0,\ T(n) \geq F_{n+1}$

then F_{n+1} will be minimum cost for this recurrence

We prove it by mathematical induction

Base Case

There are two base cases

For n = 0, T(0) = 1 and $F_1 = 1$, hence $T(0) \geq F_1$

For n = 1, T(1) = 1 and $F_2 = 1$, hence $T(1) \geq F_2$

Inductive Hypothesis

Let us suppose that statement is true some $k \geq 1$

$T(k) \geq F_{k+1}$, for k =0, 1, 2,. . . and $k \geq 1$

Now we show that statement is true for k + 1

https://studyc.info/

Now, $T(k + 1) = T(k) + T(k - 1)$ By definition on $T(n)$

$T(k + 1) = T(k) + T(k - 1) \geq F_{k+1} + F_k = F_{k+2}$ Assumption

$T(k + 1) \geq F_{k+2}$

Hence the statement is true for k + 1.

We can now say with certainty that running time of this recursive Fibonacci algorithm is at least $\Omega(F_{n+1})$.

Now we have proved that

$T(n) \geq F_{n+1}$, $n \geq 0$ ……….(1)

We already proved in solution to recursive relation that

$$F_n = \frac{1}{\sqrt{5}}\Phi^n - \frac{1}{\sqrt{5}}\Phi^n \quad where\ \Phi = \left(\frac{1+\sqrt{5}}{2}\right) \text{ and } \ddot{\Phi} = \left(\frac{1-\sqrt{5}}{2}\right) \quad ....(2)$$

It can be easily verified that $F_n \geq \Phi^n/5 \geq (3/2)^n$

From the equations (1) and (2), $T(n) \geq F_{n+1} \geq F_n \geq (3/2)^n$

Hence we can conclude that running time of our recursive Fibonacci Algorithm is:

$T(n) = \Omega\ (3/2)^n$

Golden Ratio

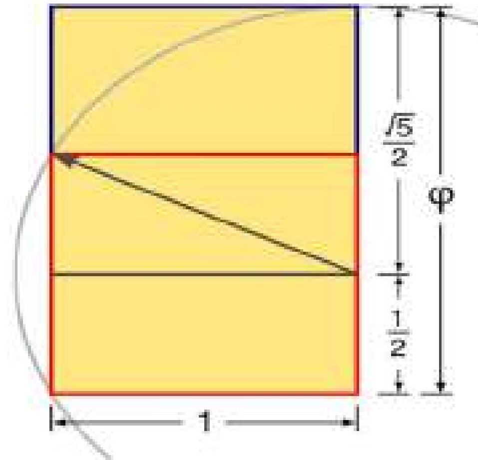
W say that two quantities, x and y, (x < y), are in the golden ratio if the ratio between the sum, x + y, of these quantities and the larger one, y, is the same as the ratio between the larger one, y, and the smaller one x.

$$\frac{x+y}{y} = \frac{y}{x} \approx 1.62$$

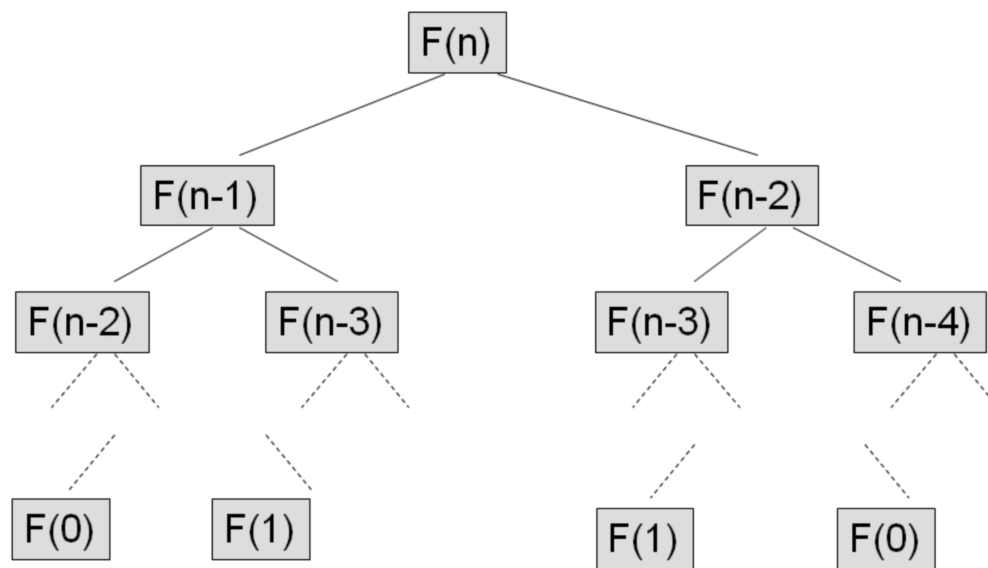
Mathematicians have studied the golden ratio because of its unique and interesting properties.

https://studyc.info/

$$\Phi = \frac{1+\sqrt{5}}{2} \approx 1.62$$
$x = \Phi - 1 \approx 0.62,$
$y = 1$
of course x < y and
$\frac{x+y}{y} = \frac{y}{x}$ i.e. $\frac{\Phi}{1} = \frac{1}{\Phi - 1}$
$\Leftrightarrow \Phi^2 - \Phi - 1 = 0$



Drawback in Recursive Algorithms



Generalization of Rabbits Problem

Statement:

Start with a pair of rabbits, one male and one female, born on January 1.

Assume that all months are of equal length and that rabbits begin to produce two months after their own birth. After reaching age of two months, each pair produces two other mixed pairs, two male and two female, and then two other mixed pair each month, and no rabbit dies.

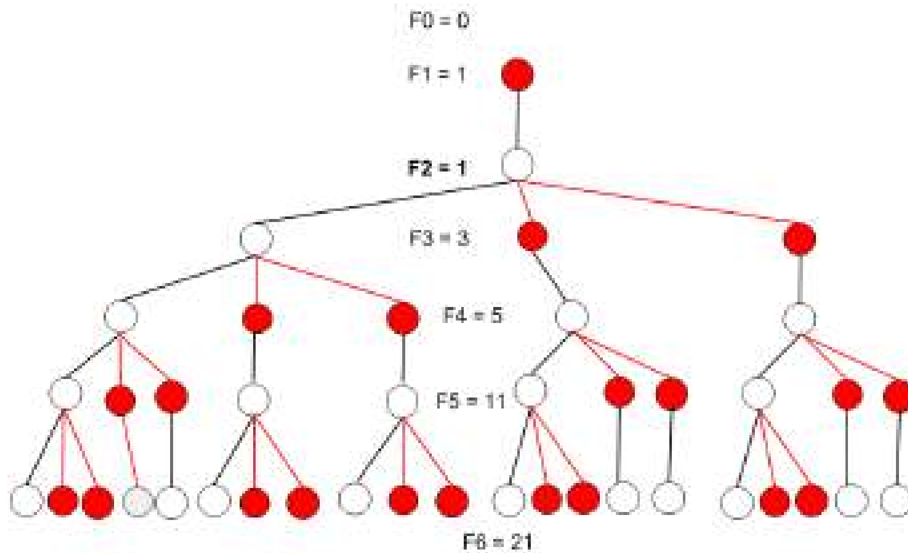
How many pairs of rabbits will there be after one year?

Answer: Generalization of Fibonacci Sequence!

0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341, 683, . . .

Construction of Mathematical Model

https://studyc.info/



Total pairs at level k =

Total pairs at level k-1 + Total pairs born at level k ………..(1)

Since

Total pairs born at level k = 2 x Total pairs at level k-2 …………. (2)

By (1) and (2), Total pairs at level k = Total pairs at level k-1 + 2 x Total pairs at level k-2

Now let us denote

F_k = Total pairs at level k

Our recursive mathematical model:

$F_k = F_{k-1} + 2.F_{k-2}$

General Model (m pairs production): $F_k = F_{k-1} + m.F_{k-2}$

<u>Generalization</u>

Recursive mathematical model for one pair production

$F_k = F_{k-1} + F_{k-2}$

Recursive mathematical model for two pairs production

$F_k = F_{k-1} + 2.F_{k-2}$

Recursive mathematical model for m pairs production

$F_k = F_{k-1} + m.F_{k-2}$

Computing Values using Mathematical Model

Since $F_k = F_{k-1} + 2.F_{k-2}$ $F_0 = 0, F_1 = 1$

$F_2 = F1 + 2.F_0 = 1 + 0 = 1$

$F_3 = F2 + 2.F_1 = 1 + 2 = 3$

$F_4 = F3 + 2.F_2 = 3 + 2 = 5$

$F_5 = F4 + 2.F_3 = 5 + 6 = 11$

$F_6 = F5 + 2.F_4 = 11 + 10 = 21$

$F_7 = F6 + 2.F_5 = 21 + 22 = 43$

$F_8 = F7 + 2.F_6 = 43 + 42 = 85$

https://studyc.info/

$F_9 = F8 + 2.F_7 = 85 + 86 = 171$
$F_{10} = F_9 + 2.F_8 = 171 + 170 = 341$
$F_{11} = F_{10} + 2.F_9 = 341 + 342 = 683$
$F_{12} = F_{11} + 2.F_{10} = 683 + 682 = 1365$ . . .

Another Generalization of Rabbits Problem

Statement:

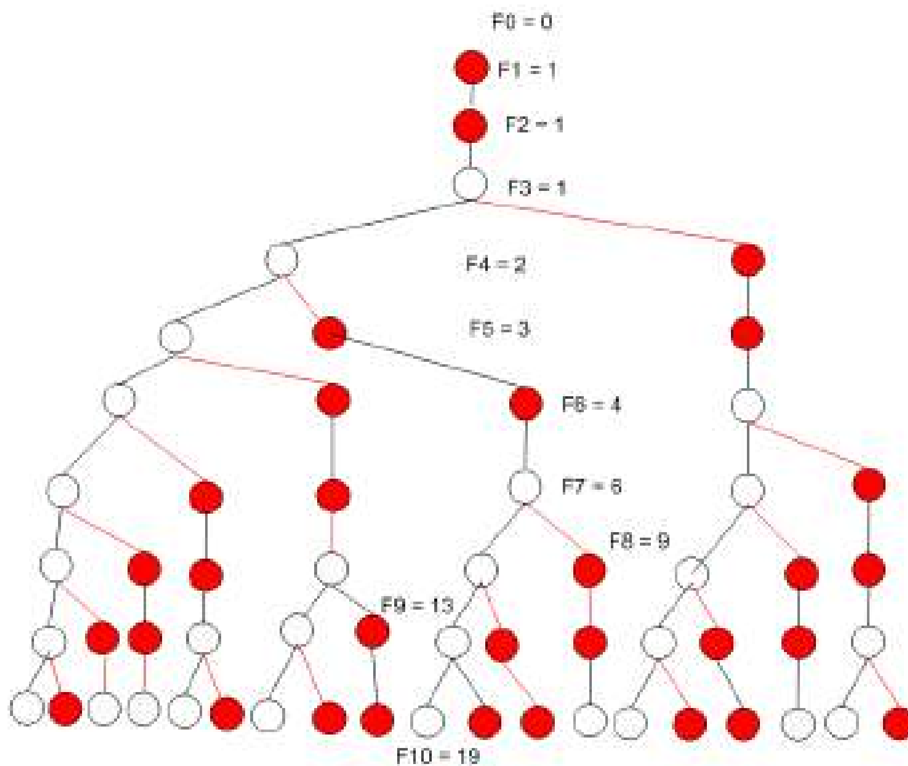
- Start with a different kind of pair of rabbits, one male and one female, born on January 1.
- Assume all months are of equal length and that rabbits begin to produce three months after their own birth.
- After reaching age of three months, each pair produces another mixed pairs, one male and other female, and then another mixed pair each month, and no rabbit dies.

How many pairs of rabbits will there be after one year?

Answer: Generalization of Fibonacci Sequence!

0, 1, 1, 1, 2, 3, 4, 6, 9, 13, 19, 28, 41, 60, . . .

Construction of Mathematical Model



Total pairs at level k = Total pairs at level k-1 + Total pairs born at level k ……….(1)

Since

Total pairs born at level k = Total pairs at level k-3 ……….(2)
By (1) and (2)

Total pairs at level k = Total pairs at level k-1 + Total pairs at level k-3
Now let us denote

F_k= Total pairs at level k
This time mathematical model: $F_k = F_{k-1} + F_{k-3}$

Computing Values using Mathematical Model
Since $F_k = F_{k-1} + F_{k-3}$ F0 = 0, F1= F2= 1
$F_3 = F_2 + F_0 = 1 + 0 = 1$
$F_4 = F_3 + F_1 = 1 + 1 = 2$
$F_5 = F_4 + F_2 = 2 + 1 = 3$
$F_6 = F_5 + F_3 = 3 + 1 = 4$
$F_7 = F_6 + F_4 = 4 + 2 = 6$
$F_8 = F_7 + F_5 = 6 + 3 = 9$
$F_9 = F_8 + F_6 = 9 + 4 = 13$
$F_{10} = F_9 + F_7 = 13 + 6 = 19$
$F_{11} = F_{10} + F_8 = 19 + 9 = 28$
$F_{12} = F_{11} + F_9 = 28 + 13 = 41$ . . .
More Generalization
Recursive mathematical model for one pair, production after three months
$F_k = F_{k-1} + F_{k-3}$
Recursive mathematical model for two pairs, production after three months
$F_k = F_{k-1} + 2.F_{k-3}$
Recursive mathematical model for m pairs, production after three months
$F_k = F_{k-1} + m.F_{k-3}$
Recursive mathematical model for m pairs, production after n months
$F_k = F_{k-1} + m.F_{k-n}$

<u>Applications of Fibonacci Sequences</u>

Fibonacci sequences are used

- in trend analysis
- By some pseudorandom number generators
- The number of petals is a Fibonacci number.
- Many plants show the Fibonacci numbers in the arrangements of the leaves around the stems.
- Seen in arrangement of seeds on flower heads
- Consecutive Fibonacci numbers give worst case behavior when used as inputs in Euclid's algorithm.
- As n approaches infinity, the ratio F(n+1)/F(n) approaches the golden ratio:
  Φ =1.6180339887498948482...

- The Greeks felt that rectangles whose sides are in the golden ratio are most pleasing

https://studyc.info/

- The Fibonacci number F(n+1) gives the number of ways for 2 x 1 dominoes to cover a 2 x n checkerboard.
- Sum of the first n Fibonacci numbers is F(n+2)-1.
- The shallow diagonals of Pascal's triangle sum to Fibonacci numbers.
- Except n = 4, if F(n) is prime, then n is prime.
- Equivalently, if n not prime, then F(n) is not prime.
- gcd(F(n), F(m)) = F(gcd(n, m))

https://studyc.info/

Lecture 07 Recurrence Relations

What is Recursion?

- Some times problems are too difficult or too complex to solve because these are too big.
- A problem can be broken down into sub-problems and find a way to solve these sub-problems
- Then build up to a solution to the entire problem.
- This is the idea behind recursion
- Recursive algorithms break down problem in pieces which you either already know the answer, or can be solved applying same algorithm to each piece
- And finally combine the results of these sub-problems to find the final solution
- More concisely, a recursive function or definition is defined in terms of itself.
- Recursion is a computer algorithm that calls itself in a steps having a termination condition.
- The successive repetitions are processed up to the critical step where the condition is met.
- In recursive algorithms, each repetition is processed from the last one called to the first.
- Recursion is a wonderful technique for dealing with many problems where problems and sub-problems have same mechanism to solve it.

Merits and Demerits of Recursion

- Recursive solutions are much easier to conceive of and code than their iterative counterparts.
- Every problem is not solvable using this approach
- What kinds of problems are solved with recursion?
- Generally, problems which are defined in terms of themselves are usually good candidates for recursive techniques.

Example

Finding factorial of a number is an easiest example one can think using recursion

Recursive Mathematical Model

Since n! can be computed as: 5! = 5*4*3*2*1.

If we have a close look at this, we notice that

$$5! = 5*4!.$$

Now if denote F(n) = n! then it can be written as

$$F(n) = n.F(n-1)$$

Assume 0! = 1 i.e. F(0) = 1, and solve till termination

$$F(n) = n.F(n-1) = n.(n-1).F(n-2) = n.(n-1).(n-2).F(n-3)$$

$$...$$

$$F(n) = n.(n-1).(n-2)...2.1.F(0) = n.(n-1).(n-2)...2.1.1$$

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \\ n.F(n-1) & \text{otherwise} \end{cases}$$

Solving Recurrence

https://studyc.info/

- The indispensable last step when analyzing an algorithm is often to solve a recurrence equation.
- With little experience recurrence equation can be solved by intelligent guesswork.
- However, there exist a powerful techniques that can be use to solve certain classes of recurrence almost automatically.

First Order Linear Homogenous Recurrence Relations with Constant Coefficients (FOLHRRCC)

Definition and Examples of FOLHRRCC

Definition

$a_k = A \cdot a_{k-1} \quad \forall\ k \geq 1$

where A is a real numbers and $A \neq 0$

$a_k = 2.a_{k-2}$ NO	$b_k = 5b_{k-1}$ YES
$a_k = a_{k-1}\ a_{k-1}$ NO	$c_k = k\frac{1}{2}c_{k-1}$ NO

Solving First Order Recurrences

Solve the recurrence $b_k = a.b_{k-1}$ if $b_0 = c$

Solution

$b_k = a.b_{k-1}$

$b_k = a.(a.b_{k-2})$

$b_k = a^2.(a.b_{k-3})$

$\dots$

$b_k = a^k.b_{k-k} = a^k.b_0$

$b_k = a^k.c$

Now the explicit formula is $c.a^k$ which is a geometric sequence.

Hence first order recurrence is nothing but G.S.

Example

Solve the recurrence $b_k = 5b_{k-1}$ if $b_0 = 3$

Solution

$b_k = 5.b_{k-1}$

$b_k = 5.(5.\ b_{k-2}) = 5^2.\ b_{k-2}$

$b_k = 52.(5.\ b_{k-3}) = 5^3.\ b_{k-3}$

$\dots$

$b_k = 5k.\ b_{k-k} = 5^k.\ b_0 = 5^k.3$

Now the solution sequence becomes

$3.5^0, 3.5^1, 3.5^2, 3.5^3, 3.5^4, \dots, 3.5^k, \dots$

3, 15, 75, 375, . . .geometric sequence

Second Order Linear Homogenous Recurrence Relations with Constant Coefficients (SOLHRRCC)

https://studyc.info/

Definition:

$a_k = A \cdot a_{k-1} + B \cdot a_{k-2} \quad \forall\ k \geq 2$

where A and B are real numbers with $B \neq 0$

$a_k = 3a_{k-1} + 2a_{k-2}$ YES	$b_k = b_{k-1} + b_{k-2} + b_{k-3}$ NO	$c_k = \frac{1}{2}c_{k-1} - \frac{3}{7}c_{k-2}$ YES
$d_k = d_{k-1}^2 + d_{k-1} \cdot d_{k-2}$ NO	$e_k = 2e_{k-2}$ YES	$f_k = 2f_{k-1} + 1$ NO
$g_k = g_{k-1} + g_{k-2}$ YES	$h_k = (-1)h_{k-1} + (k-1)h_{k-2}$ NO	

Theorem

Let A and B are real numbers. A recurrence of the form $a_k = Aa_{k-1} + Ba_{k-2}$ is satisfied by the sequence $1, t, t^2, \ldots. t^n, \ldots. t \neq 0$ where t is a non-zero real no, if and only if t satisfies the equation $t^2 - At - B = 0$

Proof:

Let us suppose that the sequence $1, t, t^2, \ldots, t^n, \ldots, where, t \neq 0$ satisfies the recurrence relation $a_k = Aa_{k-1} + Ba_{k-2}$. It means each form of sequence is equal to A times the previous form plus B times the form before it.

Hence $t^k = At^{k-1} + Bt^{k-2}$

since $t \neq 0 \Rightarrow t^{k-2} \neq 0$

Dividing both sides by t^{k-2}, we get

$t^2 - At - B = 0$

Conversely suppose that $t^2 - At - B = 0$

$\Rightarrow t^k = At^{k-1} + Bt^{k-2}$

Hence $1, t, t^2, t^3 \ldots, t^n \ldots$ satisfies the above recurrence relation.

Characteristic Equation

Given a second order linear homogeneous recurrence relation with constant coefficients

$a_k = A \cdot a_{k-1} + B \cdot a_{k-2} \quad \forall\ k \geq 2$

The characteristic equation of the relation is $t^2 - At - B = 0$

Example: Using characteristic equation, find solution to a recurrence relation

$a_k = a_{k-1} + 2 \cdot a_{k-2} \quad \forall\ k \geq 2$

Find all sequences that satisfy relation and have the form $1, t, t^2, \ldots, t^n, \ldots, t \neq 0$

Solution: Above recurrence relation is satisfied by a sequence $1, t, t^2, \ldots, t^n, \ldots, t \neq 0$ if and only if, $t^2 - t - 2 = 0$

$\Rightarrow (t-2)(t+1) = 0 \quad \Rightarrow t = 2, -1$

Hence

$2^0, 2, 2^2, 2^3, \ldots, 2^n, \ldots$ and $(-1)^0, (-1)^1, (-1)^2, (-1)^3, \ldots$

https://studyc.info/

are both particular solutions for this recurrence relation

Repeated Roots:

Theorem: Let A and B be real numbers, and suppose that the characteristic equation $t^2 - At - B = 0$ has a single root r, then the sequences

$1, r, r^2, ..., r^n$ and $0, r, 2r^2, 3r^3, ..., nr^n ...$ both satisfy the recurrence relation

$a_k = Aa_{k-1} + Ba_{k-2}$

Proof:

if $t^2 - At - B = 0$ has a single repeated root r, then

$$t^2 - At - B = (t - r)^2$$

$$\Rightarrow t^2 - At - B = t^2 - 2rt + r^2$$

$$\Rightarrow A = 2r \quad \text{and} \quad B = -r^2$$

We know that r^n is a solution. Now we prove that $S_n = nr^n$ is also a solution, i.e.

S_n satisfies the recurrence relation $a_k = Aa_{k-1} + Ba_{k-2}$

i.e., $S_k = AS_{k-1} + BS_{k-2} = kr^k$

In fact we have to prove that

$A.S_{k-1} + B.S_{k-2} = k.r^k$

Consider the left hand side

$$AS_{k-1} + BS_{k-2} = A\left((k-1) \cdot r^{k-1} \right) + B\left((k-2) \cdot r^{k-2} \right)$$
$$= 2r \cdot (k-1) \cdot r^{k-1} - r^2 \cdot (k-2) \cdot r^{k-2}$$
$$= 2(k-1) \cdot r^k - (k-2)r^k$$
$$= (2k - 2 - k + 2)r^k$$
$$= k.r^k = \text{RHS}, \quad \text{hence proved}$$

Example 1: Single Root case

Suppose sequence, $b_0, b_1, b_2, ...$ satisfies recurrence relation

$b_k = 4b_{k-1} - 4b_{k-2} \quad \forall\ k \geq 2$

with initial condition: $b_0 = 1$ and $b_1 = 3$

then find the explicit formula for $b_0, b_1, b_2, ...$

Solution: Characteristic equation is $t^2 - 4t + 4 = 0$

$(t-2)^2 = 0 \Rightarrow t = 2,$ is repeated root

2^n and $n.2^n$ are sequences which satisfy the same recurrence relation, but do not satisfy the initial conditions

Suppose general solution: $b_n = C.2^n + D.n.2^n$

which satisfies the original recurrence , C and D are constants

Since $b_0 = 1, \quad b_1 = 3$

For $n = 0, \quad C + D \cdot 0 \cdot 2^0 = 1 \Rightarrow \quad C = 1$

https://studyc.info/

For $n=1$, $\quad b_1 = C\cdot 2^1 + D\cdot 1\cdot 2^1$

$\Rightarrow\ 2C+2D=3 \Rightarrow 2\cdot 1+2D=3 \Rightarrow\ D=\frac{3-2}{2}=\frac{1}{2}$

Hence, $b_n = 1\cdot 2^n + \frac{1}{2}\cdot n\cdot 2^n = 2^n\left(1+\frac{n}{2}\right)$

$=\left(1+\frac{n}{2}\right)2^n$ is the required solution for repeated roots.

Checking explicit formula:

$b_n = (1 + n/2).2^n$

First term	$b_0 = (1 + 0/2).2^0 = 1.1 = 1$
Second term	$b_1 = (1 + 1/2).2^1 = 3/2.2 = 3$
Third term	$b_2 = (1 + 2/2).2^2 = 2.4 = 8$
Fourth term	$b_3 = (1 + 3/2).2^3 = 5/2.8 = 20$, and so on

Theorem (Linear Combination is also a Solution)

If $r_0, r_1, r_2, ...$ and $s_0, s_1, s_2, ...$ are sequences that satisfy the some second order linear homogeneous recurrence relation with constant coefficients, and if C and D are any numbers then the sequence $a_0, a_1, a_2, ...$ defined by the formula $a_n = Cr_n + Ds_n \quad \forall\ n \ge 0$ also satisfies the same recurrence relation.

Proof:

Since $r_0, r_1, r_2, ...$ and $s_0, s_1, s_2, ...$ satisfy the same second order LHRRWCC

$\Rightarrow\ \exists\, A$ and B constants real numbers such that

$r_k = Ar_{k-1} + Br_{k-2}$ and $s_k = As_{k-1} + Bs_{k-2}$

If $a_n = Cr_n + Ds_n \quad \forall\ n \ge 0$

Then we have to prove that $a_k = Aa_{k-1} + Ba_{k-2}$

Consider R.H.S.

$$\begin{aligned} Aa_{k-1} + Ba_{k-2} &= A\left(C\cdot r_{k-1} + D\cdot s_{k-1}\right) + B\left(Cr_{k-2} + Ds_{k-2}\right) \\ &= C\left(Ar_{k-1} + Br_{k-2}\right) + D\left(As_{k-1} + Bs_{k-2}\right) \\ &= C\left(r_k\right) + D(s_k) = a_k \\ &\Rightarrow\ a_k = Aa_{k-1} + Ba_{k-2} \end{aligned}$$

It proves that $a_n = Cr_n + Ds_n$ satisfies same recurrence.

Example 1: Find a sequence that satisfies the recurrence relation

$a_k = a_{k-1} + 2\cdot a_{k-2} \quad \forall\ k \ge 2$ and that also satisfies the initial condition $a_0 = 1$ and $a_1 = 8$

Solution: The characteristic equation of the relation $a_k = a_{k-1} + 2\cdot a_{k-2} \quad \forall\ k \ge 2$ is

$t^2 - t - 2 = 0 \quad \Rightarrow \quad (t-2)(t+1) = 0$

$t = -1, 2$

https://studyc.info/

$r_n = 2^0, 2^1, 2^2, \ldots$ and $s_n = (-1)^0, (-1)^1, (-1)^2, \ldots$

Are both sequences which satisfy above relation but neither one is having $a_0 = 1$ and $a_1 = 8$

Now since $a_n = Cr_n + Ds_n$ also satisfies the same recurrence relation and

For n = 0 we get $a_o = Cr_o + Ds_o \Rightarrow 1 = C + D$ (1)

For n = 1 we get $a_1 = Cr_1 + Ds_1 \Rightarrow 8 = 2C - D$ (2)

Solving equation (1) and (2) we get, $C = 3$ *and* D = -2

Now $a_n = Cr_n + Ds_n \Rightarrow a_n = 3 \cdot (2)^n - 2(-1)^n$

is the required sequence which satisfies the given conditions

K order: General Homogenous Recurrence

We extend technique of recurrence equation with resolution of homogeneous linear recurrence with constant coefficient that is recurrence of the form

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0 \quad (1)$$

where t_i are value we are looking for

In equation (1), values of t_i such that (1 < i < k) are need to determine a sequence.

Equation 1 typically has infinite many solutions, because linear combination is also a solution.

k = 1: General Homogenous Recurrence

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0 \quad (1)$$

If we put k = 1, then above equation becomes

$$a_0 t_n + a_1 t_{n-1} = 0$$
$$t_n = -(a_1 / a_0) t_{n-1}$$

The resultant equation becomes a recurrence relation which is first order, linear, homogenous, with constant coefficients.

k = 2: General Homogenous Recurrence

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0 \quad (1)$$

If we put k = 2, then above equation becomes

$$a_0 t_n + a_1 t_{n-1} + a_2 t_{n-2} = 0$$
$$t_n = -(a_1 / a_0) t_{n-1} - (a_2 / a_0) t_{n-2} = c_1 t_{n-1} + c_0 t_{n-2}$$

This time we have a recurrence relation which is second order, linear, homogenous, with constant coefficients.

k = 3: General Homogenous Recurrence

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0 \quad (1)$$

If we put k = 3, then above equation becomes

https://studyc.info/

$a_0 t_n + a_1 t_{n-1} + a_2 t_{n-2} + a_3 t_{n-3} = 0$

$t_n = -(a_1 / a_0) t_{n-1} - (a_2 / a_0) t_{n-2} - (a_3 / a_0) t_{n-3} = c_1 t_{n-1} + c_0 t_{n-2}$

This is a recurrence relation which is third order, linear, homogenous, with constant coefficients.

Similarly we can have fourth order and higher order recurrence relations.

Characteristics of General Recurrence

$a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = 0$ (1)

The recurrence is:

- Linear because it does not contain terms of the form $t_{n-i} . t_{n-j}$,t^2_{n-i} and so on.
- Homogeneous because the linear combination of the t_{n-i} equal to zero
- This is k^{th} order because current term is linear combination of previous k number of terms
- Constant coefficients because all a_i are constants

Example:

Consider Fibonacci sequence: $f_n = f_{n-1} + f_{n-2}$

This recurrence fits Eq. (1) after obvious rewriting.

$f_n - f_{n-1} - f_{n-2} = 0$

Observation of Homogenous Recurrence

$f_n - f_{n-1} - f_{n-2} = 0$

The Fibonacci sequence corresponds to a second homogeneous linear recurrence relation with constant coefficient, where

$k = 2, \ a_0 = 1, \ a_1 = -1 \ , a_2 = -1$

Before we even start to look for solutions to Equation 1, it would be interesting to note that any linear combination of solutions is itself a solution.

Theorem:

Statement: Prove that any linear combination of solutions of equation given below is also a solution.

$a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = 0$

Proof: Assume that f_n and g_n are solutions of above equation and hence satisfy it, i.e.

$\sum_{i=0}^{k} a_i f_{n-i} = 0$ and $\sum_{i=0}^{k} a_i g_{n-i} = 0$

If we set $t_n = c.f_n + d.g_n$ for arbitrary constants c and d, we have to prove that t_n is also solution, i.e., $a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = 0$

https://studyc.info/

$a_0 t_n + a_1 t_{n\text{-}1} + ... + a_k \ t_{n\text{-}k} \ =$

$a_0 \ (cf_n + dg_n) + a1(cf_{n\text{-}1} + dg_{n\text{-}1}) + . . . + a_k (cf_{n\text{-}k} + dg_{n\text{-}k}) \ =$

$c(\ a_0 f_n \ + \ a_1 f_{n\text{-}1} \ + \ . . . + \ a_k f_{n\text{-}k} \ ) \ + d(a_0 g_n + a_1 g_{n\text{-}1} + \ . . . + a_k g_{n\text{-}k}) \ =$

$= \ c.0 \ + \ d.0 \ = \ 0$

Hence $a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = 0$

Note: It is to be noted that this rule can be generalized to linear combinations of any number of solutions.

More Generalized Results:

If f_n, g_n and h_n are solutions to recurrence below then their linear combination is also a solution of $a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = 0$

Proof:

As f_n, g_n and h_n are solutions of above, hence

$\sum_{i=0}^{k} a_i f_{n-i} = 0, \ \ \sum_{i=0}^{k} a_i g_{n-i} = 0, \text{ and } \sum_{i=0}^{k} a_i h_{n-i} = 0$

Let $t_n = c.f_n + d.g_n + e.h_n$ for arbitrary constants c, d and e, now we prove that t_n is also solution, i.e. $a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = 0$ (easy to prove)

https://studyc.info/

Lecture 08 Recurrence Relations

Solution to Generalized Recurrence with Distinct Roots

Statement:

Find the general solution of the kth order recurrence given below assuming that all roots are distinct

$$a_0 t^n + a_1 t^{n-1} + \ldots + a_k t^{n-k} = 0$$

Solution: Let $T_n = x^n$, x is a constant as yet unknown.
If we assume that Tn is a solution of equation

$$a_0 t^n + a_1 t^{n-1} + \ldots + a_k t^{n-k} = 0$$

Then, $a_0 x^n + a_1 x^{n-1} + \ldots + a_k x^{n-k} = 0$

Equation satisfied if x = 0, trivial solution, no interest.
Otherwise, the equation is satisfied if and only if

$$a_0 + a_1 x^1 + \ldots + a_k x^k = 0$$

This equation of degree k in x is called the characteristic equation of above recurrence and

$$P(x) = a_0 + a_1 x^1 + \ldots + a_k x^k$$

is called its characteristic polynomial

Fundamental theorem of algebra states that polynomial P(x) of degree k has k roots, not necessarily distinct
It means that it can be factorized as a product of k terms

$$P(x) = (x-r_1)(x-r_2)(x-r_3)\ldots(x-r_k)$$

where ri may be complex numbers.

Moreover, these r_i are only solutions of equation P(x) = 0
Consider any root r_i of the characteristic polynomial

$$P(x) = a_0 + a_1 x^1 + \ldots + a_k x^k = \prod_{i=1}^{k}(x - r_i)$$

Since, $p(r_1) = 0, p(r_2) = 0, p(r_3) = 0, \ldots, p(r_k) = 0$
Hence all $x = r_i$, for $i \in \{1, 2, \ldots, k\}$ are solutions to above characteristic polynomial.
Therefore, $r_1^n, r_2^n, \ldots, r_k^n$ are solution to our original recurrence relation.
Since linear combination of solutions is also a solution to recurrence, therefore below is a solution.

$$T(n) = \sum_{i=1}^{k} c_i r_i^n$$

where, $c_1, c_2, \ldots, c_k$ are all constants to be determined finding particular solution
The remarkable fact is that this equation has only solutions of this form provided all r_i are distinct.
Constants can be determined from k initial conditions by solving system of k linear equations in k unknowns

Problem:

Consider the recurrence

$t_n = n$ if $n = 0, 1, 2$

https://studyc.info/

$t_n = 7.t_{n-2} + 6.t_{n-3}$ otherwise

Find the general solution of the recurrence above.

Solution: First we rewrite the recurrence.

$t_n = 7.t_{n-2} + 6.t_{n-3}$

The characteristic equation becomes $x^3 - 7x - 6 = (x + 1)(x + 2)(x - 3)$

The roots are: $r_1 = -1$, $r_2 = -2$ and $r_3 = +3$

$t_n = c_1 (-1)^n + c_2 (-2)^n + c_3 (3)^n$

The initial conditions give

$c_1 + c_2 + c_3 = 0$ for $n = 0$

$-c_1 - 2c_2 + 3c_3 = 1$ for $n = 1$

$c_1 + 4c_2 + 9c_3 = 2$ for $n = 2$

Solving these equations, we obtain

$c_1 = -1/4$, $c_2 = 0$ and $c_3 = 1/4$

Therefore, $t_n = (-1/4)(-1)^n + (1/4).(3)^n$

Solution to Generalized Recurrence with One Repeated Root

Statement:

If the characteristic polynomial
$P(x) = a_0 + a_1x^1 + \ldots\ldots + a_kx^k$

then conclude that if r is a double root then $t_n = r^n$ and $t_n = n\,r^n$ are both solutions to recurrence.

Solution:
It can be found as in case of second order linear homogenous recurrence relation.
Since r is a multiple root. By definition of multiple roots, there exists a polynomial q(x) of degree k-2 such that the following holds

$p(x) = (x - r)^2 q(x)$, for every $n \geq k$

Consider the kth degree polynomials

$u_n(x) = a_0 x^n + a_1 x^{n-1} + \ldots + a_k x^{n-k}$ and

$v_n(x) = a_0\, n\, x^n + a_1 (n-1)x^{n-1} + \ldots + a_k (n-k)x^{n-k}$

It is to be noted that $v_n(x) = x.u_n'(x)$, where $u_n'(x)$ denotes the derivative of $u_n(x)$ with respect to x .

But $u_n(x)$ can be written as

$u_n(x) = x^{n-k} p(x) = x^{n-k} (x-r)^2 q(x) = (x-r)^2 (x^{n-k} q(x))$

Using rule for computing the derivative of a product of functions, we obtain that derivative of $u_n(x)$ with respect to x is given by

$u_n'(x) = 2(x-r)\, x^{n-k} q(x) + (x-r)^2 (x^{n-k} q(x))'$

therefore $u_n'(r) = 0$, which implies that $v_n(r) = r \,.\, u_n'(r) = 0$ for all $n \geq k$.

It means: $a_0\, n\, r^n + a_1 (n-1)\, r^{n-1} + \ldots + a_k (n-k)\, r^{n-k} = 0$

https://studyc.info/

Hence, $t_n = n\ r^n$ is also solution to the recurrence.
Now we conclude that if r is a double root then $t_n = r^n$ and $t_n = n\ r^n$ are both solutions to the recurrence.
Rest of k-2 are all distinct roots hence general solution

$$t_n = c_1 r^n + c_2 n r^n + b_1 r_1^n + b_2 r_2^n + \ldots + b_{k-2} r_{k-2}^n$$

where $c_1, c_2, b_1, b_2, \ldots,$ and b_{k-2} are all real constants

Higher Order Homogenous Recurrence with k-multiplicity of a Root

Now if we have to solve recurrence order k, then to solve polynomial, degree k, given below is sufficient

Statement:

If the characteristic polynomial

$P(x) = a_0 + a_1 x^1 + \ldots\ldots + a_k x^k$

- has r as double root then it can be written as
  $p(x) = (x - r)^2 q(x)$, for every $n \geq k$
  and solutions are: r^n and nr^n
- has r as triple root then it can be written as
  $p(x) = (x - r)^3 q_1(x)$, for every $n \geq k$
  and solutions are: r^n, nr^n and $n^2 r^n$
- r has multiplicity k then it can be written as
  $p(x) = (x - r)^k$, for every $n \geq k$
  and solutions are: $r^n, n.r^n, n^2.r^n, \ldots, n^{k-1}.r^n$

then general solution is

$$t_n = c_1 r^n + c_2 n r^n + \ldots + c_k n^{k-1} r^n$$

$$t_n = \sum_{j=1}^{k} c_j n^{j-1} r^n$$

where $b_1, b_2, \ldots, b_k$ are all real constants

Multiplicity of Roots: More General Result

If there are I roots, $r_1, r_2, \ldots, r_l$ with multiplicities $m_1, m_2, \ldots, m_l$ respectively, of the polynomial:

$P(x) = a_0 + a_1 x^1 + \ldots + a_k x^k$ *such that* $m_1 + m_2 + \ldots + m_l = k$

then the general solution to the recurrence is

$$\begin{aligned} t_n = {} & c_{11} r_1^n + c_{12} n r_1^n + c_{13} n^2 r_1^n + \ldots + c_{1m_1} n^{m_1 - 1} r_1^n + \\ & c_{21} r_2^n + c_{22} n r_2^n + c_{23} n^2 r_2^n + \ldots + c_{2m_2} n^{m_2 - 1} r_2^n + \\ & \ldots \\ & c_{l1} r_l^n + c_{l2} n r_l^n + c_{l3} n^2 r_l^n + \ldots + c_{lm_l} n^{m_l - 1} r_l^n \end{aligned}$$

$$t_n = \sum_{j=1}^{m_1} c_{1j} n^{j-1} r_1^n + \sum_{j=1}^{m_2} c_{2j} n^{j-1} r_2^n + \ldots + \sum_{j=1}^{m_l} c_{lj} n^{j-1} r_l^n$$

https://studyc.info/

$$t_n = \sum_{i=1}^{l} \sum_{j=1}^{m_i} c_{ij} n^{j-1} r_i^n$$ where all $c_{i,\,j}$ are constants

Problem:

Statement: Consider the recurrence

$t_n = n$ if $n = 0, 1, 2$

$t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3}$ otherwise

Find general solution of the recurrence above.

Solution: First we rewrite the recurrence.

$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$

The characteristic equation become.

$x^3 - 5x^2 + 8x - 4 = (x-1)(x-2)^2$

The roots are: $r_1 = 1$ of multiplicity $m_1 = 1$ and $r_2 = 2$ of multiplicity $m_2 = 2$, and hence the general solution is $t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$

The initial conditions give

$c_1 + c_2 = 0$ for $n = 0$

$c_1 + 2c_2 + 2c_3 = 1$ for $n = 1$

$c_1 + 4c_2 + 8c_3 = 2$ for $n = 2$

Solving these equations, we obtain

$c_1 = -2$, $c_2 = 2$ and $c_3 = -1/2$

Therefore,

$t_n = c1\ 1^n + c_2 2^n + c_3 n 2^n$

$= -2 + 2.2^n - ½.n.2^n = 2^{n+1} - n.2^{n-1} - 2$

Non-homogeneous Recurrence of Higher Order

Solution of a linear recurrence with constant coefficients becomes more difficult when the recurrence is not homogeneous

That is when linear combination is not equal to zero

In particular, it is no longer true that any linear combination of solutions is a solution.

Consider the following recurrence

$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n)$

The left-hand side is the same as before, but on the right-hand side we have $b^n p(n)$ where b is a constant and p(n) is a polynomial in n of degree d.

Generalization: Non-homogeneous Recurrences

If a recurrence is of the form $a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1^n p_1(n)$

Then the characteristics polynomial is $(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - b_1^{d+1})$

which contains one factor for the left hand side

And other factor corresponding to the right hand side, where d is degree of polynomial

Characteristics polynomial can be used to solve the above recurrence

Problem 1:

Consider the recurrence below. Find its solution

$t_n - 2t_{n-1} = 3^n$

https://studyc.info/

Solution:

Compare the above recurrence with

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n)$$

Here: b = 3, p(n) = 1, a polynomial of degree 0.

Reducing to homogeneous case, we are given

$$t_n - 2t_{n-1} = 3^n \qquad (1)$$

Replace n by n - 1 and multiply by 3, we obtain

$$t_{n-1} - 2t_{n-2} = 3^{n-1}$$

$$3t_{n-1} - 6t_{n-2} = 3^n \qquad (2)$$

From (1) and (2)

$$t_n - 2t_{n-1} = 3^n \qquad (1)$$

$$+ 3t_{n-1} - 6t_{n-2} = 3^n \qquad (2)$$

Subtracting 2 from equation 1, we get

$$t_n - 5t_{n-1} + 6t_{n-2} = 0$$

The characteristic equation is $x^2 - 5x + 6 = 0$

Roots of this equation are: $x = 2$ and $x = 3$

And therefore general solution is $t_n = c_1 2^n + c_2 3^n$

It is not always true that an arbitrary choice of c_1 and c_2 produces a solution to the recurrence even when initial conditions are not taken into account.

Note:

It is to be noted that solutions

$t_n = 2^n$ and

$t_n = 3^n$

which are solutions to reduced recurrence, are not solution to original one.

What is the reason?

Problem 2:

Find general solution of the following recurrence.

$$t_n - 2t_{n-1} = (n + 5) 3^n \; ; \qquad n \geq 1$$

Solution:

The manipulation needed to transform this into a homogeneous recurrence is slightly more complicated than with first example.

$$t_n - 2t_{n-1} = (n + 5) 3^n \; ; \qquad n \geq 1 \qquad (1)$$

replace n by n-1, n-2, we get

$$t_{n-1} - 2t_{n-2} = (n + 4) 3^{n-1} \; ; \qquad n \geq 2 \qquad (2)$$

$$t_{n-2} - 2t_{n-3} = (n + 3) 3^{n-2} \; ; \qquad n \geq 3 \qquad (3)$$

Above equations can be written as

$$t_n - 2t_{n-1} = 9(n + 5) 3^{n-2} \; ; \qquad n \geq 1 \qquad (4)$$

$$t_{n-1} - 2t_{n-2} = 3(n + 4) 3^{n-2} \; ; \qquad n \geq 2 \qquad (5)$$

$$t_{n-2} - 2t_{n-3} = (n + 3) 3^{n-2} \; ; \qquad n \geq 3 \qquad (6)$$

Our objective is to eliminate the right hand side of the above equations to make it homogenous.

Multiply (5) by -6, and (6) by 9 we get

$$t_n - 2t_{n-1} = 9(n + 5) 3^{n-2}$$

https://studyc.info/

$- 6t_{n-1} + 12t_{n-2} \quad = -18(n + 4)\, 3^{n-2}$

$+ 9t_{n-2} - 18t_{n-3} = 9(n + 3)\, 3^{n-2}$

After simplification, the above equations can be written as

$t_n - 2t_{n-1} = (9n + 45)\, 3^{n-2}$

$- 6t_{n-1} + 12t_{n-2} = (-18n – 72)\, 3^{n-2}$

$+ 9t_{n-2} - 18t_{n-3} = (9n + 27)\, 3^{n-2}$

Adding these equation, we get homogenous equation, which can be solved easily

$t_n - 8t_{n-1} + 21t_{n-2} - 18t_{n-3} = 0$

The characteristics equation of the above homogenous equation is:

$x^3 – 8x^2 +21x -18 = 0$

$(x-2) (x-3)^2 = 0$

and hence, x = 2, 3, 3

General solution is: $t_n = c_1 2^n + c_2 3^n + c_3\, n\, 3^n$

For n = 0, 1, 2

We can find values of c_1, c_2, c_3 and then

$t_n = (t_0 - 9)\, 2^n + (n + 3)3^{n+1}$

Problem 3: Tower of Hanoi

- Tower of Hanoi is a mathematical game or puzzle. It consists of three towers, a number of disks of different sizes which can slide onto any tower. The puzzle starts with disks stacked in order of size on one tower, smallest at top, making a conical shape.
- Objective is to move entire stack to another tower, obeying following rules:
- Only one disk may be moved at a time.
- Each move consists of taking upper disk from one of towers and sliding it onto another tower
- You can put on top of other disks already present
- No disk may be placed on top of a smaller disk.
- If a recurrence is of the form $a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \ldots$
- Then the characteristics polynomial is $(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - b_1^{d_1+1})(x - b_2^{d_2+1})\ldots,$
- Which contains one factor for the left hand side
- And other factor corresponding to the each term on right hand side.
- Once the characteristics polynomial is obtained the recurrence can be solved as before.

Problem 4 : Non-homogeneous Recurrences

Consider the recurrence

$t_n = 2t_{n-1} + n + 2^n$ otherwise

Solution: Compare the recurrence: $t_n - 2t_{n-1} = n + 2^n$ with

$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \ldots$

Here, $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$, and $p_2(n) = 1$.

Degree of $p_1(n) = d_1 = 1$,

Degree of $p_2(n) = d_2 = 0$.

The characteristic polynomial: $(x-2)\,(x-1)^2\,(x-2)$

https://studyc.info/

The roots are, x = 1, 2, both of multiplicity 2.
All solutions of recurrence therefore have form

$$t_n = c_1 1^n + c_2 n1^n + c_3 2^n + c_4 n 2^n$$

$$n + 2^n = (2c_2 - c_1) - c_2 n + c_4 2^n$$

For n = 0, 1, 2, 3 c_1, c_2, c_3 and c_4 can be solved and hence solution is

$$t_n = n.2^n + 2^{n+1} - n - 2$$

Conclusion:

- Recursive relations are important because used in divide and conquer, dynamic programming and in many other e.g. optimization problems
- Analysis of such algorithms is required to compute complexity
- Recursive algorithms may not follow homogenous behavior, it means there must be some cost in addition to recursive cost
- Solution to homogenous recursive relations is comparatively easy, but not so easy in case of non-homogenous recursive relations

https://studyc.info/

Lecture 09 Further Techniques Solving Recurrence Relations

Assumption in Solving Recurrence Relation

- Neglect certain technical details solve recurrences
- Assume integer arguments to functions because running time T (n) is always defined when n is an integer
- Consequently, for convenience, we shall omit statements of boundary conditions of recurrences and assume that T (n) is constant for small n
- Recurrence for worst-case time of MERGE-SORT

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T\left\lceil \frac{n}{2} \right\rceil + T\left\lfloor \frac{n}{2} \right\rfloor + \Theta(n) & \text{otherwise} \end{cases}$$

Assumption in Solving Recurrence Relation

- We do not give any explicit value for small n
- Because changing n, T (1) changes solution to recurrence. Solution typically doesn't change by more than a constant factor, so order of growth is unchanged
- Solve recurrences; often omit floors, ceilings, etc.
- First analyze without these details and later determine whether such assumptions matter or not.
- Usually it does not matter but it is important to know when it does and when it does not

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2.T(\frac{n}{2}) + \Theta(n) & \text{otherwise} \end{cases}$$

Methods Solving Recurrence Relation

The Substitution Method

- Substitution method has two steps
  - Guess the form of the solution
  - Use mathematical induction to find constants and show that the solution does work
- The name Substitution comes from the substitution of guessed answer for the function when the inductive hypothesis is applied to smaller values.
- Method is powerful, but it can be applied only in cases when it is easy to guess the form of answer
- The substitution method can be used to establish either upper or lower bounds on a recurrence.

Some Important Rules used in this Section

Prove that $\log_a b \cdot \log_b c = \log_a c$

https://studyc.info/

Proof: Let us suppose that $\log_a^b = s$ and $\log_b^c = t$

$\Rightarrow \quad a^s = b \quad$ and $\quad b^t = c$

$Now \quad b^t = c, \qquad (a^s)^t = c$

$a^{st} = c, \qquad \log_a^c = s \cdot t$

$s \cdot t = \log_a^c, \qquad \log_a^b \cdot \log_b^c = \log_a^c \quad$ proved

Prove that $3^{\log_4^n} = n^{\log_4^3}$

Proof: $3^{\log_4^n} = n^{\log_4^3}$

$\Leftrightarrow \quad \log_3(3^{\log_4^n}) = \log_3(n^{\log_4^3})$

$\Leftrightarrow \quad \log_4^n \cdot \log_3^3 = \log_4^3 \cdot \log_3^n$

$\Leftrightarrow \quad \log_4^n = \log_4^3 \cdot \log_3^n$

$\Leftrightarrow \quad \log_4^n = \log_4^n$

The Substitution Method

Solve the recurrence relation given below.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3 \cdot T(\frac{n}{4}) + n & \text{otherwise} \end{cases}$$

Solution: (1) $T(n) = 3 \cdot T(\frac{n}{4}) + n$

replace the value of n by $n/4$ in (1)

(2) $T(\frac{n}{4}) = 3 \cdot T(\frac{n}{4^2}) + \frac{n}{4}$

(3) $T(\frac{n}{4^2}) = 3 \cdot T(\frac{n}{4^3}) + \frac{n}{4^2}$ and so on

(4) $T(\frac{n}{4^{k-1}}) = 3 \cdot T(\frac{n}{4^k}) + \frac{n}{4^{k-1}}$

Now substitute the value of $T(\frac{n}{4})$ from (2) to (1)

(5) $T(n) = 3\left[3T(\frac{n}{4^2}) + \frac{n}{4}\right] + n$

$$= 3^2 \cdot T(\frac{n}{4^2}) + (\frac{3}{4}) \cdot n + n$$

substitute the value of $T(\frac{n}{4^2})$ from (3) to (5) equation, we have

$$T(n) = 3^2\left[3 \cdot T(\frac{n}{4^3}) + \frac{n}{4^2}\right] + (\frac{3}{4})n + n$$

After continuing this process

$$T(n) = 3^k \cdot T(\frac{n}{4^k}) + (\frac{3}{4})^{k-1} n + (\frac{3}{4})^{k-2} + \cdots + (\frac{3}{4})n + (\frac{3}{4})^0 n$$

https://studyc.info/

Let us suppose that n can be expressed as $n = 4^k$

$$T(n) = 3^k \cdot T(\frac{n}{n}) + n\left[1 + (\frac{3}{4}) + (\frac{3}{4})^2 + ... + (\frac{3}{4})^{k-1}\right]$$

$$T(n) = 3^k \cdot T(1) + n\left[1 \cdot (\frac{1-(\frac{3}{4})^k}{1-\frac{3}{4}})\right] \qquad \because 1 + x + x^2 + ... + x^{k-1} = 1 \cdot (\frac{1-x^k}{1-x})$$

$$T(n) = 3^k \cdot 1 + 4n(1 - \frac{3^k}{4^k}) \qquad \because T(1) = 1$$

$$T(n) = 3^k + 4n \cdot (1 - \frac{3^k}{4^k}) = 3^k + 4n \cdot (1 - \frac{3^k}{n})$$

$$= 3^k + 4n(\frac{n-3^k}{n})$$

$$= 3^k + 4(n - 3^k) = 1 \cdot 3^k + 4n - 4 \cdot 3^k$$

$$= 4 \cdot n - 3 \cdot 3^k = 4n - 3 \cdot 3^{\log_4^n} \qquad \because 4^k = n, \quad k = \log_4^n$$

$$T(n) = 4n - 3 \cdot 3^{\log_4^n}$$

$$T(n) = 4n - 3 \cdot n^{\log_4^3}$$

$$T(n) = 4n - 3 \cdot n^{\alpha} \qquad \because 0 \le \alpha \le 1$$

$$\Rightarrow \; T(n) = 4n - 3 \cdot n^{\alpha}$$

Hence $T(n) \in \theta(n)$ *let* $\alpha = \log_4^3$

<u>A Hard Example</u>

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ aT(\frac{n}{b}) + n & \text{otherwise} \end{cases} \qquad \text{where} \quad 0 < a < b$$

<u>Solution:</u>

$$T(n) = aT(\frac{n}{b}) + n = a \cdot a \cdot T(\frac{n}{b^2}) + a \cdot \frac{n}{b} + n$$

$$= a^2 \cdot T(\frac{n}{b^2}) + (\frac{a}{b})n + n = a^2\left[aT(\frac{n}{b^3}) + \frac{n}{b^2}\right] + (\frac{a}{b})n + n$$

$$= a^3 T(\frac{n}{b^3}) + (\frac{a}{b})^2 n + (\frac{a}{b})n + n$$

Continuing this process, we get

$$T(n) = a^k T(\frac{n}{b^k}) + (\frac{a}{b})^{k-1} n + (\frac{a}{b})^{k-2} n + ... + (\frac{a}{b})^1 n + n$$

$$= a^k T(\frac{n}{b^k}) + n\left[1 + (\frac{a}{b}) + (\frac{a}{b})^2 + ... + (\frac{a}{b})^{k-1}\right]$$

Let us suppose that, $n = b^k$

https://studyc.info/

$$T(n) = a^k \cdot T(1) + n\left[1 + (\frac{a}{b}) + ... + (\frac{a}{b})^{k-1}\right]$$

$$= a^k T(1) + n\left[\frac{1-(\frac{a}{b})^k}{1-\frac{a}{b}}\right] = a^k \cdot T(1) + n(\frac{b}{b-a})(1-\frac{a^k}{b^k})$$

$$= a^k \cdot 1 + n(\frac{b}{b-a})(\frac{b^k - a^k}{b^k})$$

$$= a^k + n(\frac{b}{b-a})(\frac{b^k - a^k}{n}) \qquad \because n = b^k$$

$$= a^k + (\frac{b}{b-a})(b^k - a^k)$$

$$= a^k + \frac{b}{b-a} \cdot b^k - \frac{b}{b-a} \cdot a^k$$

$$= \frac{b}{b-a} \cdot b^k + (\frac{b-a-b}{b-a})a^k$$

$$T(n) = (\frac{b}{b-a}) \cdot b^k - \frac{a}{b-a} \cdot a^{\log_b^n}$$

$$= \frac{b}{b-a} \cdot b^k - \frac{a}{b-a} \cdot n^{\log_b^a} \qquad \text{let} \quad \alpha = \log_b^a$$

$$\because a < b \Rightarrow \log_b^a < 1 \qquad \Rightarrow 0 < \alpha < 1$$

$$\Rightarrow T(n) = \frac{b}{b-a} \cdot n - \frac{a}{b-a} \cdot n^{\alpha} \Rightarrow T(n) \in \theta(n)$$

<u>More Hard Example using Substitution Method</u>

Solve the recurrence relation given below.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3 \cdot T(\frac{n}{4}) + cn^2 & \text{otherwise} \end{cases}$$

<u>Solution:</u>

(1) $T(n) = 3 \cdot T(\frac{n}{4}) + cn^2$

(2) $T(\frac{n}{4}) = 3 \cdot T(\frac{n}{4^2}) + c(\frac{n}{4})^2$

(3) $T(\frac{n}{4^2}) = 3 \cdot T(\frac{n}{4^3}) + c(\frac{n}{4^2})^2$

(4) $T(\frac{n}{4^3}) = 3 \cdot T(\frac{n}{4^4}) + c(\frac{n}{4^3})^2$

and so on

https://studyc.info/

(5) $T(\frac{n}{4^{k-1}}) = 3 \cdot T(\frac{n}{4^k}) + c(\frac{n}{4^{k-1}})^2$

Now substitute the value of $T(\frac{n}{4})$ from (2) to (1)

(6) $T(n) = 3\left[3T(\frac{n}{4^2}) + c(\frac{n}{4})^2\right] + cn^2$

$$= 3^2 T(\frac{n}{4^2}) + 3c(\frac{n}{4})^2 + cn^2$$

substitute the value of $T(\frac{n}{4^2})$ from (3) to (6), we have

$$T(n) = 3^2\left[3T(\frac{n}{4^3}) + c(\frac{n}{4^2})^2\right] + 3c \cdot (\frac{n}{4})^2 + cn^2$$

$$= 3^3 T(\frac{n}{4^3}) + 3^2 c(\frac{n}{4^2})^2 + 3c \cdot (\frac{n}{4})^2 + cn^2$$

$$= 3^3 T(\frac{n}{4^3}) + (\frac{3}{4^2})^2 cn^2 + (\frac{3}{4^2})^1 cn^2 + cn^2$$

After continuing this process, we get

$$T(n) = 3^k T(\frac{n}{4^k}) + \left[(\frac{3}{4^2})^{k-1} + (\frac{3}{4^2})^{k-2} + \cdots + (\frac{3}{4^2})^0\right] cn^2$$

Let us suppose that n can be expressed as $n = 4^k$

$$T(n) = 3^k T(1) + cn^2\left[1 + \frac{3}{16} + (\frac{3}{16})^2 + \cdots + (\frac{3}{16})^{k-1}\right]$$

$$T(n) = 3^k \cdot 1 + cn^2\left[(\frac{1-(\frac{3}{16})^k}{1-\frac{3}{16}})\right]$$

$$= 3^k + cn^2 \cdot \frac{16}{13}(1 - \frac{3^k}{16^k})$$

since $4^k = n \Rightarrow (4^k)^2 = n^2$

$$\Rightarrow (4^2)^k = n^2 \quad \Rightarrow 16^k = n^2$$

$$T(n) = 3^k + cn^2 \cdot \frac{16}{13}(1 - \frac{3^k}{16^k})$$

$$= 3^k + cn^2 \cdot \frac{16}{13}(\frac{n^2 - 3^k}{n^2}) = 3^k + \frac{16}{13} \cdot c \cdot (n^2 - 3^k) = \frac{16}{13} c \cdot n^2 + (1 - \frac{16}{13}c)3^k$$

$$T(n) = \frac{16}{13}cn^2 + (1 - \frac{16}{13}c)3^{\log_4^n}$$

$$= \frac{16}{13}cn^2 + (1 - \frac{16}{13}c)n^{\log_4 3}$$

let $\log_4^3 = \alpha$ where $0 < \alpha < 1$

$$T(n) = \frac{16}{13}cn^2 + (1 - \frac{16}{13}c)n^\alpha$$

Hence $T(n) \in \theta(n^2)$

Observations:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ a \cdot T(\frac{n}{b}) + cn^k & \text{otherwise} \end{cases}$$

$$= en^k + fn^{\log_b a} \qquad \text{suppose } b^{k_1} = a$$

$$= en^k + fn^{\log_b b^{k_1}}$$

$$= en^k + f.n^{k_1.\log_b b} = en^k + f.n^{k_1}$$

$$= \Theta(n^{\max(k,k_1)})$$

Recursion Tree Method

- Although substitution method can provide a sufficient proof that a solution to a recurrence is correct, sometimes difficult to give a good guess.
- Drawing out a recursion tree, is a straight forward way to devise a good guess.
- In ***recursion tree***, nodes represent costs of a sub-problems in the set of recursive function invocations.
- We sum costs within each level of the tree to obtain a set of per-level costs.
- And then we sum all per-level costs to determine the total cost of all levels of the recursion.
- Recursion trees are particularly useful when recurrence describes running time of divide and conquer algos.
- Recursion tree is best one used to generate a good guess, which is then verified by substitution method
- When using a recursion tree to generate a good guess, we can often tolerate a small amount of sloppiness since we have to verify it later on.
- If we are careful when drawing out a recursion tree and summing costs, then we can use a recursion tree as a direct proof of a solution to any recurrence of any problem.
- Here, we will use recursion trees directly to prove theorem that forms the basis of the master method.

Example:

Solve the following recurrence using recurrence tree method.

$$T(n) = \begin{cases} \Theta(1) & \text{if} \quad n = 1 \\ 3.T(\frac{n}{4}) + \Theta(n^2) & \text{if} \quad \text{otherwise} \end{cases}$$

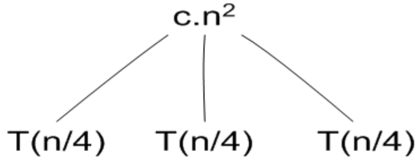
Solution:

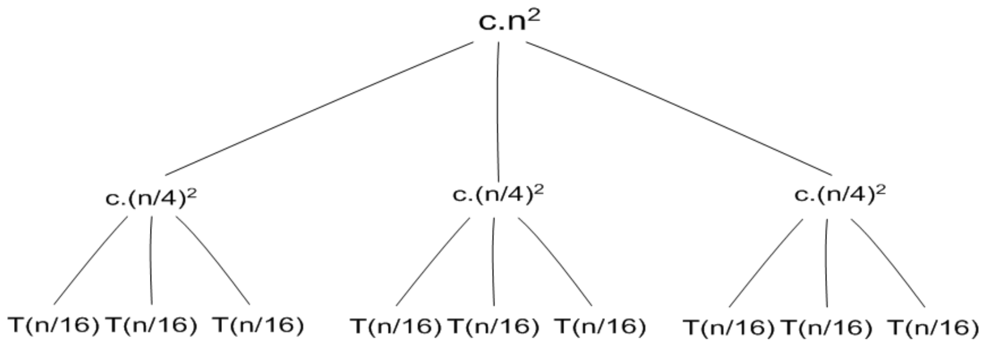
The above recurrence can be written in the form

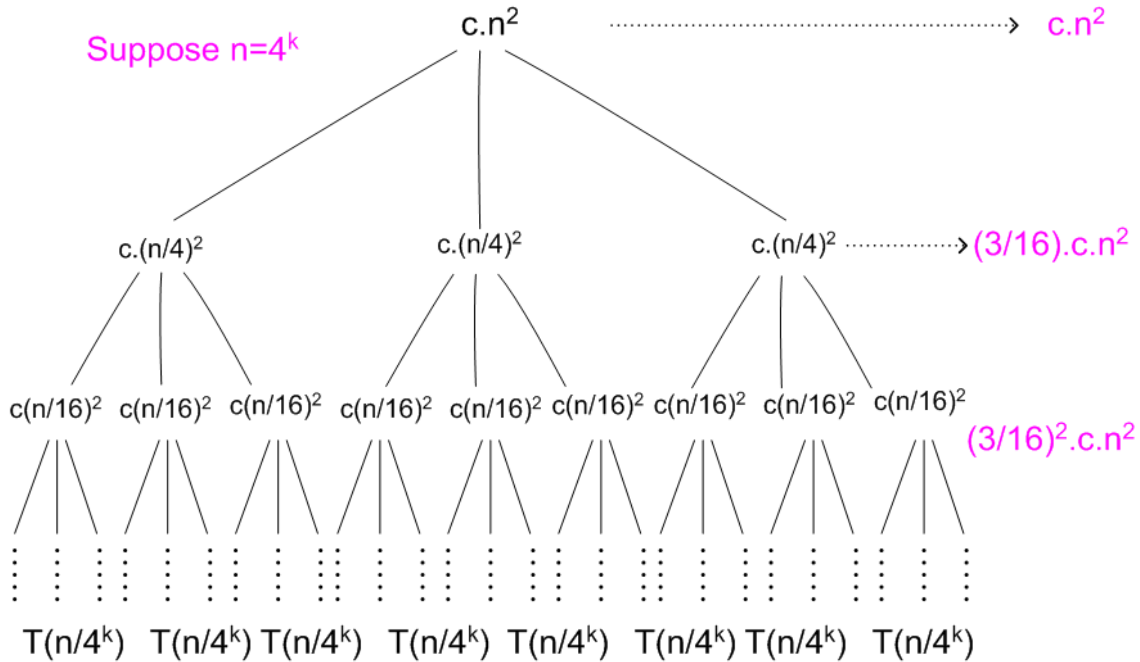
https://studyc.info/

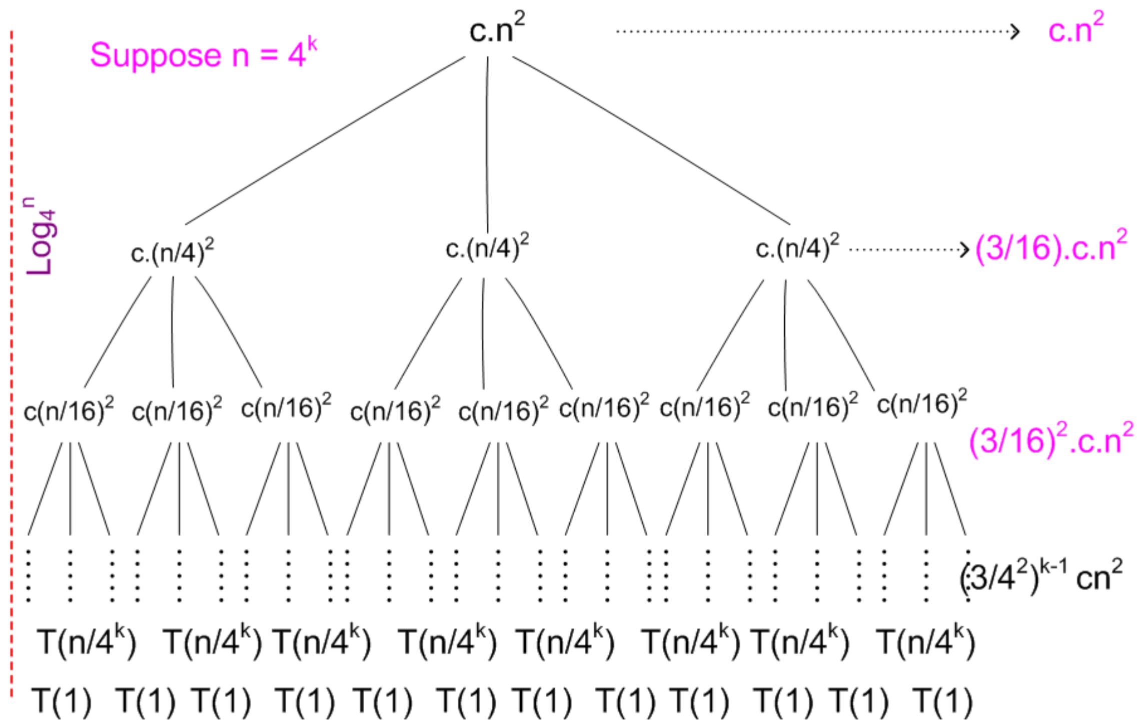
$$T(n) = \begin{cases} 1 & if \quad n=1 \\ 3.T(\frac{n}{4}) + cn^2 & if \quad otherwise \end{cases}$$

Assumption: We assume that n is exact power of 4.









Now the total computation cost would be = Cost of Childs + Cost of tree excluding childes
= Cost of Child x total number of Childs + Cost of all levels excluding childes level
= total number of Childs + sum of costs at each level excluding childes level.

$T(n) = 3^{\log_4^n}$ + cost at Levels above child level

$$T(n) = \Theta(n^{\log_3^4}) + \left[(\frac{3}{4^2})^0 + (\frac{3}{4^2})^1 + \cdots + (\frac{3}{4^2})^{k-1} \right] cn^2$$

Now total computational cost can be calculated as

$$T(n) = \Theta(n^{\log_3^4}) + \left[(\frac{3}{4^2})^0 + (\frac{3}{4^2})^1 + \cdots + (\frac{3}{4^2})^{k-1} \right] cn^2$$

where $4^k = n \Rightarrow k = \log_4 n$

$$T(n) \le \Theta(n^{\log_3^4}) + \left[(\frac{3}{4^2})^0 + (\frac{3}{4^2})^1 + \cdots \right] cn^2$$

$$T(n) \le \Theta(n^{\log_3^4}) + (\frac{1}{1-(\frac{3}{16})})cn^2 = \Theta(n^{\log_3^4}) + \frac{16}{13}cn^2$$

Hence $T(n) = O(n^2)$

Master Theorem

Lemma 1:

Let a ≥ 1, b > 1 be constants, f(n) a non-negative function defined on exact power of b by recurrence

$$T(n) = \begin{cases} \Theta(1) & if \quad n = 1 \\ a.T(\frac{n}{b}) + f(n) & if \quad n = b^i \end{cases}$$

where i is a positive integer. Then

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b^n - 1} a^j . f(\frac{n}{b^j})$$

Lemma 2:

Let a ≥ 1, b > 1 be constants, let f(n) be a non-negative function defined on exact power of b. A function g(n) defined over exact powers of b by

$$g(n) = \sum_{j=0}^{\log_b^n - 1} a^j . f(\frac{n}{b^j})$$

can be bounded asymptotically for exact powers of b as

1. If $f(n) = O(n^{\log_b^{a-\varepsilon}})$, for some constants $\varepsilon >$ 0, then $g(n) = O(n^{\log_b^a})$
2. If $f(n) = \Theta(n^{\log_b^a})$, then $g(n) = \Theta(n^{\log_b^a} .\lg\ n)$
3. If a.f(n/b) ≤ c.f(n), for some constant c < 1 and for all n ≥ b, then g(n) = Θ(f(n))

Proof:

Case 1: $f(n) = O(n^{\log_b^{a-\varepsilon}})$

Which implies that $f(\frac{n}{b^j}) = O((\frac{n}{b^j})^{\log_b^{a-\varepsilon}})$ (1)

We are given that: $g(n) = \sum_{j=0}^{\log_b^n - 1} a^j . f(\frac{n}{b^j})$ (2)

Substituting value from Equation (1) in Equation (2)

$$g(n) = O(\sum_{j=0}^{\log_b^n - 1} a^j . (\frac{n}{b^j})^{\log_b^{a-\varepsilon}})$$

Consider $\sum_{j=0}^{\log_b^n - 1} a^j . (\frac{n}{b^j})^{\log_b^{a-\varepsilon}} = n^{\log_b^{a-\varepsilon}} . \sum_{j=0}^{\log_b^n - 1} (\frac{a.b^{\varepsilon}}{b^{\log_b^a}})^j$

Assume that $\frac{a}{b^{\log_b^a}} = x$

taking log on both sides $\log_b(\frac{a}{b^{\log_b^a}}) = \log_b x$

$$\log_b^a - \log_b^{b^{\log_b^a}} = \log_b x \Leftrightarrow \log_b^a - \log_b^a . \log_b^b = \log_b x$$

$$\log_b^a - \log_b^a = \log_b x \Leftrightarrow 0 = \log_b x \Leftrightarrow x = 1$$

$$\sum_{j=0}^{\log_b^n - 1} a^j . (\frac{n}{b^j})^{\log_b^{a-\varepsilon}} = n^{\log_b^{a-\varepsilon}} . \sum_{j=0}^{\log_b^n - 1} (\frac{a.b^{\varepsilon}}{b^{\log_b^a}})^j = n^{\log_b^{a-\varepsilon}} . \sum_{j=0}^{\log_b^n - 1} (b^{\varepsilon})^j$$

$$= n^{\log_b^{a-\varepsilon}} . ((b^{\varepsilon})^0 + (b^{\varepsilon})^1 + ... + (b^{\varepsilon})^{\log_b^n - 1})$$

https://studyc.info/

It is a geometric series with first term 1, common ratio b^{ε} and number of terms as $\log_b n$.
Hence g(n),

$$= n^{\log_b^{a-\varepsilon}} (\frac{(b^{\varepsilon})^{\log_b^n} - 1}{b^{\varepsilon} - 1}) = n^{\log_b^{a-\varepsilon}} (\frac{b^{\varepsilon.\log_b n} - 1}{b^{\varepsilon} - 1}) = n^{\log_b^{a-\varepsilon}} (\frac{n^{\varepsilon.\log_b b} - 1}{b^{\varepsilon} - 1})$$

$$= n^{\log_b^{a-\varepsilon}} (\frac{n^{\varepsilon} - 1}{b^{\varepsilon} - 1}) = c.n^{\log_b^a} - c.n^{\log_b^{a-\varepsilon}} = O(n^{\log_b^a})$$ Hence proved

<u>Case 2:</u> $f(n) = \Theta(n^{\log_b^a})$

Which implies that $f(\frac{n}{b^j}) = \Theta((\frac{n}{b^j})^{\log_b^a})$ (3)

We are given that $g(n) = \sum_{j=0}^{\log_b^n - 1} a^j.f(\frac{n}{b^j})$ (4)

Substituting value from Equation (3) in Equation (4)

$$g(n) = \Theta(\sum_{j=0}^{\log_b^n - 1} a^j.(\frac{n}{b^j})^{\log_b^a})$$

$$g(n) = \Theta(\sum_{j=0}^{\log_b^n - 1} a^j.(\frac{n}{b^j})^{\log_b^a}) = \Theta(n^{\log_b^a}.\sum_{j=0}^{\log_b^n - 1} (\frac{a}{b^{\log_b^a}})^j)$$

We have already proved that $\frac{a}{b^{\log_b^a}} = 1$

$$g(n) = \Theta(n^{\log_b^a}.\sum_{j=0}^{\log_b^n - 1} 1) = (n^{\log_b^a}.\underbrace{(1 + 1 + ... + 1}_{\log_b^n \text{ number of terms}})$$

$= \Theta(n^{\log_b^a}.\log_b{}^n)$ case is proved

<u>Case 3:</u>

Given that

$$a.f(\frac{n}{b}) \le c.f(n) \Rightarrow f(\frac{n}{b}) \le \frac{c}{a}.f(n)$$

$$f(\frac{n}{b^2}) \le \frac{c}{a}.f(\frac{n}{b}) \le (\frac{c}{a})^2.f(n) \Rightarrow f(\frac{n}{b^2}) \le (\frac{c}{a})^2.f(n)$$

$$f(\frac{n}{b^3}) \le (\frac{c}{a})^2.f(\frac{n}{b}) \le (\frac{c}{a})^3.f(n) \Rightarrow f(\frac{n}{b^3}) \le (\frac{c}{a})^3.f(n)$$

In general $f(\frac{n}{b^j}) \le (\frac{c}{a})^j.f(n)$

Equivalently $a^j.f(\frac{n}{b^j}) \le c^j.f(n)$

We are given that $g(n) = \sum_{j=0}^{\log_b^n - 1} a^j.f(\frac{n}{b^j})$ (5)

We have proved that $a^j.f(\frac{n}{b^j}) \le c^j.f(n)$ (6)

https://studyc.info/

From equation (5) and (6)

$$g(n) = \sum_{j=0}^{\log_b^n - 1} a^j . f(\frac{n}{b^j}) \leq \sum_{j=0}^{\log_b^n - 1} c^j . f(n) \leq \sum_{j=0}^{\infty} c^j . f(n)$$

$$g(n) = f(n)(\frac{1}{1-c}) = O(f(n)) \qquad (7)$$

Since f(n) appears in definition of g(n) and all terms of g(n) are non-negative, we can conclude easily that

$$g(n) = \Omega(f(n)) \qquad (8)$$

We have already proved, equation (7), that

$$g(n) = O(f(n)) \qquad (9)$$

From Equations (8) and (9) $g(n) = \Theta(f(n))$

Hence it proves lemma

Lemma 3:

Let a ≥ 1, b > 1 be constants, let f(n) be a non-negative function defined on exact power of b. Define T(n) on exact powers of b by the recurrence

$$T(n) = \begin{cases} \Theta(1) & if \quad n = 1 \\ a.T(\frac{n}{b}) + f(n) & if \quad n = b^i \end{cases}$$

where is a positive integer. Then T(n) can be bounded asymptotically for exact powers of b as

1. If $f(n) = O(n^{\log_b^{a-\varepsilon}})$, for some constants $\varepsilon >$ 0, then $T(n) = \Theta(n^{\log_b^a})$
2. If $f(n) = \Theta(n^{\log_b^a})$, then $T(n) = \Theta(n^{\log_b^a} .\lg\ n)$
3. If $f(n) = \Omega(n^{\log_b^{a+\varepsilon}})$ for some ε > 0, and a.f(n/b) ≤ c.f(n) for some constant c < 1and sufficiently large n, then T(n) = Θ(f(n))

Proof Case 1:

Given that

$$T(n) = \begin{cases} \Theta(1) & if \quad n = 1 \\ a.T(\frac{n}{b}) + f(n) & if \quad n = b^i \end{cases}$$

By lemma 4.2: $T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b^n - 1} a^j . f(\frac{n}{b^j})$

By lemma 4.3: $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a})$

Hence for case 1 it is proved

Proof Case 2:

Again given that

https://studyc.info/

$$T(n) = \begin{cases} \Theta(1) & if \quad n = 1 \\ a.T(\frac{n}{b}) + f(n) & if \quad n = b^i \end{cases}$$

By Lemma 4.2: $T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b^n - 1} a^j . f(\frac{n}{b^j})$

By lemma 4.3: $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a}.\lg n)$

Hence for case 2 it is also proved

<u>Proof Case 3:</u>

By Lemma 4.2: $T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b^n - 1} a^j . f(\frac{n}{b^j})$

By Lemma 4.3: $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n))$

Since $f(n) = \Omega(n^{\log_b^{a+\varepsilon}})$

Hence $T(n) = \Theta(f(n))$

This proves the lemma 3.

https://studyc.info/

Lecture 10 Time Complexity of Algorithms (Asymptotic Notations)

What is Complexity?

The level in difficulty in solving mathematically posed problems as measured by

- The time (time complexity)
- number of steps or arithmetic operations (computational complexity)
- memory space required (space complexity)

Major Factors in Algorithms Design

1. Correctness

An algorithm is said to be correct if

- For every input, it halts with correct output.
- An incorrect algorithm might not halt at all OR
- It might halt with an answer other than desired one.
- Correct algorithm solves a computational problem

2. Algorithm Efficiency

Measuring efficiency of an algorithm,

- do its analysis i.e. growth rate.
- Compare efficiencies of different algorithms for the same problem.

Algorithms growth rate:

Algorithm Growth Rates

It measures algorithm efficiency

What means by efficient?

If running time is bounded by polynomial in the input.

Notations for Asymptotic performance

- How running time increases with input size
- O, Omega, Theta, etc. for asymptotic running time
- These notations defined in terms of functions whose domains are natural numbers
- convenient for worst case running time
- Algorithms, asymptotically efficient best choice

Complexity Analysis:

Algorithm analysis means predicting resources such as computational time, memory, computer hardware etc.

Worst case analysis:

- Provides an upper bound on running time
- An absolute guarantee

Average case analysis:

- Provides the expected running time

https://studyc.info/

- Very useful, but treat with care: what is "average"?
- Random (equally likely) inputs
- Real-life inputs

Worst case analysis:
Let us suppose that
D_n = set of inputs of size n for the problem
I = an element of D_n.
t(I) = number of basic operations performed on I
Define a function W by

$$W(n) = \max\{t(I) \mid I \in D_n\}$$

called the worst-case complexity of the algorithm
W(n) is the maximum number of basic operations performed by the algorithm on any input of size n.
Please note that the input, I, for which an algorithm behaves worst, depends on the particular algorithm.
Average Complexity:

Let Pr(I) be the probability that input I occurs.

Then the average behavior of the algorithm is defined as

$A(n) = \Sigma Pr(I)\, t(I)$, summation over all $I \in D_n$
We determine t(I) by analyzing the algorithm, but Pr(I) cannot be computed analytically.

Average cost = A(n) = Pr(succ)Asucc(n) + Pr(fail)Afail(n)

An element I in D_n may be thought as a set or equivalence class that affects the behavior of the algorithm

Worst Analysis computing average cost
Take all possible inputs, compute their cost, and take average

Asymptotic Notations Properties

- Categorize algorithms based on asymptotic growth rate e.g. linear, quadratic, polynomial, exponential
- Ignore small constant and small inputs
- Estimate upper bound and lower bound on growth rate of time complexity function
- Describe running time of algorithm as n grows to ∞.
- Describes behavior of function within the limit.

Limitations
- not always useful for analysis on fixed-size inputs.
- All results are for *sufficiently large* inputs.

Asymptotic Notations

Asymptotic Notations Θ, O, Ω, o, ω
- We use Θ to mean "order exactly",
- O to mean "order at most",
- Ω to mean "order at least",
- o to mean "tight upper bound",

https://studyc.info/

- ω to mean "tight lower bound",

Define a *set* of functions: which is in practice used to compare two function sizes.

Big-Oh Notation (O)

If f, g: $N \rightarrow R^+$, then we can define Big-Oh as
For a given function $g(n) \geq 0$, denoted by $O(g(n))$ the set of functions,

$O(g(n)) = \{f(n)$: there exist positive constants c and n_o such that

$0 \leq f(n) \leq cg(n)$, for all $n \geq n_o\}$

$f(n) = O(g(n))$ means function $g(n)$ is an asymptotically upper bound for $f(n)$.

We may write $f(n) = O(g(n))$ OR $f(n) \in O(g(n))$

Intuitively:
Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

Example 1:

Prove that $2n^2 \in O(n^3)$

Proof:

Assume that $f(n) = 2n^2$, and $g(n) = n^3$

Now we have to find the existence of c and n_0

$f(n) \leq c.g(n) \Rightarrow 2n^2 \leq c.n^3 \Rightarrow 2 \leq c.n$

if we take, $c = 1$ and $n_0 = 2$ OR

$c = 2$ and $n_0 = 1$ then

$2n^2 \leq c.n^3$

Hence $f(n) \in O(g(n))$, $c = 1$ and $n_0 = 2$

Example 2:

Example 2: Prove that $n^2 \in O(n^2)$
Proof:

Assume that $f(n) = n^2$, and $g(n) = n^2$

Now we have to show that $f(n) \in O(g(n))$

Since

$f(n) \leq c.g(n) \Rightarrow n^2 \leq c.n^2 \Rightarrow 1 \leq c$, take, $c = 1$, $n_0 = 1$

Then

$n^2 \leq c.n^2$ for $c = 1$ and $n \geq 1$

Hence, $2n^2 \in O(n^2)$, where $c = 1$ and $n_0 = 1$

https://studyc.info/

Example 3:

Prove that $1000.n^2 + 1000.n \in O(n^2)$

Proof:

Assume that $f(n) = 1000.n^2 + 1000.n$, and $g(n) = n^2$

We have to find existence of c and n_0 such that

$0 \leq f(n) \leq c.g(n)$ A $n \geq n_0$

$1000.n^2 + 1000.n \leq c.n^2 = 1001.n^2$, for $c = 1001$

$1000.n^2 + 1000.n \leq 1001.n^2$

$\Rightarrow 1000.n \leq n^2 \Rightarrow n^2 \geq 1000.n \Rightarrow n^2 - 1000.n \geq 0$

$\Rightarrow n(n-1000) \geq 0$, this true for $n \geq 1000$

$f(n) \leq c.g(n)$ A $n \geq n_0$ and $c = 1001$

Hence $f(n) \in O(g(n))$ for $c = 1001$ and $n_0 = 1000$

Example 4:

Prove that $n^3 \in O(n^2)$

Proof:

On contrary we assume that there exist some positive constants c and n_0 such that

$0 \leq n^3 \leq c.n^2$ A $n \geq n_0$

$0 \leq n^3 \leq c.n^2 \Rightarrow n \leq c$

Since c is any fixed number and n is any arbitrary constant, therefore $n \leq c$ is not possible in general.

Hence our supposition is wrong and $n^3 \leq c.n^2$,

A $n \geq n_0$ is not true for any combination of c and n_0. And hence, $n^3 \in O(n^2)$

Some more Examples:

- $n^2 + n^3 = O(n^4)$
- $n^2 / \log(n) = O(n . \log n)$
- $5n + \log(n) = O(n)$
- $n^{\log n} = O(n^{100})$
- $3^n = O(2^n . n^{100})$
- $n! = O(3^n)$
- $n + 1 = O(n)$
- $2n+1 = O(2n)$
- $(n+1)! = O(n!)$
- $1 + c + c^2 + \ldots + c^n = O(c^n)$ for $c > 1$
- $1 + c + c^2 + \ldots + c^n = O(1)$ for $c < 1$

Big-Omega Notation (Ω)

If f, g: $N \rightarrow R^+$, then we can define Big-Omega as

https://studyc.info/

For a given function $g(n)$ denote by $\Omega(g(n))$ the set of functions,

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_o such that

$0 \le cg(n) \le f(n)$ for all $n \ge n_o\}$

$f(n) = \Omega(g(n))$, means that function $g(n)$ is an asymptotically lower bound for $f(n)$.

We may write f(n) = Ω(g(n)) OR f(n) ∈ Ω(g(n))
Intuitively:
Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

<u>Example 1:</u> Prove that $5.n^2 \in \Omega(n)$

<u>Proof:</u>
Assume that $f(n) = 5.n^2$, and $g(n) = n$
We have to find the existence of c and n_0 such that

$c.g(n) \le f(n)$ for all $n \ge n_0$

$c.n \le 5.n^2 \Rightarrow c \le 5.n$

if we take, $c = 5$ and $n_0 = 1$ then

$c.n \le 5.n^2$ A $n \ge n_0$

And hence $f(n) \in \Omega(g(n))$, for $c = 5$ and $n_0 = 1$

<u>Example 2:</u> Prove that $100.n + 5 \notin \Omega(n^2)$

<u>Proof:</u>

Let $f(n) = 100.n + 5$, and $g(n) = n^2$

Assume that $f(n) \in \Omega(g(n))$

Now if $f(n) \in \Omega(g(n))$ then there exist c and n_0 such that

$c.g(n) \le f(n)$ for all $n \ge n_0$

$c.n^2 \le 100.n + 5$

$c.n \le 100 + 5/n$

$n \le 100/c$, for a very large n, which is not possible

And hence $f(n) \notin \Omega(g(n))$

<u>Theta Notation (Θ)</u>

If f, g: $N \to R^+$, then we can define Big-Theta as
For a given function $g(n)$ denoted by $\Theta(g(n))$ the set of functions,

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_o such that

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_o\}$

$f(n) = \Theta(g(n))$ means function $f(n)$ is equal to $g(n)$ to within a constant factor, and $g(n)$ is an asymptotically tight bound for $f(n)$.

We may write f(n) = Θ(g(n)) OR f(n) ∈ Θ(g(n))
Intuitively: Set of all functions that have same *rate of growth* as $g(n)$.

https://studyc.info/

Example 1: Prove that $\frac{1}{2}.n^2 - \frac{1}{2}.n = \Theta(n^2)$

Proof:

Assume that $f(n) = \frac{1}{2}.n^2 - \frac{1}{2}.n$, and $g(n) = n^2$

We have to find the existence of c_1, c_2 and n_0 such that

$c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n \geq n_0$

Since, $\frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2$ $\forall n \geq 0$ if $c_2 = \frac{1}{2}$ and

$\frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n . \frac{1}{2} n$ ($\forall n \geq 2$) $= \frac{1}{4} n^2$, $c_1 = \frac{1}{4}$

Hence $\frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \leq \frac{1}{2} n^2 - \frac{1}{2} n$

$c_1.g(n) \leq f(n) \leq c_2.g(n)$ $\forall n \geq 2$, $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$

Hence $f(n) \in \Theta(g(n)) \Rightarrow \frac{1}{2}.n^2 - \frac{1}{2}.n = \Theta(n^2)$

Example 2: Prove that $a.n^2 + b.n + c = \Theta(n^2)$ where a, b, c are constants and $a > 0$

Proof

If we take $c_1 = \frac{1}{4}.a$, $c_2 = 7/4.a$ and $n_0 = 2.\max((|b|/a), \sqrt{(|c|/a)})$

Then it can be easily verified that

$0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n), \forall n \geq n_0$, $c_1 = \frac{1}{4}.a$, $c_2 = 7/4.a$

Hence $f(n) \in \Theta(g(n)) \Rightarrow a.n^2 + b.n + c = \Theta(n^2)$

Hence any polynomial of degree 2 is of order $\Theta(n^2)$

Example 3: Prove that $2.n^2 + 3.n + 6 = \Theta(n^3)$

Proof:

Let $f(n) = 2.n^2 + 3.n + 6$, and $g(n) = n^3$

we have to show that f(n) ä $\Theta(g(n))$

On contrary assume that $f(n) \in \Theta(g(n))$ i.e.

there exist some positive constants c_1, c_2 and n_0 such that: $c_1.g(n) \leq f(n) \leq c_2.g(n)$

$c_1.g(n) \leq f(n) \leq c=.g(n) \Rightarrow c_1.n^3 \leq 2.n^2 + 3.n + 6 \leq c_2. n^3 \Rightarrow$

$c_1.n \leq 2 + 3/n + 6/n^2 \leq c_2. n \Rightarrow$

$c_1.n \leq 2 \leq c_2. n$, for large $n \Rightarrow$

$n \leq 2/c_1 \leq c_2/c_1.n$ which is not possible

Hence $f(n) = \Theta(g(n)) \Rightarrow 2.n^2 + 3.n + 6 = \Theta(n^3)$

Little-Oh Notation

o-notation is used to denote a upper bound that is not asymptotically tight.

https://studyc.info/

For a given function $g(n) \geq 0$, denoted by $o(g(n))$ the set of functions,

$$o(g(n)) = \begin{cases} f(n): \text{for any positive constants } c, \text{ there exists a constant } n_o \\ \text{such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_o \end{cases}$$

f(n) becomes insignificant relative to g(n) as n approaches infinity

e.g., $2n = o(n^2)$ but $2n^2 \neq o(n^2)$.. $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

g(n) is an upper bound for f(n), not asymptotically tight

Example 1: Prove that $2n^2 \in o(n^3)$

Proof:

Assume that $f(n) = 2n^2$, and $g(n) = n^3$

Now we have to find the existence n_0 for any c
f(n) < c.g(n) this is true
⇨$2n^2 < c.n^3$ ⇨ $2 < c.n$
This is true for any c, because for any arbitrary c we can choose n_0 such that the above inequality holds.

Hence $f(n) \in o(g(n))$

Example 2: Prove that $n^2 \notin o(n^2)$

Proof:

Assume that $f(n) = n^2$, and $g(n) = n^2$
Now we have to show that $f(n) \notin o(g(n))$
Since
$f(n) < c.g(n)$ ⇨ $n^2 < c.n^2$ ⇨ $1 \leq c$,

In our definition of small o, it was required to prove for any c but here there is a constraint over c. Hence, $n^2 \notin o(n^2)$, where c = 1 and $n_0 = 1$

Example 3: Prove that $1000.n^2 + 1000.n \notin o(n^2)$

Proof:

Assume that $f(n) = 1000.n^2 + 1000.n$, and $g(n) = n^2$

we have to show that $f(n) \notin o(g(n))$ i.e.

We assume that for any c there exist n_0 such that

$0 \leq f(n) < c.g(n)$ for all $n \geq n_0$

$1000.n^2 + 1000.n < c.n^2$

If we take c = 2001, then, $1000.n^2 + 1000.n < 2001.n^2$

⇨$1000.n < 1001.n^2$ which is not true

Hence $f(n) \notin o(g(n))$ for c = 2001

https://studyc.info/

Little-Omega Notation ω

Little-ω notation is used to denote a lower bound that is not asymptotically tight.

For a given function $g(n)$, denote by $\omega(g(n))$ the set of all functions.

$\omega(g(n)) = \{f(n)$: for any positive constants c, there exists a constant n_o such that

$0 \leq cg(n) < f(n)$ for all $n \geq n_o\}$

$f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity

e.g., $\frac{n^2}{2} = \omega(n)$ but $\frac{n^2}{2} \neq \omega(n^2)$..

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Example 1: Prove that $5.n^2 \in \omega(n)$

Proof:

Assume that $f(n) = 5.n^2$, and $g(n) = n$

We have to prove that for any c there exists n_0 such that, $c.g(n) < f(n)$ for all $n \geq n_0$

$c.n < 5.n^2 \Rightarrow c < 5.n$

This is true for any c, because for any arbitrary c e.g. c = 1000000, we can choose n_0 = 1000000/5 = 200000 and the above inequality does hold.

And hence $f(n) \in \omega(g(n))$,

Example 2: Prove that $5.n + 10 \notin \omega(n)$

Proof:

Assume that $f(n) = 5.n + 10$, and $g(n) = n$

We have to find the existence n_0 for any c, s.t.

$c.g(n) < f(n)$ for all $n \geq n_0$

$c.n < 5.n + 10$, if we take c = 16 then

$16.n < 5.n + 10 \Leftrightarrow 11.n < 10$ is not true for any positive integer.

Hence $f(n) \notin \omega(g(n))$

Example 3: Prove that $100.n \notin \omega(n^2)$

Proof:

Let $f(n) = 100.n$, and $g(n) = n^2$

Assume that $f(n) \in \omega(g(n))$

Now if $f(n) \in \omega(g(n))$ then there n_0 for any c such that

$c.g(n) < f(n)$ for all $n \geq n_0$ this is true

$\Rightarrow c.n^2 < 100.n \Rightarrow c.n < 100$

If we take c = 100, n < 1, not possible

Hence $f(n) \notin \omega(g(n))$ i.e. $100.n \notin \omega(n^2)$

Usefulness of Notations

https://studyc.info/

- It is not always possible to determine behaviour of an algorithm using Θ-notation.
- For example, given a problem with n inputs, we may have an algorithm to solve it in $a.n^2$ time when n is even and c.n time when n is odd. OR
- We may prove that an algorithm never uses more than $e.n^2$ time and never less than f.n time.
- In either case we can neither claim Θ(n) nor $\Theta(n^2)$ to be the order of the time usage of the algorithm.
- Big O and Ω notation will allow us to give at least partial information

https://studyc.info/

Lecture 11 Relations Over Asymptotic Notations

Reflexive Relation:

Definition:

Let X be a non-empty set and R is a relation over X then R is said to be reflexive if

$$(a, a) \in R, \forall a \in X,$$

Example 1:

Let G be a graph. Let us define a relation R over G as if node x is connected to y then $(x, y) \in G$. Reflexivity is satisfied over G if for every node there is a self loop.

Example 2:

Let P be a set of all persons, and S be a relation over P such that if $(x, y) \in S$ then x has same birthday as y.

Of course this relation is reflexive because

$$(x, x) \in S, \quad \forall a \in P,$$

Reflexivity Relations over Θ, Ω, O

Example 1

Since, $0 \leq f(n) \leq cf(n) \quad \forall n \geq n_0 = 1, \text{ if } c = 1$

Hence $f(n) = O(f(n))$

Example 2

Since, $0 \leq cf(n) \leq f(n) \quad \forall n \geq n_0 = 1, \text{ if } c = 1$

Hence $f(n) = \Omega(f(n))$

Example 3

Since, $0 \leq c_1 f(n) \leq f(n) \leq c_2 f(n) \quad \forall n \geq n_0 = 1, \text{if } c_1 = c_2 = 1$

Hence $f(n) = \Theta(f(n))$

Note: All the relations, Q, W, *O*, are reflexive

Little o and ω are not Reflexivity Relations

Example

As we can not prove that $f(n) < f(n)$, for any n, and for all $c > 0$

Therefore

1. $f(n) \neq o(f(n))$ and
2. $f(n) \neq \omega(f(n))$

Hence small o and small omega are not reflexive relations

Symmetry

Definition:

Let X be a non-empty set and R is a relation over X then R is said to be symmetric if

$$\forall a, b \in X, (a, b) \in R \Rightarrow (b, a) \in R$$

Example 1:

https://studyc.info/

Let P be a set of persons, and S be a relation over P such that if $(x, y) \in S$ then x has the same sign as y.

This relation is symmetric because

$$(x, y) \in S \Rightarrow (y, x) \in S$$

Example 2:

Let P be a set of all persons, and B be a relation over P such that if $(x, y) \in B$ then x is brother of y.

This relation is not symmetric because

$$(\text{Anwer, Sadia}) \in B \Rightarrow (\text{Sadia, Brother}) \notin B$$

Symmetry over Θ

Property: prove that $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

Proof

Since $f(n) = \Theta(g(n))$ i.e. $f(n) \in \Theta(g(n)) \Rightarrow$

$\exists$ constants $c_1, c_2 > 0$ and $n_0 \in N$ such that

$0 \le c_1g(n) \le f(n) \le c_2g(n) \quad \forall\ n \ge n_0$ (1)

(1) $\Rightarrow 0 \le c_1g(n) \le f(n) \le c_2g(n) \Rightarrow 0 \le f(n) \le c_2g(n)$

$\Rightarrow 0 \le (1/c_2)f(n) \le g(n)$ (2)

(1) $\Rightarrow 0 \le c_1g(n) \le f(n) \le c_2g(n) \Rightarrow 0 \le c_1g(n) \le f(n)$

$\Rightarrow 0 \le g(n) \le (1/c_1)f(n)$ (3)

From (2),(3): $0 \le (1/c_2)f(n) \le g(n) \wedge 0 \le g(n) \le (1/c_1)f(n)$

$\Rightarrow \quad 0 \le (1/c_2)f(n) \le g(n) \le (1/c_1)f(n)$

Suppose that $1/c_2 = c_3$, and $1/c_1 = c_4$,

Now the above equation implies that

$0 \le c_3f(n) \le g(n) \le c_4f(n),\ \forall\ n \ge n_0$

$\Rightarrow g(n) = \Theta(f(n)),\ \forall\ n \ge n_0$

Hence it proves that,

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Transitivity

Definition:

Let X be a non-empty set and R is a relation over X then R is said to be transitive if

$$\forall\ a, b, c \in X, (a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$$

Example 1:

Let P be a set of all persons, and B be a relation over P such that if $(x, y) \in B$ then x is brother of y.

This relation is transitive this is because

$$(x, y) \in B \wedge (y, z) \in B \Rightarrow (x, z) \in B$$

Example 2:

https://studyc.info/

Let P be a set of all persons, and F be a relation over P such that if $(x, y) \in F$ then x is father of y.
Of course this relation is not a transitive because if $(x, y) \in F \wedge (y, z) \in F \Rightarrow (x, z) \notin F$

Transitivity Relation over Q

Property 1

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$

Proof

Since $f(n) = \Theta(g(n))$ i.e. $f(n) \in \Theta(g(n)) \Rightarrow$

$\exists$ constants $c_1, c_2 > 0$ and $n_{01} \in N$ such that

$0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \quad \forall\ n \geq n_{01}$ (1)

Now since $g(n) = \Theta(h(n))$ i.e. $g(n) \in \Theta(h(n)) \Rightarrow$

$\exists$ constants $c_3, c_4 > 0$ and $n_{02} \in N$ such that

$0 \leq c_3h(n) \leq g(n) \leq c_4h(n) \quad \forall\ n \geq n_{02}$ (2)

Now let us suppose that $n_0 = \max(n_{01}, n_{02})$

Now we have to show that $f(n) = \Theta(h(n))$ i.e. we have to prove that

$\exists$ constants $c_5, c_6 > 0$ and $n_0 \in N$ such that

$0 \leq c_5h(n) \leq f(n) \leq c_6h(n)$

(2) $\Rightarrow 0 \leq c_3h(n) \leq g(n) \leq c_4h(n)$

$\Rightarrow 0 \leq c_3h(n) \leq g(n)$ (3)

(1) $\Rightarrow 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$

$\Rightarrow 0 \leq c_1g(n) \leq f(n)$

$\Rightarrow 0 \leq g(n) \leq (1/c_1)f(n)$ (4)

From (3) and (4), $0 \leq c_3h(n) \leq g(n) \leq (1/c_1)f(n)$

$\Rightarrow 0 \leq c_1c_3h(n) \leq f(n)$ (5)

(1) $\Rightarrow 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$

$\Rightarrow 0 \leq f(n) \leq c_2g(n) \Rightarrow 0 \leq (1/c_2)f(n) \leq g(n)$ (6)

(2) $\Rightarrow 0 \leq c_3h(n) \leq g(n) \leq c_4h(n)$

$\Rightarrow 0 \leq g(n) \leq c_4h(n)$ (7)

From (6) and (7), $0 \leq (1/c_2)f(n) \leq g(n) \leq (c_4)h(n)$

$\Rightarrow 0 \leq (1/c_2)f(n) \leq (c_4)h(n)$

$\Rightarrow 0 \leq f(n) \leq c_2c_4h(n)$ (8)

From (5), (8), $0 \leq c_1c_3h(n) \leq f(n) \wedge 0 \leq f(n) \leq c_2c_4h(n)$

$0 \leq c_1c_3h(n) \leq f(n) \leq c_2c_4h(n)$

$0 \leq c_5h(n) \leq f(n) \leq c_6h(n)$

And hence $f(n) = \Theta(h(n)) \quad \forall\ n \geq n_0$

Transitivity Relation over Big *O*

Property 2

$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

https://studyc.info/

Proof

Since $f(n) = O(g(n))$ i.e. $f(n) \in O(g(n)) \Rightarrow$

$\exists$ constants $c_1 > 0$ and $n_{01} \in N$ such that

$0 \le f(n) \le c_1 g(n) \quad \forall\, n \ge n_{01} \qquad (1)$

Now since $g(n) = O(h(n))$ i.e. $g(n) \in O(h(n)) \Rightarrow$

$\exists$ constants $c_2 > 0$ and $n_{02} \in N$ such that

$0 \le g(n) \le c_2 h(n) \quad \forall\, n \ge n_{02} \qquad (2)$

Now let us suppose that $n_0 = \max(n_{01}, n_{02})$

Now we have to two equations

$0 \le f(n) \le c_1 g(n) \quad \forall\, n \ge n_{01} \qquad (1)$

$0 \le g(n) \le c_2 h(n) \quad \forall\, n \ge n_{02} \qquad (2)$

(2) $\Rightarrow 0 \le c_1 g(n) \le c_1 c_2 h(n) \quad \forall\, n \ge n_{02} \qquad (3)$

From (1) and (3)

$0 \le f(n) \le c_1 g(n) \le c_1 c_2 h(n)$

Now suppose that $c_3 = c_1 c_2$

$0 \le f(n) \le c_1 c_2 h(n)$

And hence $f(n) = O(h(n)) \quad \forall\, n \ge n_0$

Transitivity Relation over Big Ω

Property 3

$$f(n) = \Omega(g(n)) \ \&\ g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

Proof

Since $f(n) = \Omega(g(n)) \Rightarrow$

$\exists$ constants $c_1 > 0$ and $n_{01} \in N$ such that

$0 \le c_1 g(n) \le f(n) \quad \forall\, n \ge n_{01} \qquad (1)$

Now since $g(n) = \Omega(h(n)) \Rightarrow$

$\exists$ constants $c_2 > 0$ and $n_{02} \in N$ such that

$0 \le c_2 h(n) \le g(n) \quad \forall\, n \ge n_{02} \qquad (2)$

Suppose that $n_0 = \max(n_{01}, n_{02})$

We have to show that $f(n) = \Omega(h(n))$ i.e. we have to prove that

$\exists$ constants $c_3 > 0$ and $n_0 \in N$ such that

$0 \le c_3 h(n) \le f(n) \quad \forall\, n \ge n_0$

(2) $\Rightarrow 0 \le c_2 h(n) \le g(n)$

(1) $\Rightarrow 0 \le c_1 g(n) \le f(n)$

$\Rightarrow 0 \le g(n) \le (1/c_1) f(n) \qquad (3)$

From (2) and (3), $\quad 0 \le c_2 h(n) \le g(n) \le (1/c_1) f(n)$

$\Rightarrow 0 \le c_1 c_2 h(n) \le f(n)$ hence $f(n) = \Omega(h(n)),\ \forall\, n \ge n_0$

https://studyc.info/

<u>Transitivity Relation over *little o*</u>

Property 4

$f(n) = o(g(n))$ & $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$

<u>Proof</u>

Since $f(n) = o(g(n))$ i.e. $f(n) \in o(g(n)) \Rightarrow$

$\exists$ constants $c_1 > 0$ and $n_{01} \in N$ such that

$0 \le f(n) < c_1 g(n) \quad \forall\ n \ge n_{01} \qquad (1)$

Now since $g(n) = o(h(n))$ i.e. $g(n) \in o(h(n)) \Rightarrow$

$\exists$ constants $c_2 > 0$ and $n_{02} \in N$ such that

$0 \le g(n) < c_2 h(n) \quad \forall\ n \ge n_{02} \qquad (2)$

Now let us suppose that $n_0 = \max(n_{01}, n_{02})$

Now we have to two equations

$0 \le f(n) < c_1 g(n) \quad \forall\ n \ge n_{01} \qquad (1)$

$0 \le g(n) < c_2 h(n) \quad \forall\ n \ge n_{01} \qquad (2)$

(2) $\Rightarrow 0 \le c_1 g(n) < c_1 c_2 h(n) \quad \forall\ n \ge n_{02} \qquad (3)$

From (1) and (3)

$0 \le f(n) \le c_1 g(n) < c_1 c_2 h(n)$

Now suppose that $c_3 = c_1 c_2$

$0 \le f(n) < c_1 c_2 h(n)$

And hence $f(n) = o(h(n)) \quad \forall\ n \ge n_{01}$

<u>Transitivity Relation over little ω</u>

Property 5

$$f(n) = \omega(g(n)) \ \&\ g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

<u>Proof</u>

Since $f(n) = \omega(g(n)) \Rightarrow$

$\exists$ constants $c_1 > 0$ and $n_{01} \in N$ such that

$0 \le c_1 g(n) < f(n) \quad \forall\ n \ge n_{01} \qquad (1)$

Now since $g(n) = \omega(h(n)) \Rightarrow$

$\exists$ constants $c_2 > 0$ and $n_{02} \in N$ such that

$0 \le c_2 h(n) < g(n) \quad \forall\ n \ge n_{02} \qquad (2)$

Suppose that $n_0 = \max(n_{01}, n_{02})$

We have to show that $f(n) = \omega(h(n))$ i.e. we have to prove that

$\exists$ constants $c_3 > 0$ and $n_0 \in N$ such that

$0 \le c_3 h(n) \le f(n) \quad \forall\ n \ge n_0$

(2) $\Rightarrow 0 \le c_2 h(n) < g(n)$

(1) $\Rightarrow 0 \le c_1 g(n) < f(n)$

https://studyc.info/

$\Rightarrow 0 \le g(n) < (1/c_1)f(n)$ (3)

From (2) and (3), $0 \le c_2h(n) \le g(n) < (1/c_1)f(n)$

$\Rightarrow 0 \le c_1c_2h(n) < f(n)$ hence $f(n) = \omega(h(n))$, $\forall\ n \ge n_0$

Transpose Symmetry

Property 1

Prove that $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

Proof

Since $f(n) = O(g(n)) \Rightarrow$

$\exists$ constants $c > 0$ and $n_0 \in N$ such that

$0 \le f(n) \le cg(n) \quad \forall\ n \ge n_0$

Dividing both side by c

$0 \le (1/c)f(n) \le g(n) \quad \forall\ n \ge n_0$

Put $1/c = c'$

$0 \le c'f(n) \le g(n) \quad \forall\ n \ge n_0$

Hence, $g(n) = \Omega(f(n))$

Property 2

Prove that $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega f(n))$

Proof

Since $f(n) = o(g(n)) \Rightarrow$

$\exists$ constants $c > 0$ and $n_0 \in N$ such that

$0 \le f(n) < cg(n) \quad \forall\ n \ge n_0$

Dividing both side by c

$0 \le (1/c)f(n) < g(n) \quad \forall\ n \ge n_0$

Put $1/c = c'$

$0 \le c'f(n) < g(n) \quad \forall\ n \ge n_0$

Hence, $g(n) = \omega(f(n))$

Relation between Q, W , O

Trichotmy property over real numbers

For any two real numbers *a* and *b*, exactly one of the following must hold: $a < b$,$a = b$, or $a > b$.

The asymptotic comparison of two functions f and g and the comparison of two real numbers a and b.

Tracheotomy property over Q, W and *O*

1. $f(n) = O(g(n)) \quad \approx \quad a \le b$

2. $f(n) = \Omega(g(n)) \quad \approx \quad a \geq b$
3. $f(n) = \Theta(g(n)) \quad \approx \quad a = b$
4. $f(n) = o(g(n)) \quad \approx \quad a < b$
5. $f(n) = \omega(g(n)) \quad \approx \quad a > b$

Some other Standard Notations

Monotonicity

- monotonically increasing if $m \leq n \Rightarrow f(m) \leq f(n)$.
- monotonically decreasing if $m \leq n \Rightarrow f(m) \geq f(n)$.
- strictly increasing if $m < n \Rightarrow f(m) < f(n)$.
- strictly decreasing if $m < n \Rightarrow f(m) > f(n)$.

Polynomials

Given a positive integer d, a polynomial in n of degree d is a function of the form given below, a_i are coefficient of polynomial.

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

Standard Logarithms Notations

Some Definitions

Exponent: $x = \log_b a$ $x = \log_b a$ is the exponent for $x = a^b$

Natural log: $\ln a = \log_s a$ $\ln a = \log_e a$

Binary log: $\lg a = \log_2 a$

Square of log: $\lg^2 a = (\lg a)^2$

Log of Log: $\lg \lg a = \lg(\lg a)$

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n\log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b(1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

https://studyc.info/

Lecture 12 Design of Algorithms using Brute Force Approach

Primality Testing

(given number is n binary digits)

Brute Force Approach

```
Prime (n)
   for i ← 2 to n-1
        if n ≡ 0 mod i then
                 "number is composite"
   else
        "number is prime"
```

The computational cost is $\Theta(n)$

The computational cost in terms of binary operation is $\Theta(2^n)$

Refined Algorithm for Testing Primality

```
Prime (n)
   for i ← 2 to n/2
        if n ≡ 0 mod i then
                 "number is composite"
   else
        "number is prime"
```

The computational cost is $\Theta(n/2)$

The computational cost in terms of binary operation is $\Theta(2^{n-1})$, not much improvement

Algorithm for Testing Primality

- We are not interested, how many operations are required to test if the number n is prime
- In fact, we are interested, how many operations are required to test if a number with n digits is prime.
- RSA-128 encryption uses prime numbers which are 128 bits long. Where 2^{128} is:

  340282366920938463463374607431768211456
- Therefore, to prime test a number with n binary digits by brute force, we need to check 2^n numbers.
- Thus brute-force prime testing requires *exponential time* with the n number of digits.
- Which is not accepted in this case

Lemma:

Statement

- If $n \in N$, $n > 1$ is not prime then n is divisible by some prime number $p \leq$ square root of n.

https://studyc.info/

Proof

- Since n is not prime hence it can be factored as

  n = x.y where 1 < x ≤ y < n
- x or y is a prime, if not it can be further factored out.
- Also suppose without loss of generality that x ≤ y
- Now our claim is that x ≤ sq(n)
- This is because otherwise x.y > n, a contradiction
- We only require to check till sqr(n) for primality test.

Refined Algorithm for Testing Primality

Prime (n)

for i ← 2 **to** sqr(n)

 if n ≡ 0 mod i **then**

  "number is composite"

else

 "number is prime"

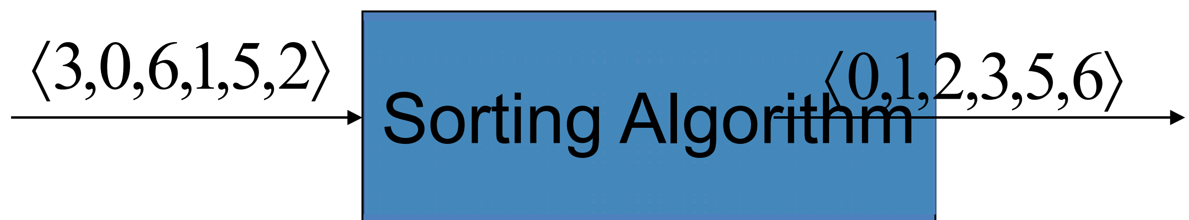
- The computational cost is Θ(sqr(n)), much faster
- The computational cost in terms of binary operation is $\Theta(2^{\sqrt{n}})$, still exponential

Sorting Sequence of Numbers:

An example of Algorithm:

Input : A sequence of n numbers (distinct) $\langle a_1, a_2, ..., a_n \rangle$

Output : A permutation, $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$



Sorting Algorithm: Brute Force Approach

Sort the array [2, 4, 1, 3] in increasing order

s1 = [4,3,2,1], s2 = [4,3,1,2], s3 = [4,1,2,3]

s4 = [4,2,3,1], s5 = [4,1,3,2], s6 = [4,2,1,3]

s7 = [3,4,2,1], s8 = [3,4,1,2], s9 = [3,1,2,4]

s10 = [3,2,4,1], s11 = [3,1,4,2], s12 = [3,2,1,4]

s13 = [2,3,4,1], s14 = [2,3,1,4], s15 = [2,1,4,3]

s16 = [2,4,3,1], s17 = [2,1,3,4], s18 = [2,4,1,3]

s19 = [1,3,2,4], s20 = [1,3,1,4], s21 = [1,4,2,3]

https://studyc.info/

s22 = [1,2,3,4], s23 = [1,4,3,2], s24 = [1,2,4,3]

There are 4! = 24 number of permutations.

For n number of elements there will be n! number of permutations. Hence cost of order n! for sorting.

Generating Permutations

```
Permute (i)                         \\initial call Permute(1)
  if i == N
      output A[N]
  else
      for j = i to N do
          swap(A[i], A[j])
          permute(i+1)
          swap(A[i], A[j])
```

- There are 4! = 24 number of permutations.
- For n number of elements there will be n! number of permutations. Hence cost of order n! for sorting.

Theorem

Prove, by mathematical induction, that computational cost of generating permutations is n!.

Proof

- If n = 1, then the statement is true, because 1! =1
- If there are k elements in set then no. of permutation = k!
- If we add one more element in any of the permutations, there will be k+1 number of ways to add it, resulting k+1 no. of permutations.
- Now total no. of permutations = k!(k+1) = (k+1)!
- Hence true for all n.

0-1 Knapsack Problem

Statement:

The knapsack problem arises whenever there is resource allocation with no financial constraints

Problem Statement

You are in Japan on an official visit and want to make shopping from a store (Best Denki). You have a list of required items. You have also a bag (knapsack), of fixed capacity, and only you can fill this bag with the selected items. Every item has a value (cost) and weight, and your objective is to seek most valuable set of items which you can buy not exceeding bag limit.

0-1 Knapsack Algorithm

https://studyc.info/

Knapsack-BF (n, V, W, C)

Compute all subsets, s, of S = {1, 2, 3, 4}

For all $s \in S$

weight = Compute sum of weights of these items

if weight > C, not feasible

new solution = Compute sum of values of these items

solution = solution $\cup$ {new solution}

Return maximum of solution

0-1 Knapsack Algorithm Analysis

Approach

- In brute force algorithm, we go through all combinations and find the one with maximum value and with total weight less or equal to $W = 16$

Complexity

- Cost of computing subsets $O(2^n)$ for n elements
- Cost of computing weight = $O(2^n)$
- Cost of computing values = $O(2^n)$
- Total cost in worst case: $O(2^n)$

The Closest Pair Problem

Finding closest pair

Problem

Given a set of n points, determine the two points that are closest to each other in terms of distance. Furthermore, if there are more than one pair of points with the closest distance, all such pairs should be identified.

Input:

is a set of n points

Output

- is a pair of points closest to each other,
- there can be more than one such pairs

Brute Force Approach: Finding Closest Pair in 2-D

ClosestPairBF(P)

1. mind $\leftarrow \infty$
2. **for** i $\leftarrow$ 1 to n
3. **do**
    4. **for** j $\leftarrow$ 1 to n
    5. **if** $i \neq j$
    6. **do**
    7. $d \leftarrow ((x_i - x_j)^2 + (y_i - y_j)^2)$
    8. **if** d < minn **then**

https://studyc.info/

8. mind ← d
9. mini ← i
10. minj ← j
11. **return** mind, p(mini, minj)

Time Complexity:

$$= \sum_{i=1}^{n}\sum_{j=1}^{n} c$$

$$= \sum_{i=1}^{n} cn$$

$$= cn^2$$

$$= \Theta(n^2)$$

Improved Version: Finding Closest Pair in 2-D

ClosestPairBF(P)

1. mind ← ∞
2. **for** i ← 1 to n − 1
3. **do**
4. **for** j ← i + 1 to n
5. **do**
6. d ← $((x_i - x_j)^2 + (y_i - y_j)^2)$
7. **if** d < minn **then**
8. mind ← d
9. mini ← i
10. minj ← j
11. **return** mind, p(mini, minj)

Time Complexity:

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} c$$

$$= \sum_{i=1}^{n-1} c(n-i)$$

$$= c(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i)$$

$$= cn(n-1) - c\frac{(n-1)n}{2}$$

$$= \Theta(n^2)$$

Brute Force Approach: Finding Closest Pair in 3-D

ClosestPairBF(P)

1. mind ← ∞
2. **for** i ← 1 to n − 1

https://studyc.info/

3. **do**
    4. **for** j ← i + 1 to n
    5. **do**
    6. d ← $((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2)$
    7. **if** d < minn **then**
        8. mind ← d
        9. mini ← i
        10. minj ← j
11. **return** mind, p(mini), p(minj)

Time Complexity

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c$$

$$= \sum_{i=1}^{n-1} c(n-i)$$

$$= c(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i)$$

$$= \Theta(n^2)$$

Maximal Points

- Dominated Point in 2-D

  A point p is said to be dominated by q if

  $p.x \le q.x$ and $p.y \le q.y$
- Dominated Point in n-D

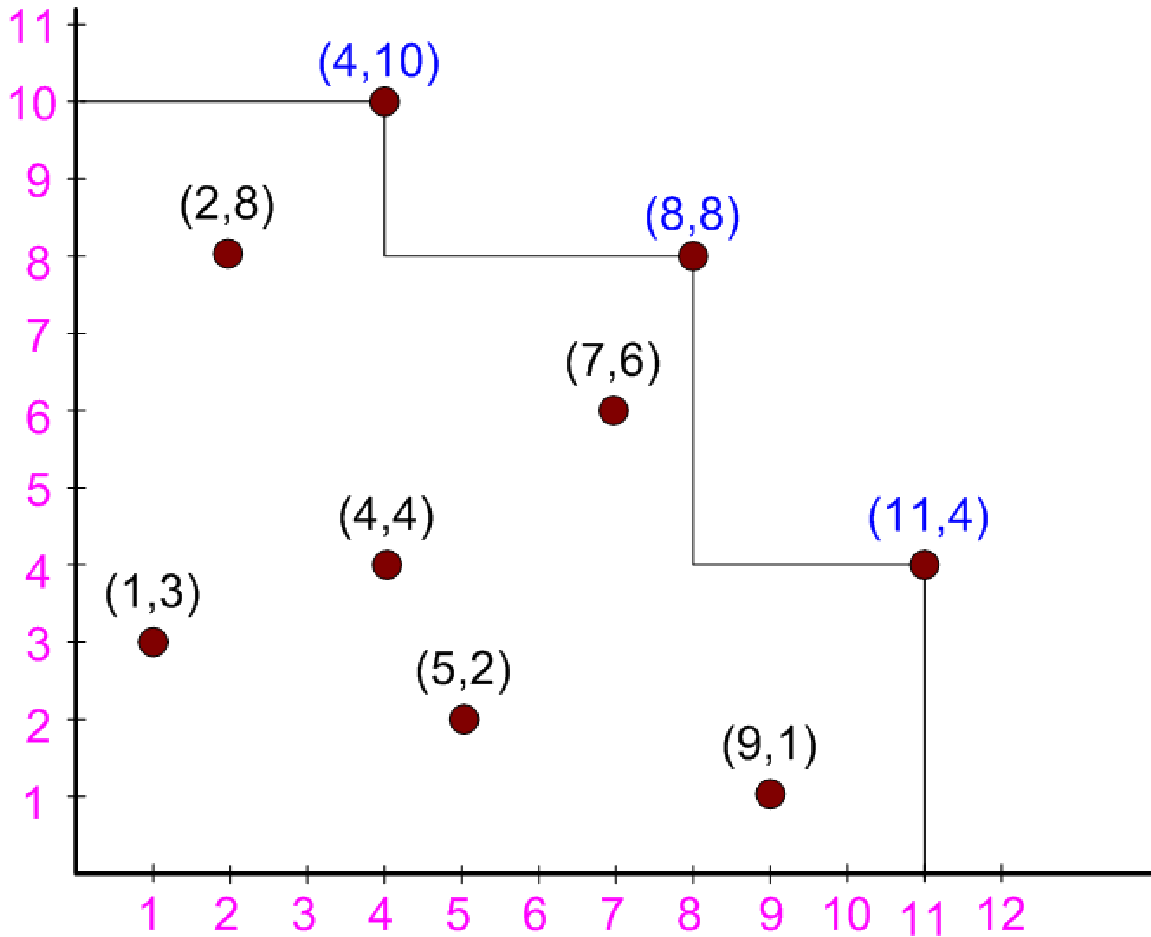
  A point p is said to be dominated by q if

  $p.x_i \le q.x_j \ \forall\ i = 1, \ldots, n$
- Maximal Point

  A point is said to be maximal if it is not dominated by any other point.

Example: Maximal Points in 2-Dimension

https://studyc.info/



Example: Buying a car

Suppose we want to buy a car which is fastest and cheapest. Fast cars are expensive. We want cheapest. We can't decide which one is more important, Speed or Price. Of course fast and cheap dominates slow and expensive car. So, given a collection of cars, we want the car which is not dominated by any other.

Formal Problem: Problem can be modeled as:

- For each car C, we define C (x, y) where

  x = speed of car and

  y = price of car
- This problem cannot be solved using maximal point algorithm.

Redefine Problem:

- For each car C', we define C' (x', y') where

  x' = speed of car and

  y' = negation of car price
- This problem is reduced to designing maximal point algorithm

https://studyc.info/

Problem Statement:

Given a set of m points, P = {p1, p2, . . . , pm}, in n- dimension. Our objective is to compute a set of maximal points i.e. set of points which are not dominated by anyone in the given list.

Mathematical Description:

Maximal Points = $\{ p \in P \mid \exists\, i \in \{1,. . . , n\}\ \&\ p.x_i \geq q.x_j,\ \forall, q \in \{p_1, p_2, . . . , p_m\}$

Brute Force Algorithm in n-dimension

```
MAXIMAL-POINTS (int m, Point P[1. . . m])
0     A = ∅;
1     for i ←1 to m    \\ m used for number of points
2     do maximal ← true
3       for j ← 1 to m
4       do
5               if (i ≠ j) &
6               for k ← 1 to n          \\ n stands for dimension
7               do
8                       P[i].x[k] ≤ P[j].x[k]
9                               then maximal ← false; break
10      if maximal
11              then A = A ∪ P[i]
```

Plane Sweep Algorithm in n-dimension

```
MAXIMAL-PINTS (int m, int n, Point P[1. . . m])
1     sort P in increasing order by first component
2     stack s;
3     for i ←1 to m    \\ m used for number of points
4     do
5       while (s.noEmpty() &
6       for j ← 2 to n  \\ n stands for dimension
7       do
8               s.top().x[j] ≤ P[i].x[j])
9       do s.pop();
10      s.push(P[i]);
11    output the contents of stack s;
```

https://studyc.info/

Lecture 13 Designing Algorithms using Brute Force and Divide & Conquer Approaches

The Closest Pair Problem

Given a set of n points, determine the two points that are closest to each other in terms of distance. Furthermore, if there are more than one pair of points with the closest distance, all such pairs should be identified.

Input: is a set of n points

Output: is a pair of points closest to each other. There can be more than one such pairs.

Distance
In mathematics, particular in geometry, distance on a given set M is a function d: M × M → R, where R denotes the set of real numbers that satisfies the following conditions:

1. $d(x, y) \geq 0$,
2. $d(x, y) = 0$ if and only if $x = y$.
3. Symmetric i.e.

   $d(x, y) = d(y, x)$.
4. Triangle inequality:

   $d(x, z) \leq d(x, y) + d(y, z)$.

Closest Pair Problem in 2-D

- A point in 2-D is an ordered pair of values (x, y).
- The Euclidean distance between two points

$P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ is

$$d(p_i, p_j) = sqr((x_i - x_j)^2 + (y_i - y_j)^2)$$

- The closest-pair problem is finding the two closest points in a set of n points.
- The brute force algorithm checks every pair of points.
- Assumption: We can avoid computing square roots by using squared distance.
  - This assumption will not lose correctness of the problem.

ClosestPairBF(P)

```
4.  mind ← ∞
5.  for i ← 1 to n − 1
6.  do
        4.  for j ← i + 1 to n
        5.  do
        6.  d ← ((xi − xj)^2 + (yi − yj)^2)
        7.  if d < minn then
                8.  mind ← d
                9.  mini ← i
                10. minj ← j
12. return mind, p(mini, minj)
```

Time Complexity

https://studyc.info/

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c$$

$$= \sum_{i=1}^{n-1} c(n-i)$$

$$= c(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i)$$

$$= cn(n-1) - c\frac{(n-1)n}{2}$$

$$= c(n^2 - n - \frac{n^2}{2} + \frac{n}{2})$$

$$= c(\frac{n^2}{2} - \frac{n}{2}) = \Theta(n^2)$$

Finding Closest Pair in 3-D

ClosestPairBF(P)

4. mind ← ∞
5. **for** i ← 1 to n − 1
6. **do**
    4. **for** j ← i + 1 to n
    5. **do**
    6. d ← $((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2)$
    7. **if** d < minn **then**
        8. mind ← d
        9. mini ← i
        10. minj ← j
12. **return** mind, p(mini), p(minj)

Time Complexity

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c$$

$$= \sum_{i=1}^{n-1} c(n-i)$$

$$= c(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i)$$

$$= \Theta(n^2)$$

Finding closest pair in n-D

ClosestPairBF(P)

1. mind ← ∞
2. **for** i ← 1 to n − 1
3. **do**
    4. **for** j ← i + 1 to n

https://studyc.info/

5. **do**
6. d ← $((x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + \ldots + (x_{in} - x_{jn})^2)$
7. **if** d < minn **then**
    8. mind ← d
    9. mini ← i
    10. minj ← j
11. **return** mind, p(mini), p(minj)

Finding Maximal in n-dimension

- Maximal Points in 2-D

  A point p is said to be dominated by q if

  p.x ≤ q.x and p.y ≤ q.y

  A point p is said to be maximal if

  p.x > q.x OR p.y > q.y
- Maximal Points in n-D

  A point p is said to be dominated by q if

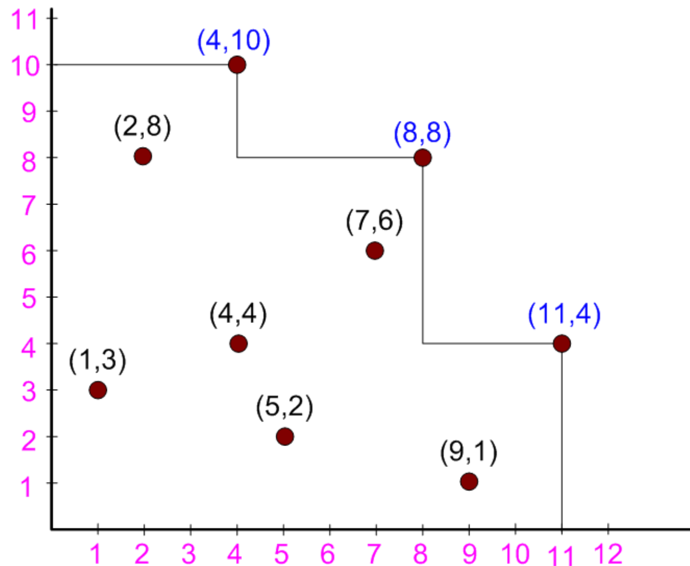
  $p.x_i \leq q.x_i \; \forall \; i = 1, \ldots, n$

  A point p is said to be maximal if

  $\exists \; i = 1, \ldots, n, \; p.x_i > q.x_i$

  A point is said to be maximal if it is not dominated by any other point.

Example: Maximal Points in 2-Dimension



Problem Statement:

Given a set of m points, P = {$p_1, p_2, \ldots, p_m$}, in n- dimension. Our objective is to compute a set of maximal points i.e. set of points which are not dominated by anyone in the given list.

https://studyc.info/

Mathematical Description:

Maximal Points = { $p \in P \mid \forall q \in \{p_1, p_2, \ldots, p_m\}, q \neq p, \exists i \in \{1, \ldots, n\}$ & $p.x_i \geq q.x_j$}

Brute Force Algorithm in n-dimension

```
MAXIMAL-POINTS (int n, Point P[1. . . m])
0     A = ∅;
1     for i ←1 to m    \\ m used for number of points
2     do maximal ← true
3       for j ← 1 to m
4       do
5               if (i ≠ j) &
6               for k ← 1 to n          \\ n stands for dimension
7               do
8                       P[i].x[k] ≤ P[j].x[k]
9                               then maximal ← false; break
10      if maximal
11              then A = A ∪ P[i]
```

https://studyc.info/

Lecture 14 Designing Algorithms using Divide & Conquer Approach

Divide and Conquer Approach

A general Divide and Conquer Algorithm

Step 1: If the problem size is small, solve this problem directly otherwise, split the original problem into 2 or more sub-problems with almost equal sizes.

Step 2: Recursively solve these sub-problems by applying this algorithm.

Step 3: Merge the solutions of the sub- problems into a solution of the original problem.

Time Complexity of General Algorithms

Time Complexity: $T(n) = \begin{cases} 2T(n/2) + S(n) + M(n), & n \geq c \\ b \quad , & n < c \end{cases}$

where S(n) is time for splitting

M(n) is time for merging

B and c are constants

Examples are *binary search*, *quick sort* and *merge sort*.

Merge Sort:

Merge-sort is based on divide-and-conquer approach and can be described by the following three steps:

Divide Step:

- If given array A has zero or one element, return S.
- Otherwise, divide A into two arrays, A1 and A2,
- Each containing about half of the elements of A.

Recursion Step:

- Recursively sort array A1, A2

Conquer Step:

- Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence.

Visualization of Merge-sort as Binary Tree

- We can visualize Merge-sort by means of binary tree where each node of the tree represents a recursive call
- Each external node represents individual elements of given array A.
- Such a tree is called Merge-sort tree.
- The heart of the Merge-sort algorithm is conquer step, which merge two sorted sequences into a single sorted sequence

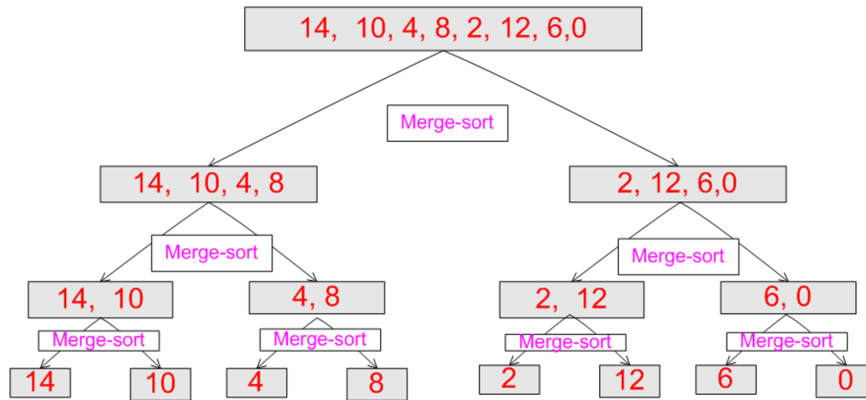
https://studyc.info/

Sorting Example: Divide and Conquer Rule

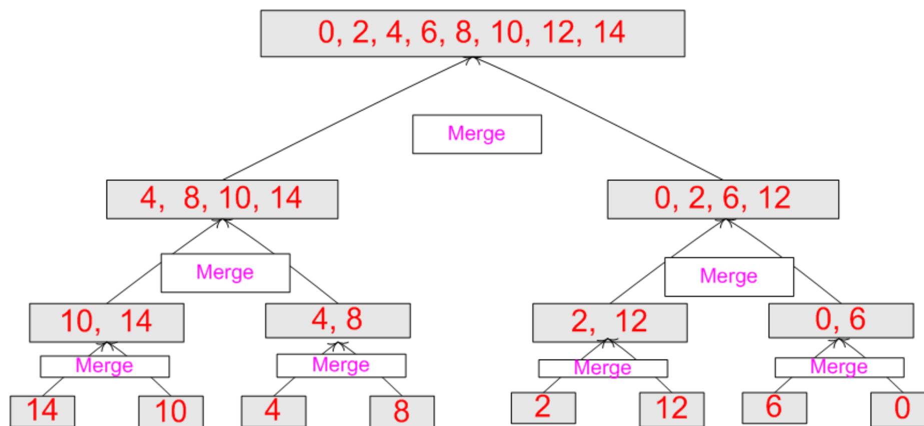
Sort the array [14, 10, 4, 8, 2, 12, 6, 0] in the ascending order

Solution:

Divide



Recursion and Conquer



Merge-sort Algorithm

Merge-sort(A, f, l)

1. **if** f < l
2. **then** m = (f + l)/2
3. Merge-sort(A, f, m)
4. Merge-sort(A, m + 1, l)
5. Merge(A, f, m, l)

Merge-sort Algorithm

Merge(A, f, m, l)

1. T[f..l] \\declare temporary array of same size
2. i ← f; k ← f; j ← m + 1 \\initialize integers i, j, and k
3. **while** (i ≤ m) and (j ≤ l)
4. **do if** (A[i] ≤ A[j]) \\comparison of elements

https://studyc.info/

5. **then** T[k++] ← A[i++]
6. **else** T[k++] ← A[j++]
7. **while** (i ≤ m)
8. **do** T[k++] ← A[i++] \\copy from A to T
9. **while** (j ≤ l)
10. **do** T[k++] ← A[j++] \\copy from A to T
11. **for** i ← p to r
12. **do** A[i] ← T[i] \\copy from T to A

Analysis of Merge-sort Algorithm

- Let $T(n)$ be the time taken by this algorithm to sort an array of n elements dividing A into sub-arrays A_1 and A_2.
- It is easy to see that the Merge (A_1, A_2, A) takes the linear time. Consequently,

$T(n) = T(n/2) + T(n/2) + \theta(n)$

$T(n) = 2T(n/2) + \theta(n)$

- The above recurrence relation is non-homogenous and can be solved by any of the methods
  - Defining characteristics polynomial
  - Substitution
  - recursion tree or
  - master method

Analysis: Substitution Method

$$T(n) = 2.T(\frac{n}{2}) + n$$

$$T(\frac{n}{2}) = 2.T(\frac{n}{2^2}) + \frac{n}{2}$$

$$T(\frac{n}{2^2}) = 2.T(\frac{n}{2^3}) + \frac{n}{2^2}$$

$$T(\frac{n}{2^3}) = 2.T(\frac{n}{2^4}) + \frac{n}{2^3} \quad . . .$$

$$T(\frac{n}{2^{k-1}}) = 2.T(\frac{n}{2^k}) + \frac{n}{2^{k-1}}$$

$$T(n) = 2.T(\frac{n}{2}) + \Theta(n) = 2^2.T(\frac{n}{2^2}) + n + n$$

$$T(n) = 2^2.T(\frac{n}{2^2}) + n + n$$

$$T(n) = 2^3.T(\frac{n}{2^3}) + n + n + n$$

$$. . .$$

$$T(n) = 2^k.T(\frac{n}{2^k}) + \underbrace{n + n + . . . + n}_{k-times}$$

$$T(n) = 2^k.T(\frac{n}{2^k}) + k.n$$

Let us suppose that: $n = 2^k \Rightarrow \log_2 n = k$

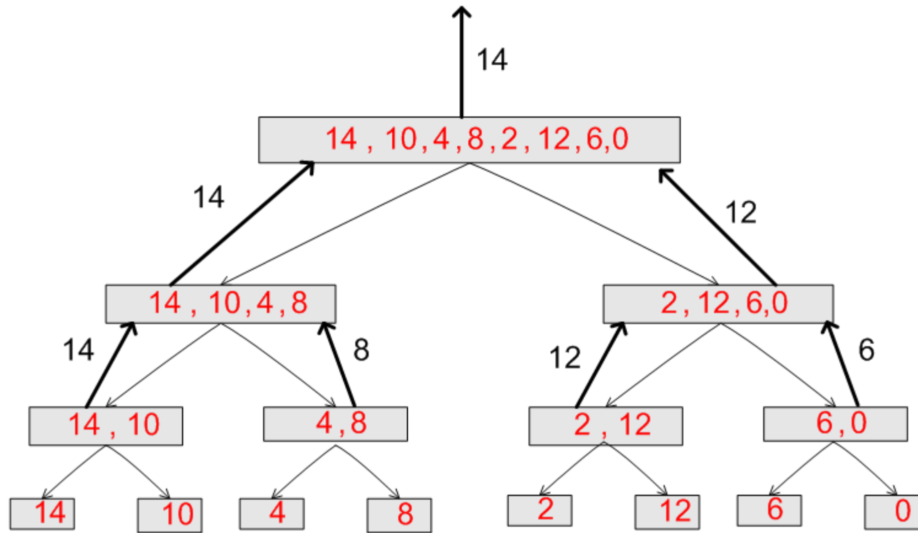
Hence, $T(n) = n.T(1) + n.\log_2 n = n + n.\log_2 n$

$T(n) = \Theta(n.\log_2 n)$

https://studyc.info/

Searching: Finding Maxima in 1-D

Finding the maximum of a set S of n numbers



Time Complexity

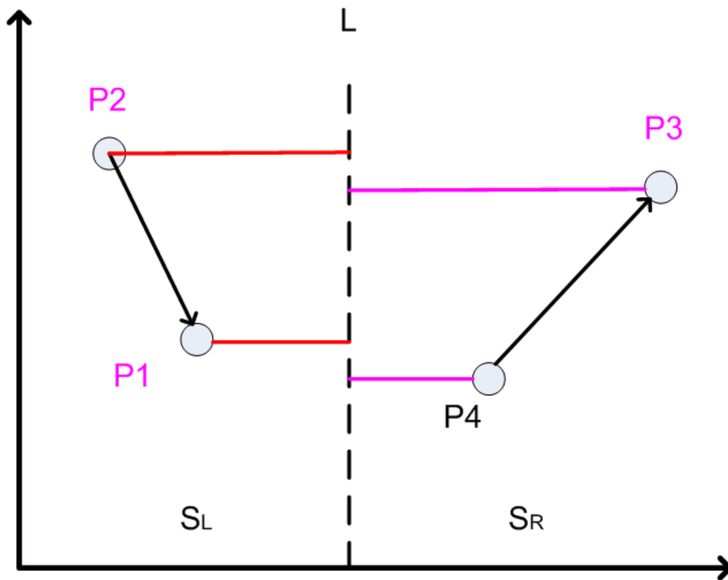
$$T(n) = \begin{cases} 2T(n/2)+1 & , \quad n > 2 \\ 1 & , \quad n \le 2 \end{cases}$$

Assume n = 2k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 1 = 2(2T(n/4) + 1) + 1 \\ &= 2^2T(n/2^2) + 2 + 1 \\ &= 2^2(2T(n/2^3) + 1) + 2 + 1 \\ &= 2^3T(n/2^3) + 2^2 + 2^1 + 1 \\ &\;\vdots \\ &= 2^{k-1}T(n/2^{k-1}) + 2^{k-2} + \ldots + 2^2 + 2^1 + 1 \\ &= 2^{k-1}T(2) + 2^{k-2} + \ldots + 2^2 + 2^1 + 1 \\ &= 2^{k-1} + 2^{k-2} + \ldots + 4 + 2 + 1 = 2k - 1 = n - 1 = \Theta(n) \end{aligned}$$

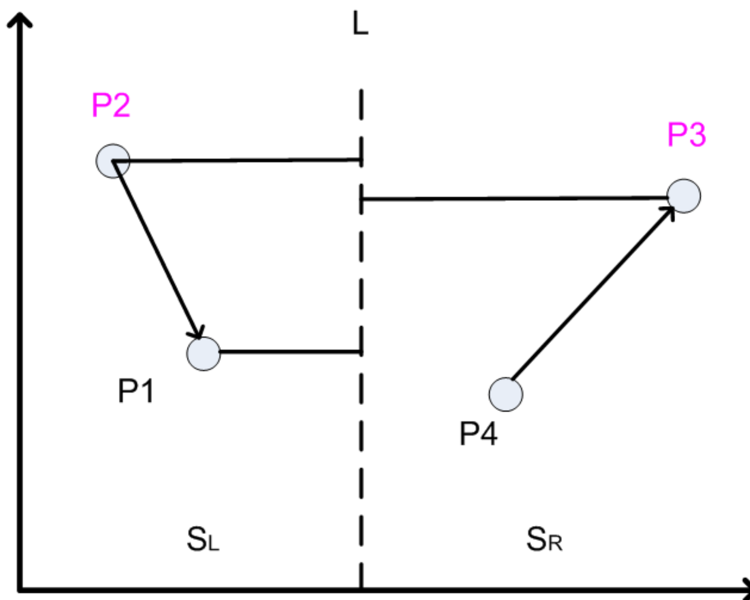
Finding Maxima in 2-D using Divide and Conquer

https://studyc.info/



{P_1, P_2} both maximal in S_L and {P_3} only maxima in S_R

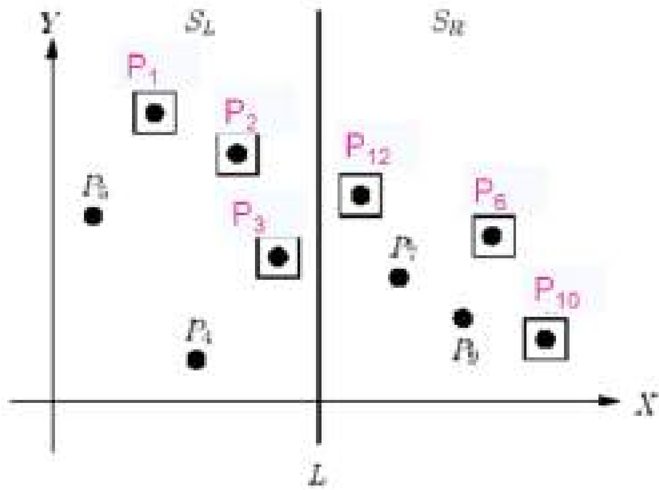
<u>Merging S_L and S_R</u>



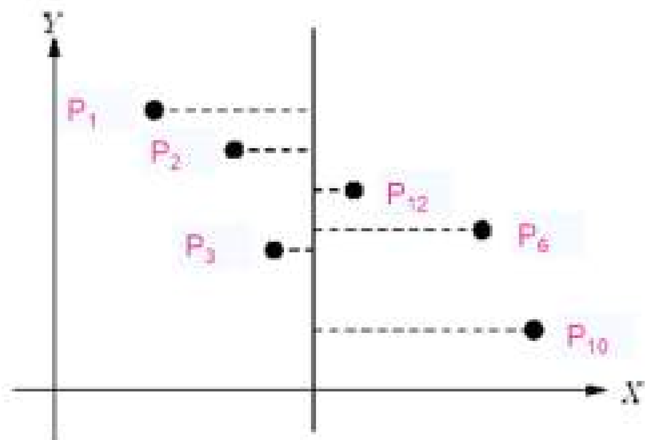
After Merging Maximal in S_L and S_R we get {P_2, P_3} only maximal

<u>Divide and Conquer for Maxima Finding Problem</u>

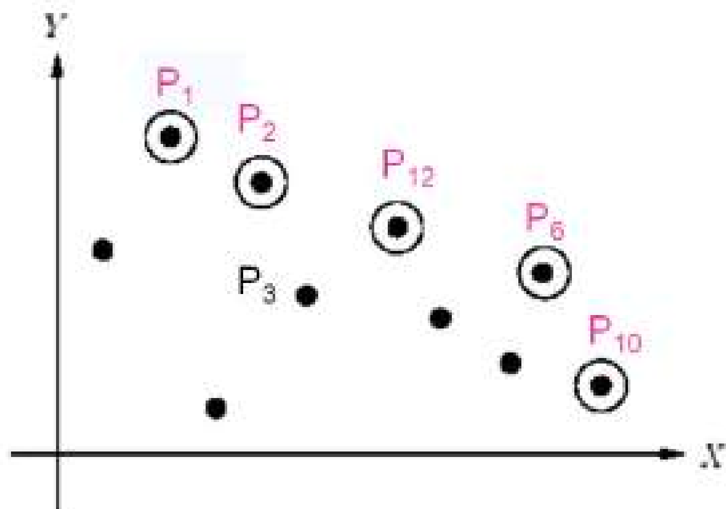
https://studyc.info/



The maximal points of S_L and S_R



P3 is not maximal point of S_L



Algorithm: Maxima Finding Problem

https://studyc.info/

Input: A set S of 2-dimensional points.

Output: The maximal set of S.

Maxima(P[1..n])

1. Sort the points in ascending order w. r .t. X axis
2. If |S| = 1, then return it, else

   find a line perpendicular to X-axis which separates S into S_L and S_R, each of which consisting of n/2 points.
3. Recursively find the maxima's S_L and S_R
4. Project the maxima's of S_L and S_R onto L and sort these points according to their y-values.
5. Conduct a linear scan on the projections and discard each of maxima of S_L if its y-value is less than the y-value of some maxima's of S_R.

Time Complexity

$$T(n) = \begin{cases} 2T(n/2) + O(n) + O(n) & , \quad n \geq 2 \\ 1 & , \quad n < 2 \end{cases}$$

Assume n = 2k, then

$$\begin{aligned} T(n) &= 2T(n/2) + n + n \\ &= 2(2T(n/4) + n/2 + n/2) + n + n \\ &= 2^2T(n/2^2) + n + n + n + n \\ &= 2^2T(n/2^2) + 4n \\ &= 2^2(2T(n/2^3) + n/4 + n/4) + 4n \\ &= 2^3T(n/2^3) + n + n + 6n \end{aligned}$$

$$T(n) = 2^3T(n/2^3) + n + n + 6n$$

.

.

$$\begin{aligned} T(n) &= 2^kT(n/2^k) + 2kn \\ &= 2^kT(2^k/2^k) + 2kn \quad \text{Since } n = 2^k \end{aligned}$$

Hence

$$T(n) = 2k + 2kn$$

$$T(n) = 2k + 2kn \qquad n = 2^k \Rightarrow k = \log(n)$$

$$T(n) = n + 2n.\log n = \Theta(n.\log n)$$

Necessary Dividing Problem into two Parts?

Maximal Points: Dividing Problem into four Parts

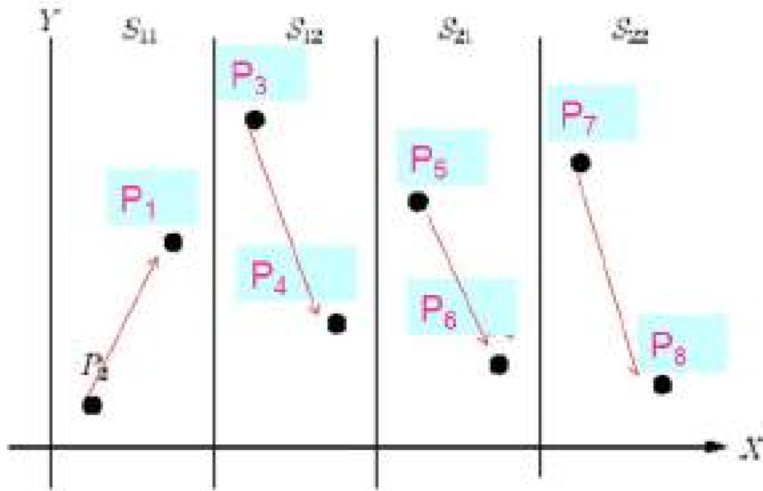
Maximal points in $S_{11} = \{P_1\}$

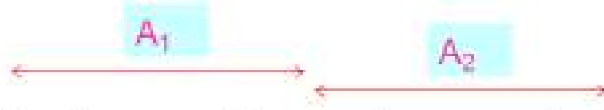
Maximal points in $S_{12} = \{P_3, P_4\}$

Maximal points in $S_{21} = \{P_5, P_6\}$

https://studyc.info/

Maximal points in $S_{22} = \{P_7, P_8\}$

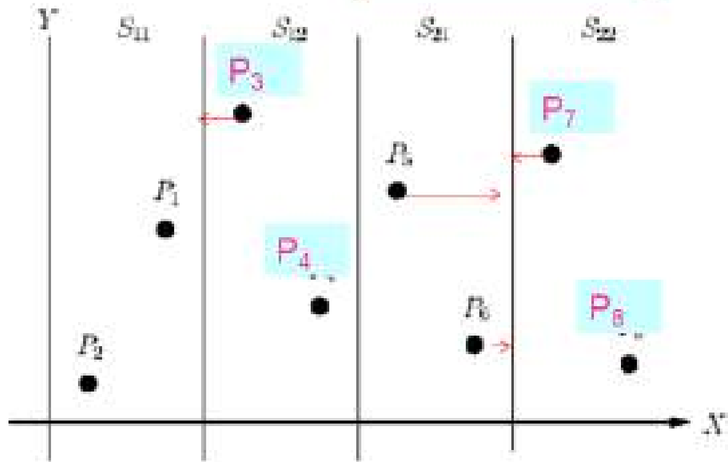




Merging S_{12}, S_{12}
$A_1 = \{P_3, P_4\}$

Merging S_{21}, S_{22}
$A_2 = \{P_7, P_8\}$

Merging A_1, A_2
$A = \{P_3, P_7, P_8\}$



https://studyc.info/

Merging S_{12}, S_{12}

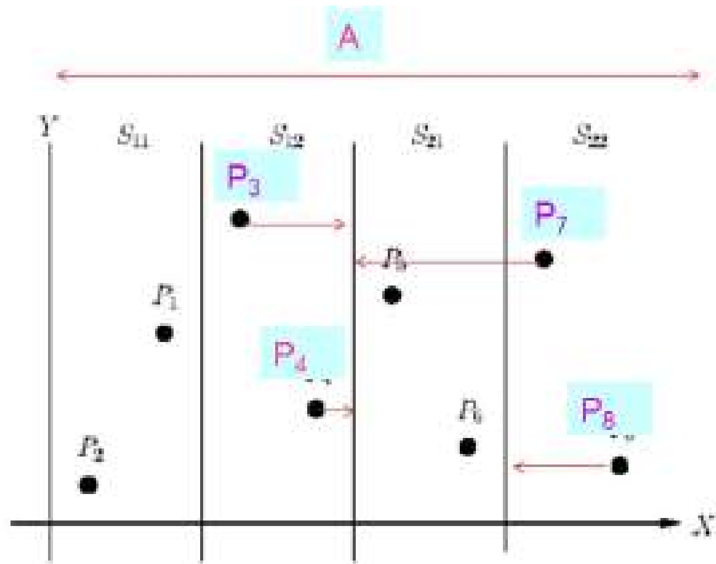
$A_1 = \{P_3, P_4\}$

Merging S_{21}, S_{22}

$A_2 = \{P_7, P_8\}$

Merging A_1, A_2

$A = \{P_3, P_7, P_8\}$



Closest pair problem in 2-D using Divide and Conquer

The closest pair problem is defined as follows:

- Given a set of n points
- Determine the two points that are closest to each other in terms of distance.
- Furthermore, if there is more than one pair of points with the closest distances, all such pairs should be identified.

First we sort the points on x-coordinate basis, and divide into left and right parts

$p_1\ p_2\ ...\ p_{n/2}$ and $p_{n/2+1}\ ...\ P_n$

Solve recursively the left and right sub-problems

How do we combine two solutions?

Let d = min $\{d_i, d_r\}$ where d is distance of closest pair where both points are either in left or in right something is missing. We have to check where one point is from left and the other from the right. Such closest-pair can only be in a strip of width 2d around the dividing line, otherwise the points would be more than d units apart.

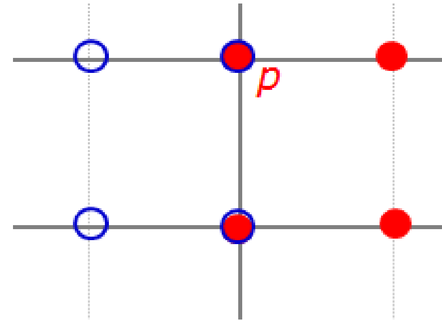
Combining Solutions

Finding the closest pair in a strip of width 2d, knowing that no one in any two given pairs is closer than d

For a given point p from one partition, where can there be a point q from the other partition, that can form the closest pair with p?

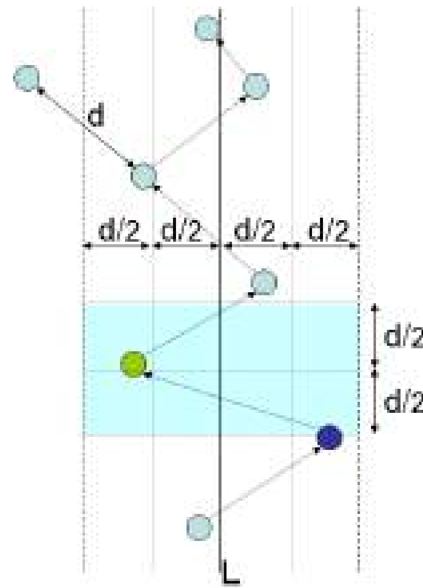
https://studyc.info/

- How many points can there be in this square?
  At most 4



Algorithm for checking the strip:

– Sort all the points in the strip on the y-coordinate
– For each point p only 7 points ahead of it in the order have to be checked to see if any of them is closer to p than d



1. Partition the strip into squares of length d/2 as shown in the picture.
2. Each square contains at most 1 point by definition of d.
3. If there are at least 2 squares between points then they cannot be the closest points.
4. There are at most 8 squares to check.

Closest Pair: Divide and Conquer Approach

Closest-Pair(P, l, r)
01 **if** r – l < 3 **then return ClosestPairBF(P)**
02 q ¬ é(l+r)/2ù
03 dl ¬ *Closest-Pair*(P, l, q-1)
04 dr ¬ *Closest-Pair*(P, q, r)
05 d ¬ min(dl, dr)
06 **for** i ¬ l **to** r **do**
07 **if** P[q].x - d £ P[i].x £ P[q].x + d **then**
08 append P[i] to S
09 Sort S on y-coordinate
10 **for** j ¬ 1 **to** size_of(S)-1 **do**
11 Check if any of d(S[j],S[j]+1), ..., d(S[j],S[j]+7) is smaller than d, if so set d to the smallest of them
12 **return** d

Running Time

Running time of a divide-and-conquer algorithm can be described by a recurrence

https://studyc.info/

- Divide = O(1)
- Combine = O(n lg n)
- This gives the recurrence given below
- Total running time: O(n $\log_2$ n)

$$T(n) = \begin{cases} n & n \le 3 \\ 2T(\frac{n}{2}) + n\log n & \text{otherwise} \end{cases}$$

Improved Version: Divide and Conquer Approach

- Sort all the points by x and y coordinate once
- Before recursive calls, partition the sorted lists into two sorted sublists for the left and right halves, it will take simple time $O(n)$
- When combining, run through the y-sorted list once and select all points that are in a $2d$ strip around partition line, again time $O(n)$
- New recurrence:

$$T(n) = \begin{cases} n & n \le 3 \\ 2T(\frac{n}{2}) + n & \text{otherwise} \end{cases}$$

https://studyc.info/

Lecture 15 Dynamic Programming for Solving Optimization Problems

Optimization Problems

If a problem has only one correct solution, then optimization is not required. For example, there is only one sorted sequence containing a given set of numbers. Optimization problems have many solutions. We want to compute an optimal solution e. g. with minimal cost and maximal gain. There could be many solutions having optimal value. Dynamic programming is very effective technique. Development of dynamic programming algorithms can be broken into a sequence steps as in the next.

Steps in Development of Dynamic Algorithms

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed information

Note: Steps 1-3 form the basis of a dynamic programming solution to a problem. Step 4 can be omitted only if the value of an optimal solution is required.

Why Dynamic Programming?

- Dynamic programming, like divide and conquer method, solves problems by combining the solutions to sub-problems.
- Divide and conquer algorithms:
  - partition the problem into independent sub-problem
  - Solve the sub-problem recursively and
  - Combine their solutions to solve the original problem
- In contrast, dynamic programming is applicable when the sub-problems are not independent.
- Dynamic programming is typically applied to optimization problems.

Time Complexity in Dynamic Algorithms

If there are polynomial number of sub-problems and each sub-problem can be computed in polynomial time then the solution of whole problem can be found in polynomial time.

Remark: Greedy also applies a top-down strategy but usually on one sub-problem so that the order of computation is clear.

Catalan Numbers

Objective: Find C(n), the number of ways to compute product $x_1 . x_2 .... x_n$.

n	multiplication order
2	(x1 • x2)

https://studyc.info/

3	(x1 • (x2 • x3))
	((x1 • x2) • x3)
4	(x1 • (x2 • (x3 • x4)))
	(x1 • ((x2 • x3) • x4))
	((x1 • x2) • (x3 • x4))
	((x1 • (x2 • x3)) • x4)
	(((x1 • x2) • x3) • x4)

Multiplying n Numbers – small n

n	C(n)
1	1
2	1
3	2
4	5
5	14
6	42
7	132

Recursive Equation:

where is the last multiplication?

$$C(n) = \sum_{k=1}^{n-1} C(k) \cdot C(n-k)$$

Catalan Numbers: $C(n) = \frac{1}{n}\binom{2n-2}{n-1}$.

Asymptotic value: $C(n) \approx \frac{4^n}{n^{3/2}}$

$$\frac{C(n)}{C(n-1)} \to 4 \quad \text{for } n \to \infty$$

https://studyc.info/

Chain-Matrix Multiplication

Given a chain of [A_1, A_2, . . . , A_n] of n matrices where for i = 1, 2, . . . , n, matrix A_i has dimension p_{i-1} x p_i, find the order of multiplication which minimizes the number of scalar multiplications.

Note:

Order of A_1 is p_0 x p_1,

Order of A_2 is p_1 x p_2,

Order of A_3 is p_2 x p_3, etc.

Order of A_1 x A_2 x A_3 is p_0 x p_3,

Order of A_1 x A_2 x . . . x A_n is p_0 x p_n

Objective is to find order not multiplication

- Given a sequence of matrices, we want to find a most efficient way to multiply these matrices
- It means that problem is not actually to perform the multiplications, but decide the order in which these must be multiplied to reduce the cost.
- This problem is an optimization type which can be solved using dynamic programming.
- The problem is not limited to find an efficient way of multiplication of matrices, but can be used to be applied in various purposes.
- But how to transform the original problem into chain matrix multiplication, this is another issue, which is common in systems modeling.

Why this problem is of Optimization Category?

- If these matrices are all square and of same size, the multiplication order will not affect the total cost.
- If matrices are of different sizes but compatible for multiplication, then order can make big difference.
- Brute Force approach
  - The number of possible multiplication orders are exponential in n, and so trying all possible orders may take a very long time.
- Dynamic Programming
  - To find an optimal solution, we will discuss it using dynamic programming to solve it efficiently.

Assumptions (Only Multiplications Considered)

We really want is the minimum cost to multiply. But we know that cost of an algorithm depends on how many number of operations are performed i.e. we must be interested to minimize number of operations, needed to multiply out the matrices. As in matrices multiplication, there will be addition as well multiplication operations in addition to other. Since cost of multiplication is dominated over addition therefore we will minimize the number of multiplication operations in this problem. In case of two matrices, there is only one way to multiply them, so the cost fixed.

https://studyc.info/

Brute Force Approach

- If we wish to multiply two matrices: $A = a[i, j]_{p, q}$ and $B = b[i, j]_{q, r}$
- Now if C = AB then order of C is p x r.
- Since in each entry c[i, j], there are q number of scalar of multiplications
- Total number of scalar multiplications in computing C = Total entries in C x Cost of computing a single entry = p . r . q
- Hence the computational cost of AB = p . q . r will be $C[i, j] = \sum_{k=1}^{q} A[i, k] B[k, j]$

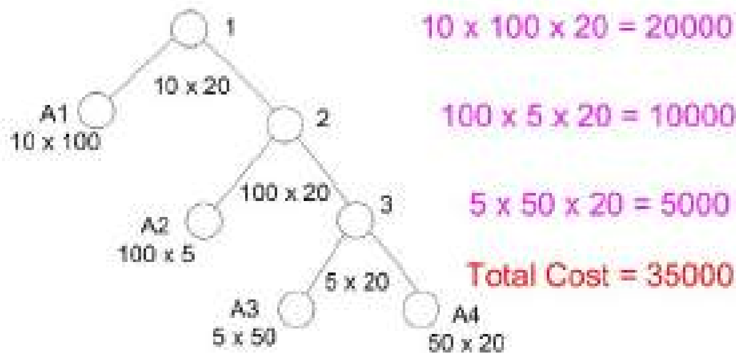
Example

- Given a sequence [A_1, A_2, A_3, A_4]
- Order of A_1 = 10 x 100
- Order of A_2 = 100 x 5
- Order of A_3 = 5x 50
- Order of A_4 = 50x 20

Compute the order of the product $A_1.A_2.A_3.A_4$ in such a way that minimizes the total number of scalar multiplications.

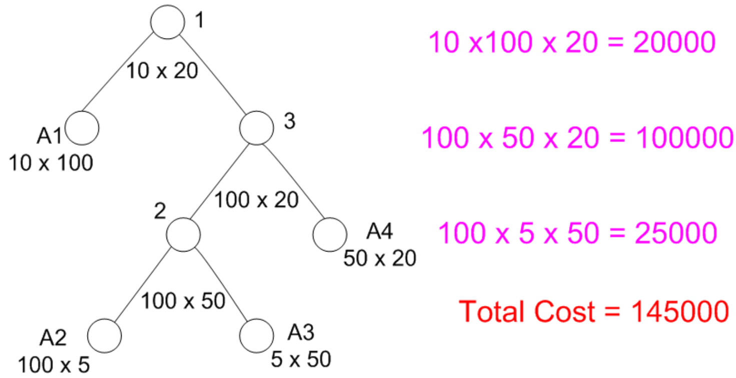
- There are five ways to parenthesize this product
- Cost of computing the matrix product may vary, depending on order of parenthesis.
- All possible ways of parenthesizing
  $(A_1 \cdot (A_2 . (A_3 . A_4)))$
  $(A_1 \cdot ((A_2 . A_3) . A_4))$
  $((A_1 \cdot A_2) . (A_3 . A_4))$
  $((A_1 \cdot (A_2 . A_3)) . A_4)$
  $(((A_1 \cdot A_2) . A_3) . A_4)$

Kinds of problems solved by algorithms

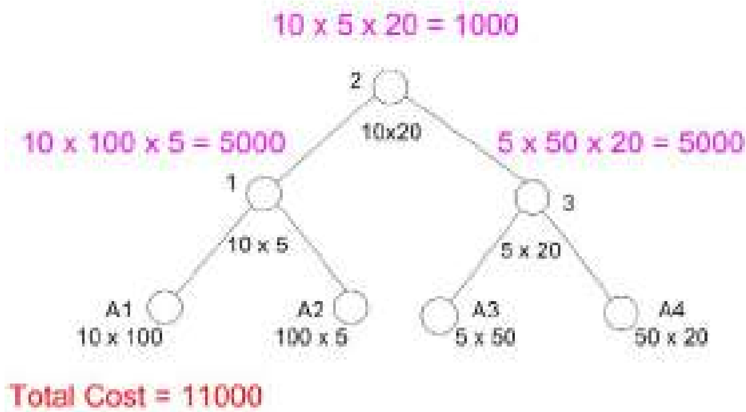


First Chain: $(A_1 \cdot (A_2 . (A_3 . A_4)))$

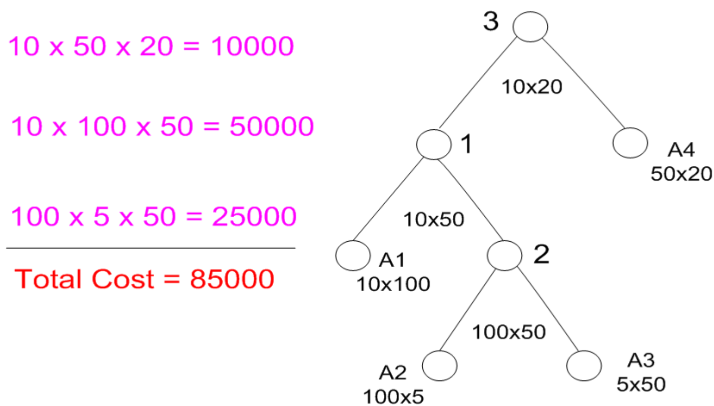
https://studyc.info/



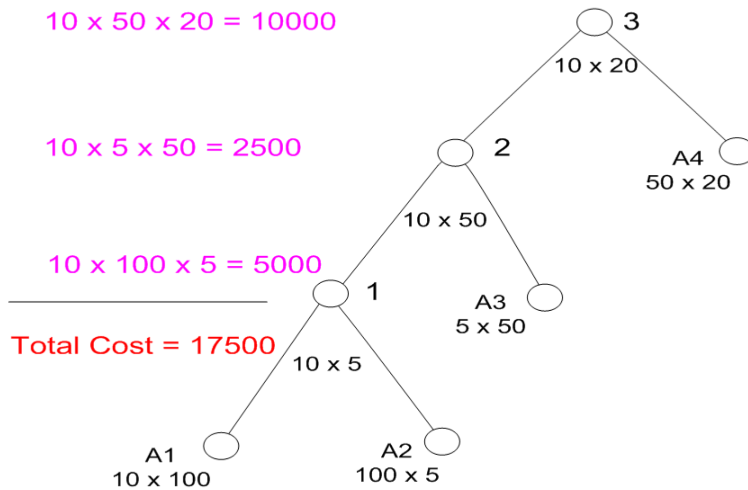
Second Chain: $(A_1 \cdot ((A_2 . A_3). A_4))$



Third Chain: $((A_1 \cdot A_2). (A_3. A_4))$



Fourth Chain: $((A_1 \cdot (A_2. A_3)). A_4)$



Fifth Chain: $(((A_1 \cdot A_2). A_3). A_4)$

Chain Matrix Cost

First Chain	35,000
Second Chain	145,000
Third Chain	**11,000**
Fourth Chain	85,000
Fifth Chain	17,500

Generalization of Brute Force Approach

If there is sequence of n matrices, $[A_1, A_2, \ldots, A_n]$. A_i has dimension $p_{i-1} \times p_i$, where for i = 1, 2, . . . , n. Find order of multiplication that minimizes number of scalar multiplications using brute force approach

Recurrence Relation: After k^{th} matrix, create two sub-lists, one with k and other with n - k matrices i.e. $(A_1 A_2 A_3 A_4 A_5 \ldots A_k)$ $(A_{k+1} A_{k+2} \ldots A_n)$
Let P(n) be the number of different ways of parenthesizing n items

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 \end{cases}$$

If n = 2

P(2) = P(1).P(1) = 1.1 = 1

If n = 3

P(3) = P(1).P(2) + P(2).P(1) = 1.1 + 1.1 = 2

$(A_1 A_2 A_3) = ((A_1 . A_2). A_3)$ OR $(A_1 . (A_2. A_3))$

If n = 4

P(4) = P(1).P(3) + P(2).P(2) + P(3).P(1) = 1.2 + 1.1 + 2.1 = 5

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

Why Brute Force Approach is not Economical?

This is related to a famous function in combinatorics called the Catalan numbers. Catalan numbers are related with the number of different binary trees on n nodes.

$P(n) \in (4^n/n^{3/2})$

The dominating term is the exponential 4^n thus P(n) will grow large very quickly and hence this approach is not economical.

https://studyc.info/

Lecture 16 Chain Matrix Multiplication Problem using Dynamic Programming

Problem Statement: Chain Matrix Multiplication

Given a chain of [A_1, A_2, . . . , A_n] of n matrices where for i = 1, 2, . . . , n, matrix A_i has dimension p_{i-1} x p_i, find the order of multiplication which minimizes the number of scalar multiplications.

Note:

Order of A_1 is p_0 x p_1,
Order of A_2 is p_1 x p_2,
Order of A_3 is p_2 x p_3, etc.
Order of A_1 x A_2 x A_3 is p_0 x p_3,
Order of A_1 x A_2 x . . . x A_n is p_0 x p_n

Why Dynamic Programming in this problem?

- Problem is of type optimization
- Sub-problems are dependent
- Optimal structure can be characterized and
- Can be defined recursively
- Solution for base cases exits
- Optimal solution can be constructed
- Hence here is dynamic programming

Dynamic Programming Formulation

- Let $A_{i..j} = A_i \cdot A_{i+1} \ldots A_j$
- Order of $A_i = p_{i-1}$ x p_i and order of $A_j = p_{j-1}$ x p_j
- Order of $A_{i..j}$ = rows in A_i x columns in $A_j = p_{i-1} \times p_j$
- At the highest level of parenthesisation,
  $A_{i..j} = A_{i..k} \times A_{k+1..j}$ $\quad i \le k < j$
- Let m[i, j] = minimum number of multiplications needed to compute $A_{i..j}$, for $1 \le i \le j \le n$
- Objective function = finding minimum number of multiplications needed to compute $A_{1..n}$ i.e. to compute m[1, n]
- $A_{i..j} = (A_i. A_{i+1} \ldots . A_k). (A_{k+1}. A_{k+2} \ldots . A_j) = A_{i..k} \times A_{k+1..j}$ $\quad i \le k < j$
- Order of $A_{i..k} = p_{i-1}$ x p_k, and order of $A_{k+1..j} = p_k$ x p_j,
- m[i, k] = minimum number of multiplications needed to compute $A_{i..k}$
- m[k+1, j] = minimum number of multiplications needed to compute $A_{k+1..j}$

Mathematical Model

$$m[i,j] = 0$$

$$m[i,j] = \min_{i \le k < j}\left(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\right)$$

https://studyc.info/

Example: Compute optimal multiplication order for a series of matrices given below

$$\underset{10\times100}{A_1}.\underset{100\times5}{A_2}.\underset{5\times50}{A_3}.\underset{50\times20}{A_4}$$

m[1,1]	m[1,2]	m[1,3]	m[1,4]
	m[2,2]	m[2,3]	m[2,4]
		m[3,3]	m[3,4]
			m[4,4]

$P_0 = 10$
$P_1 = 100$
$P_2 = 5$
$P_3 = 50$
$P_4 = 20$

$$m[i,i] = 0, \forall i = 1,...,4$$

$$m[i,j] = \min_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j)$$

Main Diagonal:

m[1, 1] = 0 , m[2, 2] = 0
m[3, 3] = 0 , m[4, 4] = 0

Computing m[1, 2]

$$m[i,j] = \min_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j)$$

$$m[1,2] = \min_{1 \le k < 2}(m[1,k] + m[k+1,2] + p_0.p_k.p_2)$$

$$m[1,2] = \min(m[1,1] + m[2,2] + p_0.p_1.p_2)$$

$$m[1, 2] = 0 + 0 + 10 . 100 . 5 = 5000$$

$$s[1, 2] = k = 1$$

Computing m[2, 3]

$$m[i,j] = \min_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j)$$

$$m[2,3] = \min_{2 \le k < 3}(m[2,k] + m[k+1,3] + p_1.p_k.p_3)$$

$$m[2,3] = \min(m[2,2] + m[3,3] + p_1.p_2.p3)$$

$$m[2, 3] = 0 + 0 + 100 . 5 . 50 = 25000$$

$$s[2, 3] = k = 2$$

Computing m[3, 4]

https://studyc.info/

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j)$$

$$m[3,4] = \min_{3 \le k < 4} (m[3,k] + m[k+1,4] + p_2.p_k.p_4)$$

$$m[3,4] = \min (m[3,3] + m[4,4] + p_2.p_3.p_4)$$

$$m[3, 4] = 0 + 0 + 5 . 50 . 20 = 5000$$

$$s[3, 4] = k = 3$$

Computing m[1, 3]

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j)$$

$$m[1,3] = \min_{1 \le k < 3} (m[1,k] + m[k+1,3] + p_0.p_k.p_3)$$

$$m[1,3] = \min (m[1,1] + m[2,3] + p_0.p_1.p_3,$$

$$m[1,2] + m[3,3] + p_0.p_2.p_3))$$

$$m[1, 3] = \min\left(0 + 25000 + 10.100.50,\ 5000 + 0 + 10.5.50\right)$$

$$= \min\left(75000,\ 2500\right) = 2500$$

$$s[1, 3] = k = 2$$

Computing [2,4]

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j)$$

$$m[2,4] = \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1.p_k.p_4)$$

$$m[2,4] = \min (m[2,2] + m[3,4] + p_1.p_2.p_4,$$

$$m[2,3] + m[4,4] + p_1.p_3.p_4))$$

$$m[2, 4] = \min\left(0 + 5000 + 100.5.20,\ 25000 + 0 + 100.50.20\right)$$

$$= \min\left(15000,\ 35000\right) = 15000$$

$$s[2, 4] = k = 2$$

Computing m[1,4]

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}.p_k.p_j)$$

$$m[1,4] = \min_{1 \le k < 4} (m[1,k] + m[k+1,4] + p_0.p_k.p_4)$$

$$m[1,4] = \min (m[1,1] + m[2,4] + p_0.p_1.p_4,$$

$$m[1,2] + m[3,4] + p_0.p_2.p_4, m[1,3] + m[4,4] + p_0.p_3.p_4)$$

https://studyc.info/

$$m[1, 4] = \min(0 + 15000 + 10.100.20,\ 5000 + 5000 + 10.5.20,\ 2500 + 0 + 10.50.20)$$
$$= \min(35000,\ 11000,\ 35000) = 11000$$
$$s[1, 4] = k = 2$$

Final Cost Matrix and Its Order of Computation

Final cost matrix

0	5000	2500	11000
	0	25000	15000
		0	5000
			0

Order of Computation

1	5	8	10
	2	6	9
		3	7
			4

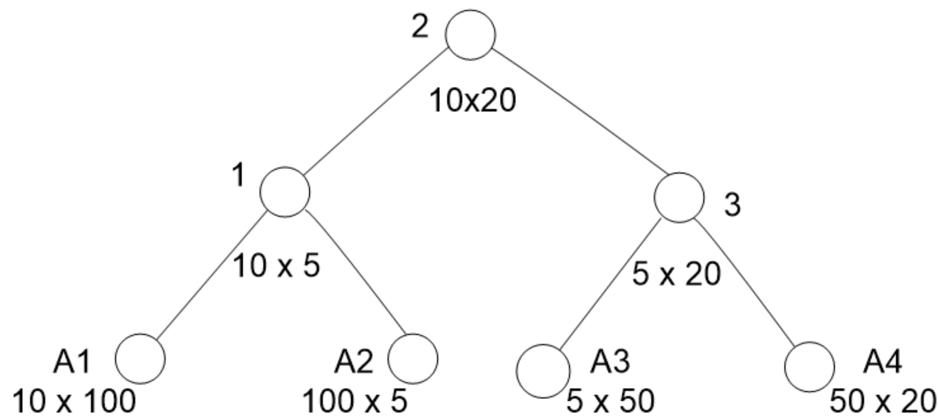
k's Values Leading Minimum m[i, j]

0	1	2	2
	0	2	2
		0	3
			0

Representing Order using Binary Tree

- The above computation shows that the minimum cost for multiplying those four matrices is 11000.
- The optimal order for multiplication is $((A_1 . A_2) . (A_3 . A_4))$
  for, m(1, 4)

  k = 2



Chain-Matrix-Order(p)

```
1.  n ← length[p] – 1
2.  for i ← 1 to n
3.      do m[i, i] ← 0
4.  for l  ← 2 to n,
5.      do for i ← 1 to n-l+1
6.          do j ← i+l-1
7.              m[i, j] ← ∞
8.              for k ← i to j-1
9.                  do q ← m[i, k] + m[k+1, j] + p_{i-1} . p_k . p_j
10.                     if q < m[i, j]
11.                         then m[i, j] = q
12.                             s[i, j] ← k
13. return m and s,            "l is chain length"
```

Computational Cost

$$T(n) = n + \sum_{i=1}^{n} \sum_{j=i+1}^{n} (j-i) = \sum_{i=1}^{n} \sum_{k=1}^{n-i} k$$

$$T(n) = n + \sum_{i=1}^{n} \frac{(n-i)(n-i+1)}{2}$$

$$T(n) = n + \frac{1}{2} \sum_{i=1}^{n} (n^2 - 2ni + i^2 + n - i)$$

$$T(n) = n + \frac{1}{2} (\sum_{i=1}^{n} n^2 - \sum_{i=1}^{n} 2ni + \sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} n - \sum_{i=1}^{n} i)$$

$$T(n) = n + \frac{1}{2} (n^2 \sum_{i=1}^{n} 1 - 2n \sum_{i=1}^{n} i + \sum_{i=1}^{n} i^2 + n \sum_{i=1}^{n} 1 - \sum_{i=1}^{n} i)$$

$$T(n) = n + \frac{1}{2} (n^2.n - 2n.\frac{n(n+1)}{2} + \frac{n(n+1)(2n+1)}{6} + n.n - \frac{n(n+1)}{2})$$

$$T(n) = n + \frac{1}{2} (n^3 - n^2(n+1) + \frac{n(n+1)(2n+1)}{6} + n^2 - \frac{n(n+1)}{2})$$

$$T(n) = n + \frac{1}{12} (6n^3 - 6n^3 - 6n^2 + 2n^3 + 3n^2 + n + 6n^2 - 3n^2 - 3n)$$

$$T(n) = \frac{1}{12} (12n + 2n^3 - 2n) = \frac{1}{12} (10n + 2n^3) = \frac{1}{6} (5n + n^3)$$

Cost Comparison Brute Force Dynamic Programming

A simple inspection of the nested loop structure yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index (l , i, and k) takes on at most n-1 values.

Brute Force Approach: $P(n) = C(n - 1) C(n) \in (4^n/n^{3/2})$

Generalization: Sequence of Objects

Although this algorithm applies well to the problem of matrix chain multiplication. Many researchers have noted that it generalizes well to solving a more abstract problem

- given a linear sequence of objects
- an associative binary operation on those objects hold
- the objective to find a way to compute the cost of performing that operation on any two given objects
- and finally computing the minimum cost for grouping these objects to apply the operation over the entire sequence.

It is obvious that this problem can be solved using chain matrix multiplication, because there is a one to one correspondence between both problems.

Generalization: String Concatenation

One common special case of chain matrix multiplication problem is string concatenation.

- For example, we are given a list of strings.
    - The cost of concatenating two strings of length m and n is for example O(m + n)
    - Since we need O(m) time to find the end of the first string and O(n) time to copy the second string onto the end of it.
    - Using this cost function, we can write a dynamic programming algorithm to find the fastest way to concatenate a sequence of strings
    - It is possible to concatenate all in time proportional to sum of their lengths, but here we are interested to link this problem with chain matrix multiplication.

Generalization: Parallel Processors

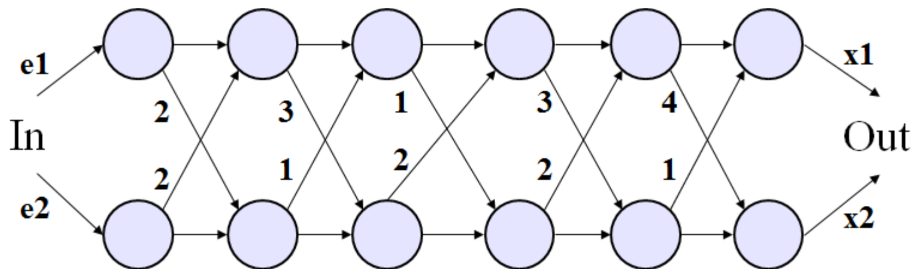
- Another generalization is to solve the problem when many parallel processors are available.
- In this case, instead of adding the costs of computing each subsequence, we just take the maximum, because we can do them both simultaneously.
- This can drastically affect both the minimum cost and the final optimal grouping
- But of course more balanced groupings that keep all the processors busy is more favorable solution
- There exists some more sophisticated approaches to solve this problem

https://studyc.info/

Lecture 17 Assembly-Line Scheduling Problem

- There are two assembly lines each with n stations
- The jth station on line i is denoted by $S_{i,j}$
- The assembly time at that station is $a_{i,j}$.
- An auto enters factory, goes into line i taking time e_i
- After going through the jth station on a line i, the auto goes on to the (j+1)st station on either line
- There is no transfer cost if it stays on the same line
- It takes time $t_{i,j}$ to transfer to other line after station $S_{i,j}$
- After exiting the nth station on a line, it takes time x_i for the completed auto to exit the factory.
- Problem is to determine which stations to choose from lines 1 and 2 to minimize total time through the factory.

Notations: Assembly-Line Scheduling Problem



Stations $S_{i,j}$:
2 assembly lines, i = 1,2
n stations, j = 1,...,n.
$a_{i,j}$ = assembly time at $S_{i,j}$
$t_{i,j}$ = transfer time from $S_{i,j}$ (to $S_{i-1,j+1}$ OR $S_{i+1,j+1}$)
e_i = entry time from line i
x_i = exit time from line i

Brute Force Solution

Total Computational Time = possible ways to enter in stations at level n x one way Cost
Possible ways to enter in stations at level 1 = 2^1
Possible ways to enter in stations at level 2 = $2^2 \cdots$
Possible ways to enter in stations at level 2 = 2^n
Total Computational Time = $n.2^n$

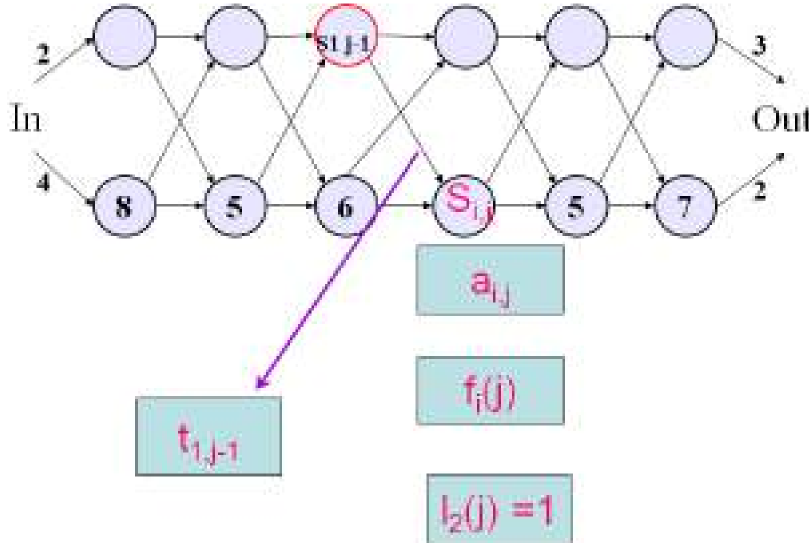
Dynamic Programming Solution

- Let $f_i[j]$ = fastest time from starting point station $S_{i,j}$
- $f_1[n]$ = fastest time from starting point station $S_{1\,n}$
- $f_2[n]$ = fastest time from starting point station $S_{2\,n}$

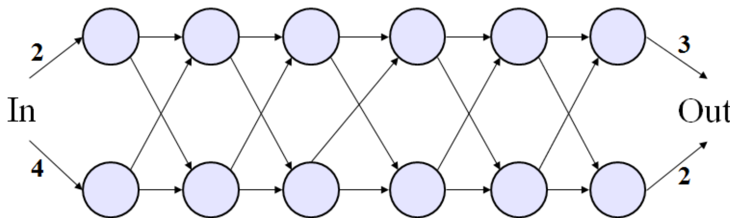
https://studyc.info/

- $l_i[j]$ = The line number, 1 or 2, whose station j-1 is used in a fastest way through station $S_{i,j}$.
- It is to be noted that $l_i[1]$ is not required to be defined because there is no station before 1
- $t_i[j-1]$ = transfer time from line i to station $S_{i-1,j}$ or $S_{i+1,j}$
- Objective function = $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$
- l^* = to be the line no. whose n^{th} station is used in a fastest way.

Notations: Finding Objective Function



Mathematical Model: Finding Objective Function



$f_1[1] = e_1 + a_{1,1}$;

$f_2[1] = e_2 + a_{2,1}$.

$f_1[j] = \min (f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$ for $j \geq 2$;

$f_2[j] = \min (f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})$ for $j \geq 2$;

Complete Model: Finding Objective Function

Base Cases

- $f_1[1] = e_1 + a_{1,1}$
- $f_2[1] = e_2 + a_{2,1}$

Two possible ways of computing $f_1[j]$

- $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$ OR $f_1[j] = f_1[j-1] + a_{1,j}$

  For j = 2, 3, . . ., n

  $f_1[j] = \min (f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$

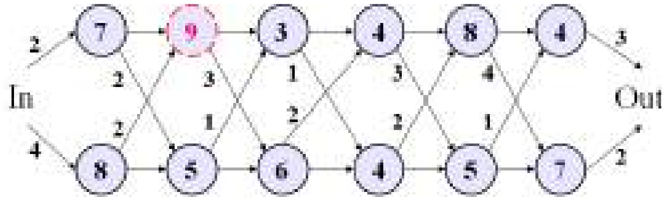
Symmetrically

For $j = 2, 3, \ldots, n$

$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

Objective function $= f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$

Computation of $f_1[2]$



$f_1[1] = e_1 + a_{1,1} = 2 + 7 = 9$

$f_2[1] = e_2 + a_{2,1} = 4 + 8 = 12$

$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$

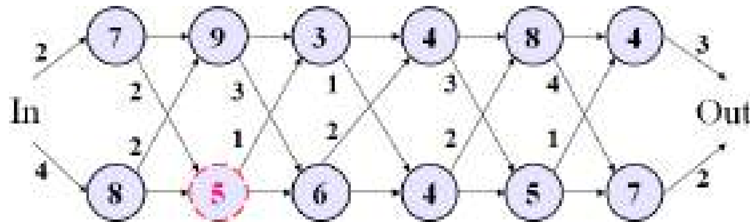
$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

$j = 2$

$f_1[2] = \min(f_1[1] + a_{1,2},\ f_2[1] + t_{2,1} + a_{1,2})$

$= \min(9 + 9,\ 12 + 2 + 9) = \min(18, 23) = 18,\ l_1[2] = 1$

Computation of $f_2[2]$



$f_1[1] = e_1 + a_{1,1} = 2 + 7 = 9$

$f_2[1] = e_2 + a_{2,1} = 4 + 8 = 12$

$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$

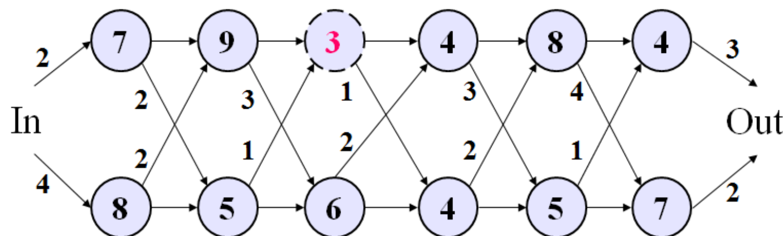
$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

$j = 2$

$f_2[2] = \min(f_2[1] + a_{2,2},\ f_1[1] + t_{1,1} + a_{2,2})$

$= \min(12 + 5,\ 9 + 2 + 5) = \min(17, 16) = 16,\quad l_2[2] = 1$

Computation of $f_1[3]$



https://studyc.info/

$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,\ j-1} + a_{1,j})$

$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

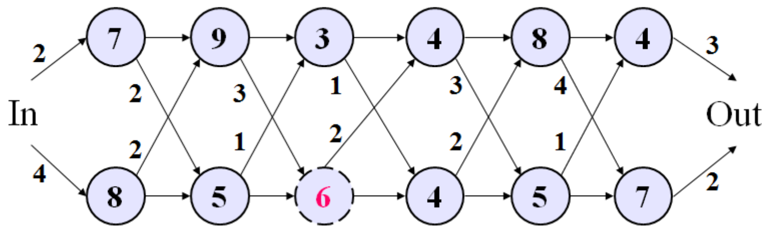
$j = 3$

$f_1[3] = \min(f_1[2] + a_{1,3},\ f_2[2] + t_{2,\ 2} + a_{1,3})$

$= \min(18 + 3,\ 16 + 1 + 3)$

$= \min(21, 20) = 20, \qquad l_1[3] = 2$

<u>Computation of $f_2[3]$</u>



$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,\ j-1} + a_{1,j})$

$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

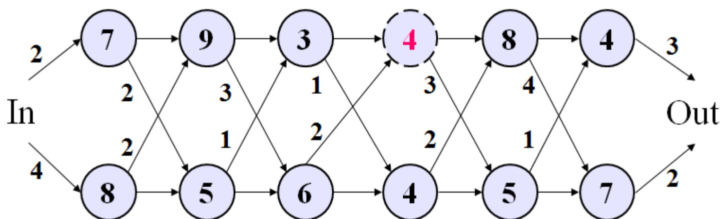
$j = 3$

$f_2[3] = \min(f_2[2] + a_{2,3},\ f_1[2] + t_{1,2} + a_{2,3})$

$= \min(16 + 6,\ 18 + 3 + 6)$

$= \min(22, 27) = 22, \qquad l_2[3] = 2$

<u>Computation of $f_1[4]$</u>



$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,\ j-1} + a_{1,j})$

$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

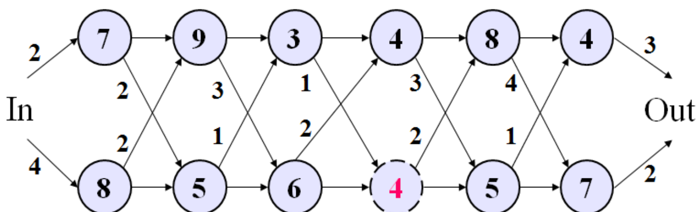
$j = 4$

$f_1[4] = \min(f_1[3] + a_{1,4},\ f_2[3] + t_{2,3} + a_{1,4})$

$= \min(20 + 4,\ 22 + 1 + 4)$

$= \min(24, 27) = 24, \qquad l_1[4] = 1$

<u>Computation of $f_2[4]$</u>



$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,\ j-1} + a_{1,j})$

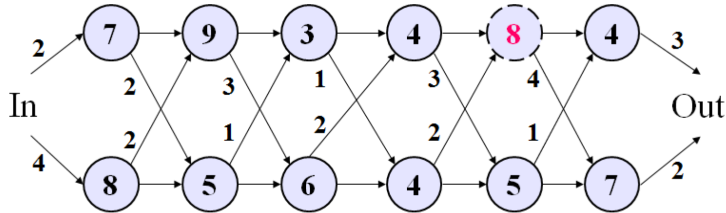
https://studyc.info/

$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

$j = 4$

$$\begin{aligned} f_2[4] &= \min(f_2[3] + a_{2,4},\ f_1[3] + t_{1,3} + a_{2,4}) \\ &= \min(22 + 4,\ 20 + 1 + 4) \\ &= \min(26, 25) = 25, \qquad l_2[4] = 1 \end{aligned}$$

<u>Computation of $f_1[5]$</u>



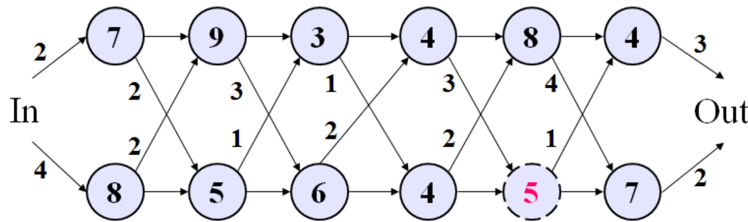
$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$

$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

$j = 5$

$$\begin{aligned} f_1[5] &= \min(f_1[4] + a_{1,5},\ f_2[4] + t_{2,4} + a_{1,5}) \\ &= \min(24 + 8,\ 25 + 2 + 8) \\ &= \min(32, 35) = 32, \qquad l_1[5] = 1 \end{aligned}$$

<u>Computation of $f_2[5]$</u>



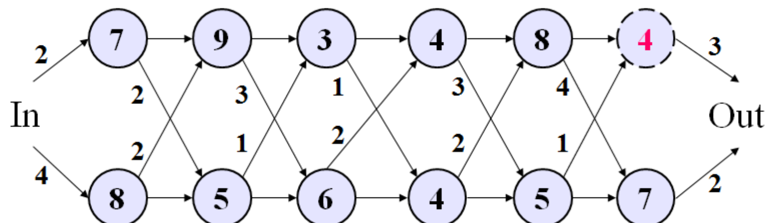
$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$

$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

$j = 5$

$$\begin{aligned} f_2[5] &= \min(f_2[4] + a_{2,5},\ f_1[4] + t_{1,4} + a_{2,5}) \\ &= \min(25 + 5,\ 24 + 3 + 5) \\ &= \min(30, 32) = 30, \qquad l_2[5] = 2 \end{aligned}$$

<u>Computation of $f_1[6]$</u>



$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$

$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

https://studyc.info/

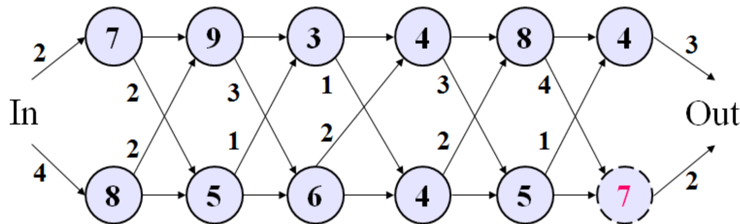
j = 6

$f_1[6] = \min(f_1[5] + a_{1,6}, f_2[5] + t_{2,5} + a_{1,6})$

$= \min(32 + 4, 30 + 1 + 4)$

$= \min(36, 35) = 35,$ $l_1[6] = 2$

Computation of $f_2[6]$



$f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$

$f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})$

j = 6

$f_2[6] = \min(f_2[5] + a_{2,6}, f_1[5] + t_{1,5} + a_{2,6})$

$= \min(30 + 7, 32 + 4 + 7)$

$= \min(37, 43) = 37,$ $l_2[6] = 2$

Keeping Track Constructing Optimal Solution

$f^* = \min(f_1[6] + x_1, f_2[6] + x_2)$

$= \min(35 + 3, 37 + 2)$

$= \min(38, 39) = 38$

$l^* = 1$

$l^* = 1$ => Station $S_{1,6}$

$l_1[6] = 2$ => Station $S_{2,5}$

$l_2[5] = 2$ => Station $S_{2,4}$

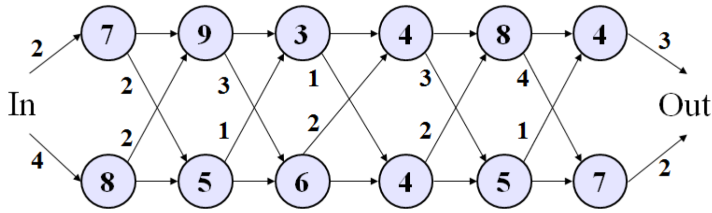
$l_2[4] = 1$ => Station $S_{1,3}$

$l_1[3] = 2$ => Station $S_{2,2}$

$l_2[2] = 1$ => Station $S_{1,1}$

Entire Solution Set: Assembly-Line Scheduling

https://studyc.info/



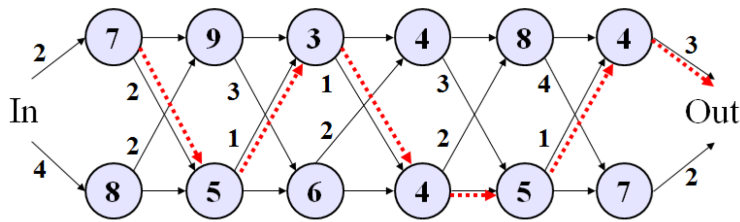
$f_i(j)$ \ j	1	2	3	4	5	6
1	9	18	20	24	32	35
2	12	16	22	25	30	37

$f^* = 38$

$l_i(j)$ \ j	2	3	4	5	6
1	1	2	1	1	2
2	1	2	1	2	2

$l^* = 1$

Fastest Way: Assembly-Line Scheduling



l^* = 1 => Station $S_{1,6}$
$l_1[6]$ = 2 => Station $S_{2,5}$
$l_2[5]$ = 2 => Station $S_{2,4}$
$l_2[4]$ = 1 => Station $S_{1,3}$
$l_1[3]$ = 2 => Station $S_{2,2}$
$l_2[2]$ = 1 => Station $S_{1,1}$

Lecture 18 2-Line Assembly Scheduling Problem

- There are two assembly lines each with n stations
- The jth station on line i is denoted by $S_{i,j}$
- The assembly time at that station is $a_{i,j}$.
- An auto enters factory, goes into line i taking time e_i
- After going through the jth station on a line i, the auto goes on to the (j+1)st station on either line
- There is no transfer cost if it stays on the same line
- It takes time $t_{i,j}$ to transfer to other line after station $S_{i,j}$
- After exiting the nth station on a line, it takes time x_i for the completed auto to exit the factory.
- Problem is to determine which stations to choose from lines 1 and 2 to minimize total time through the factory.

Mathematical Model Defining Objective Function

Base Cases

- $f_1[1] = e_1 + a_{1,1}$
- $f_2[1] = e_2 + a_{2,1}$

Two possible ways of computing $f_1[j]$

- $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$ OR $f_1[j] = f_1[j-1] + a_{1,j}$

  For j = 2, 3, . . ., n

  $f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$

Symmetrically

For j = 2, 3, . . ., n

$f_2[j] = \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j})$

Objective function = $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$

Dynamic Algorithm

FASTEST-WAY(*a, t, e, x, n*)

```
1  f1[1] = e1 + a1,1
2  f2[1] = e2 + a2,1
3  for j = 2 to n
4      do if f1[j - 1] + a1,j ≤ f2[j - 1] + t2,j - 1 + a1,j
5             then f1[j] = f1[j - 1] + a1,j
6                  l1[j] = 1
7             else f1[j] = f2[j - 1] + t2,j - 1 + a1,j
8                  l1[j] = 2
9          if f2[j - 1] + a2,j ≤ f1[j - 1] + t1,j - 1 + a2,j
10 then f2[j] = f2[j - 1] + a2,j
11                 l2[j] = 2
12            else f2[j] = f1[j - 1] + t1,j - 1 + a2,j
13                 l2[j] = 1
14  if f1[n] + x1 ≤ f2[n] + x2
15   then f* = f1[n] + x1
16        l* = 1
17   else f* = f2[n] + x2
```

https://studyc.info/

18 $l^* = 2$

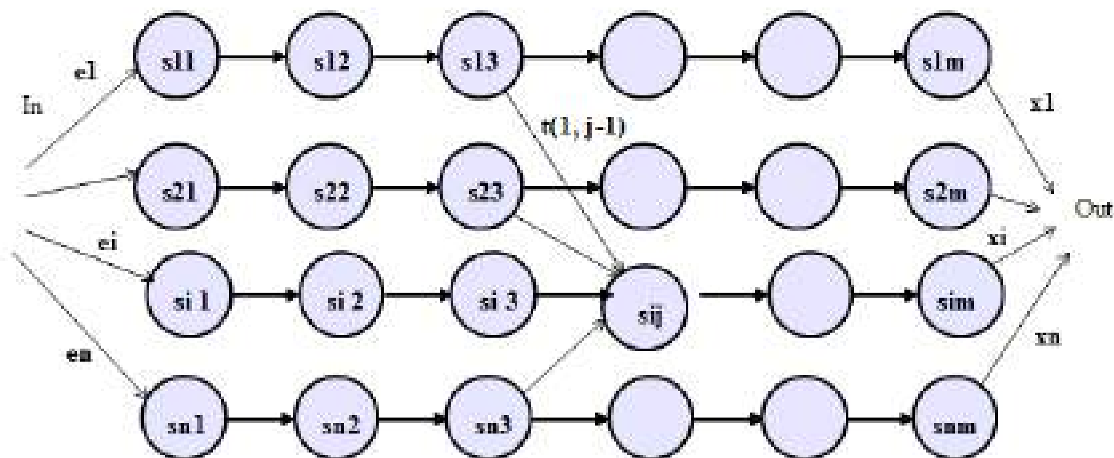
Optimal Solution: Constructing The Fastest Way

1. Print-Stations (l, n)
2. i ← l*
3. print “line” i “, station” n
4. **for** j ← n **downto** 2
5. **do** i ← $l_i[j]$
6. print “line” i “, station” j - 1

n-Line Assembly Problem

- There are n assembly lines each with m stations
- The jth station on line i is denoted by $S_{i,j}$
- The assembly time at that station is $a_{i,j}$.
- An auto enters factory, goes into line i taking time e_i
- After going through the jth station on a line i, the auto goes on to the (j+1)st station on either line
- It takes time $t_{i,j}$ to transfer from line i, station j to line i’ and station j+1
- After exiting the nth station on a line i, it takes time x_i for the completed auto to exit the factory.
- Problem is to determine which stations to choose from lines 1 to n to minimize total time through the factory.

n-Line: Brute Force Solution



Total Computational Time = possible ways to enter in stations at level n x one way Cost
Possible ways to enter in stations at level 1 = n^1
Possible ways to enter in stations at level 2 = n^2 . . .
Possible ways to enter in stations at level m = n^m
Total Computational Time = $\Theta(m.m^n)$

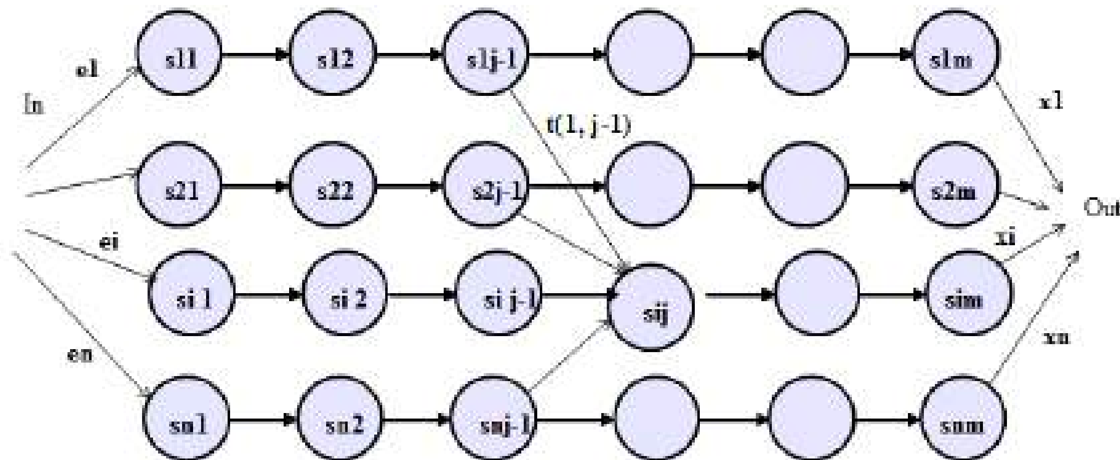
Dynamic Solution

Notations: n-Line Assembly

https://studyc.info/

- Let $f_i[j]$ = fastest time from starting point to station $S_{i, j}$
- $f_1[m]$ = fastest time from starting point to station $S_{1\ m}$
- $f_2[m]$ = fastest time from starting point to station $S_{2,m}$
- $f_n[m]$ = fastest time from starting point to station $S_{n,m}$
- $l_i[j]$ = The line number, 1 to n, whose station j-1 is used in a fastest way through station $S_{i', j}$
- $t_i[j-1]$ = transfer time from station $S_{i, j-1}$ to station $S_{i,, j}$
- a[i, j] = time of assembling at station $S_{i, j}$
- f* = is minimum time through any way
- l* = the line no. whose m^{th} station is used in a fastest way

Possible Lines to reach Station S(i, j)



Time from Line 1, f[1, j-1] + t[1, j-1] + a[i, j]
Time from Line 2, f[2, j-1] + t[2, j-1] + a[i, j]
Time from Line 3, f[3, j-1] + t[3, j-1] + a[i, j]
. . .
Time from Line n, f[n, j-1] + t[n, j-1] + a[i, j]

Values of f(i, j) and l* at Station S(i, j)

f[i, j] = min{f[1, j-1] + t[1, j-1] + a[i, j], f[2, j-1] + t[2, j-1] + a[i, j], . . , f[n, j-1] + t[n, j-1] + a[i, j]}
f[1, 1] = e_1 + a[1, 1]; f [2, 1] = e_2 + a[2, 1], … ,f [n, 1] = e_n + a[n, 1]
f* = min{f[1, n] +x_1, f[2, n] + x_2, . . , f[n, m] + x_n}
*l** = line number of mth station used

n-Line Assembly: Dynamic Algorithm

FASTEST-WAY(a, t, e, x, n, m)

```
1    for i ← 1 to n
2        f[i,1] ←e[i] + a[i,1]
3    for j ← 2 to m
4      for i ← 1 to n
5        f[i, j] ← f[1, j-1] + t[1, j-1] + a[i, j]
```

```
6                         L[1, j] = 1
7                  for k ← 2 to n
8                    if f[i,j] > f[k, j-1] + t[2, j-1] + a[i, j]
9                      then f[i,j] ← f[k, j-1] + t[2, j-1] + a[i, j]
                 L[i, j] = k
10               end if
11       f* ← f[1, m] + x[1]
12       l* = 1
13       for k ← 2 to n
14           if f* > f[k, m] + x[k]
              15  then f* ← f[k, m] + x[k]
              16  l* = k
```

Constructing the Fastest Way: n-Line

```
1. Print-Stations (l*, m)
2.    i ← l*
3.    print "line" i ", station" m
4.    for j ← m downto 2
5.        do i ← l_i[j]
6.               print "line" i ", station" j - 1
```

Generalization: Cyclic Assembly Line Scheduling

Title: Moving policies in cyclic assembly line scheduling

Source: Theoretical Computer Science, Volume 351, Issue (February 2006)

Summary: Assembly line problem occurs in various kinds of production automation. In this paper, originality lies in the automated manufacturing of PC boards.

- In this case, the assembly line has to process number of identical work pieces in a cyclic fashion. In contrast to common variant of assembly line scheduling.
- Each station may process parts of several work-pieces at the same time, and parts of a work-piece may be processed by several stations at the same time.

Application: Multiprocessor Scheduling

- The assembly line problem is well known in the area of multiprocessor scheduling.
- In this problem, we are given a set of tasks to be executed by a system with n identical processors.
- Each task, Ti, requires a fixed, known time pi to execute.
- Tasks are indivisible, so that at most one processor may be executing a given task at any time
- They are un-interruptible, i.e., once assigned a task, may not leave it until task is complete.
- The precedence ordering restrictions between tasks may be represented by a tree or forest of trees

https://studyc.info/

Lecture 19 0-1 Knapsack Problem using Dynamic Programming

General Knapsack Problem

- Given a set of items, each with a cost and a value, then determine the items to include in a collection so that the total cost is less than some given cost and the total value is as large as possible.
- Knapsack problem is of combinatorial optimization
- It derives its name from the maximization problem of choosing possible essentials that can fit into one bag, of maximum weight, to be carried on a trip.
- A similar problem very often appears in business, complexity theory, cryptography and applied mathematics.

0-1 Knapsack Problem Statement

The knapsack problem arises whenever there is resource allocation with no financial constraints

Problem Statement

A thief robbing a store and can carry a maximal weight of W into his knapsack. There are n items and i^{th} item weight is w_i and worth is v_i dollars. What items should thief take, not exceeding the bag capacity, to maximize value?

Assumption:

The items may not be broken into smaller pieces, so thief may decide either to take an item or to leave it, but may not take a fraction of an item.

0-1 Knapsack Problem (Another Statement)

Problem Statement

You are in Japan on an official visit and want to make shopping from a store (Best Denki). A list of required items is available at the store. You are given a bag (knapsack), of fixed capacity, and only you can fill this bag with the selected items from the list. Every item has a value (cost) and weight and your objective is to seek most valuable set of items which you can buy not exceeding bag limit.

Assumption: Each item must be put entirely in the knapsack or not included at all that is why the problem is called 0-1 knapsack problem

Remarks: Because an item cannot be broken up arbitrarily, so it is its 0-1 property that makes the knapsack problem hard. If an item can be broken and allowed to take part of it then algorithm can be solved using greedy approach optimally

Notations: 0-1 Knapsack Problem Construction

https://studyc.info/

- You have prepared a list of n objects for which you are interested to buy, The items are numbered as $i_1, i_2, \ldots, i_n$
- Capacity of bag is W
- Each item i has value v_i, and weigh w_i
- We want to select a set of items among $i_1, i_2, \ldots, i_n$ which do not exceed (in total weight) capacity W of the bag
- Total value of selected items must be maximum
- How should we select the items?

Model: 0-1 Knapsack Problem Construction

Formal Construction of Problem

- Given a list: $i_1, i_2, \ldots, i_n$, values: $v_1, v_2, \ldots, v_n$ and weights: $w_1, w_2, \ldots, w_n$ respectively

- Of course $W \geq 0$, and we wish to find a set S of items such that $S \subseteq \{i_1, i_2, \ldots, i_n\}$ that maximizes $\sum_{i \in S} v_i$ subject to $\sum_{i \in S} w_i \leq W$.

Brute Force Solution

- Compute all the subsets of $\{i_1, i_2, \ldots, i_n\}$, there will be 2^n number of subsets.
- Find sum of the weights of total items in each set and list only those sets whose sum does not increase by W (capacity of knapsack)
- Compute sum of values of items in each selected list and find the highest one
- This highest value is the required solution
- The computational cost of Brute Force Approach is exponential and not economical
- Find some other way!

Divide and Conquer Approach

Approach

- Partition the knapsack problem into sub-problems
- Find the solutions of the sub-problems
- Combine these solutions to solve original problem

Comments

- In this case the sub-problems are not independent
- And the sub-problems share sub-sub-problems
- Algorithm repeatedly solves common sub-sub-problems and takes more effort than required
- Because this is an optimization problem and hence dynamic approach is another solution if we are able to construct problem dynamically

Steps in Dynamic Programming

Step1 (Structure):

https://studyc.info/

- Characterize the structure of an optimal solution
- Next decompose the problem into sub-problems
- Relate structure of the optimal solution of original problem and solutions of sub-problems

Step 2 (Principal of Optimality)

- Define value of an optimal solution recursively
- Then express solution of the main problem in terms of optimal solutions of sub-problems.

Step3 (Bottom-up Computation):

In this step, compute the value of an optimal solution in a bottom-up fashion by using structure of the table already constructed.

Step 4 (Construction of an Optimal Solution)

Construct an optimal solution from the computed information based on Steps 1-3.

Note: Sometime people combine the steps 3 and 4. Step 1-3 form basis of dynamic problem. Step 4 may be omitted if only optimal solution of the problem is required

Mathematical Model: Dynamic Programming

Step1 (Structure):

- Decompose problem into smaller problems
- Construct an array V[0..n, 0..W]
- V[i, w] = maximum value of items selected from {1, 2,. . ., i}, that can fit into a bag with capacity w, where $1 \leq i \leq n$, $1 \leq w \leq W$
- V[n, W] = contains maximum value of the items selected from {1,2,…,n} that can fit into the bag with capacity W storage
- Hence V[n, W] is the required solution for our knapsack problem

Step 2 (Principal of Optimality)

- Recursively define value of an optimal solution in terms of solutions to sub-problems
  Base Case: Since
- V[0, w] = 0, $0 \leq w \leq W$, no items are available
- V[0, w] = $-\infty$, $w < 0$, invalid
- V[i, 0] = 0, $0 \leq i \leq n$, no capacity available
  Recursion:
  $V[i, w] = \max(V[i-1, w], v_i + V[i-1, w - w_i])$
  for $1 \leq i \leq n$, $0 \leq w \leq W$

Proof of Correctness

Prove that: $V[i, w] = \max(V[i-1, w], v_i + V[i-1, w - w_i])$ for $1 \leq i \leq n$, $0 \leq w \leq W$

https://studyc.info/

Proof:

To compute V[i, w], we have only two choices for i

1. Do not Select Item i
   Items left = {1,2,. . . , i - 1} and
   storage limit = w, hence
   Max. value, selected from {1,2, …,i} = V[i-1,w], (1)

2. Select Item i (possible if $w_i \leq w$)

- In this way, we gain value v_i but use capacity w_i
- Items left = {1,2,. . . , i-1}, storage limit = w - w_i,
- Max. value, from items {1,2, …,i-1} = V[i-1,w – w_i]
- Total value if we select item i = v_i + V[i-1,w – w_i]
- Finally, the solution will be optimal if we take the maximum of
  V[i-1,w] and
  v_i + V[i-1,w – w_i]
- Hence V[i, w] = max(V[i-1,w], v_i + V[i-1,w – w_i]

Problem: Developing Algorithm for Knapsack

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

Capacity: 10

V[1, 1] = 0, V[1, 2] = 0, V[1, 3] = 0, V[1, 4] = 0

V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);
V[1, 5] = max(V[0, 5], v_1 + V[0, 5 – w_1]);
= max(V[0, 5], 10 + V[0, 5 - 5])
= max(V[0, 5], 10 + V[0, 0])
= max(0, 10 + 0) = max(0, 10) = 10

Keep(1, 5) = 1

V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);
V[1, 6] = max(V[0, 6], v_1 + V[0, 6 – w_1]);
= max(V[0, 6], 10 + V[0, 6 - 5])
= max(V[0, 6], 10 + V[0, 1])
= max(0, 10 + 0) = max(0, 10) = 10,

V[1, 7] = max(V[0, 7], v_1 + V[0, 7 – w_1]);
= max(V[0, 7], 10 + V[0, 7 - 5])
= max(V[0, 7], 10 + V[0, 2])
= max(0, 10 + 0) = max(0, 10) = 10

Keep(1, 6) = 1; Keep(1, 7) = 1

https://studyc.info/

$V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

$V[1, 8] = \max(V[0, 8], v_1 + V[0, 8 - w_1]);$
$= \max(V[0, 8], 10 + V[0, 8 - 5])$
$= \max(V[0, 8], 10 + V[0, 3])$
$= \max(0, 10 + 0) = \max(0, 10) = 10$

$V[1, 9] = \max(V[0, 9], v_1 + V[0, 9 - w_1]);$
$= \max(V[0, 9], 10 + V[0, 9 - 5])$
$= \max(V[0, 7], 10 + V[0, 4])$
$= \max(0, 10 + 0) = \max(0, 10) = 10$

Keep(1, 8) = 1; Keep(1, 9) = 1
$V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$
$V[1, 10] = \max(V[0, 10], v_1 + V[0, 10 - w_1]);$
$= \max(V[0, 10], 10 + V[0, 10 - 5])$
$= \max(V[0, 10], 10 + V[0, 5])$
$= \max(0, 10 + 0) = \max(0, 10) = 10$
Keep(1, 10) = 1

V[2, 1] = 0; V[2, 2] = 0; V[2, 3] = 0;

$V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$
$V[2, 4] = \max(V[1, 4], v_2 + V[1, 4 - w_2]);$
$= \max(V[1, 4], 40 + V[1, 4 - 4])$
$= \max(V[1, 4], 40 + V[1, 0])$
$= \max(0, 40 + 0) = \max(0, 40) = 40$

$V[2, 5] = \max(V[1, 5], v_2 + V[1, 5 - w_2]);$
$= \max(V[1, 5], 40 + V[1, 5 - 4])$
$= \max(V[1, 5], 40 + V[1, 1])$
$= \max(10, 40 + 0) = \max(0, 40) = 40$
Keep(2, 4) = 1; Keep(2, 5) = 1

$V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

$V[2, 6] = \max(V[1, 6], v_2 + V[1, 6 - w_2]);$
$= \max(V[1, 6], 40 + V[1, 6 - 4])$
$= \max(V[1, 6], 40 + V[1, 2])$
$= \max(10, 40 + 0) = \max(10, 40) = 40$

$V[2, 7] = \max(V[1, 7], v_2 + V[1, 7 - w_2]);$
$= \max(V[1, 7], 40 + V[1, 7 - 4])$
$= \max(V[1, 7], 40 + V[1, 2])$
$= \max(10, 40 + 0) = \max(10, 40) = 40$

$V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

https://studyc.info/

$V[2, 8] = max(V[1, 8], v_2 + V[1, 8 – w_2]);$
$= max(V[1, 8], 40 + V[1, 8 - 4])$
$= max(V[1, 8], 40 + V[1, 4])$
$= max(10, 40 + 0) = max(10, 40) = 40$

$V[2, 9] = max(V[1, 9], v_2 + V[1, 9 – w_2]);$
$= max(V[1, 9], 40 + V[1, 9 - 4])$
$= max(V[1, 9], 40 + V[1, 5])$
$= max(10, 40 + 10) = max(10, 50) = 50$

$V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);$
$V[2, 10] = max(V[1, 10], v_2 + V[1, 10 – w_2]);$
$= max(V[1, 10], 40 + V[1, 10 - 4])$
$= max(V[1, 10], 40 + V[1, 6])$
$= max(10, 40 + 10) = max(10, 50) = 50$

V[3, 1] = 0;   V[3, 2] = 0;   V[3, 3] = 0;

$V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);$
$V[3, 4] = max(V[2, 4], v_3 + V[2, 4 – w_3]);$
$= max(V[2, 4], 30 + V[2, 4 - 6])$
$= max(V[2, 4], 30 + V[2, -2]) = V[2, 4] = 40$

$V[3, 5] = max(V[2, 5], v_3 + V[2, 5 – w_2]);$
$= max(V[2, 5], 30 + V[2, 5 - 6])$
$= max(V[2, 5], 30 + V[2, -1])$
$= V[2, 5] = 40$

$V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);$
$V[3, 6] = max(V[2, 6], v_3 + V[2, 6 – w_3]);$
$= max(V[2, 6], 30 + V[2, 6 - 6])$
$= max(V[2, 6], 30 + V[2, 0])$
$= max(V[2, 6], 30 + V[2, 0])$
$= max(40, 30) = 40$

$V[3, 7] = max(V[2, 7], v_3 + V[2, 7 – w_3]);$
$= max(V[2, 7], 30 + V[2, 7 - 6])$
$= max(V[2, 7], 30 + V[2, 1])$
$= max(V[2, 7], 30 + V[2, 1])$
$= max(40, 30) = 40$

$V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);$
$V[3, 8] = max(V[2, 8], v_3 + V[2, 8 – w_3]);$
$= max(V[2, 8], 30 + V[2, 8 - 6])$
$= max(V[2, 8], 30 + V[2, 2])$
$= max(V[2, 8], 30 + V[2, 2])$

https://studyc.info/

$= max(40, 30 + 0) = 40$

$V[3, 9] = max(V[2, 9], v_3 + V[2, 9 – w_3]);$
$= max(V[2, 9], 30 + V[2, 9 - 6])$
$= max(V[2, 9], 30 + V[2, 3])$
$= max(V[2, 9], 30 + V[2, 3])$
$= max(50, 30 + 0) = 50$

$V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);$
$V[3, 10] = max(V[2, 10], v_3 + V[2, 10 – w_3]);$
$= max(V[2, 10], 30 + V[2, 10 - 6])$
$= max(V[2, 10], 30 + V[2, 4])$
$= max(V[2, 10], 30 + V[2, 4])$
$= max(50, 30 + 40) = 70$

V[4, 1] = 0;   V[4, 2] = 0;

$V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);$
$V[4, 3] = max(V[3, 3], v_4 + V[3, 3 – w_4]);$
$= max(V[3, 3], 50 + V[3, 3 - 3])$
$= max(V[3, 3], 50 + V[3, 3 - 3])$
$= max(V[3, 3], 50 + V[3, 0])$ $= max(0, 50) = 50$

$V[4, 4] = max(V[3, 4], v_4 + V[3, 4 – w_4]);$
$= max(V[3, 4], 50 + V[3, 4 - 3])$
$= max(V[3, 4], 50 + V[3, 4 - 3])$
$= max(V[3, 4], 50 + V[3, 1])$
$= max(40, 50) = 50$

$V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);$
$V[4, 5] = max(V[3, 5], v_4 + V[3, 5 – w_4]);$
$= max(V[3, 5], 50 + V[3, 5 - 3])$
$= max(V[3, 5], 50 + V[3, 5 - 3])$
$= max(V[3, 5], 50 + V[3, 2])$
$= max(40, 50) = 50$

$V[4, 6] = max(V[3, 6], v_4 + V[3, 6 – w_4]);$
$= max(V[3, 6], 50 + V[3, 6 - 3])$
$= max(V[3, 6], 50 + V[3, 6 - 3])$
$= max(V[3, 6], 50 + V[3, 3])$
$= max(40, 50) = 50$

$V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);$
$V[4, 7] = max(V[3, 7], v_4 + V[3, 7 – w_4]);$
$= max(V[3, 7], 50 + V[3, 7 - 3])$
$= max(V[3, 7], 50 + V[3, 7 - 3])$
$= max(V[3, 7], 50 + V[3, 4])$

https://studyc.info/

$= \max(40, 50 + 40) = 90$

$V[4, 8] = \max(V[3, 8], v_4 + V[3, 8 – w_4]);$
$= \max(V[3, 8], 50 + V[3, 8 - 3])$
$= \max(V[3, 8], 50 + V[3, 8 - 3])$
$= \max(V[3, 8], 50 + V[3, 5])$
$= \max(40, 50 + 40) = 90$

$V[i, j] = \max(V[i-1, j], v_i + V[i-1, j – w_i]);$
$V[4, 9] = \max(V[3, 9], v_4 + V[3, 9 – w_4]);$
$= \max(V[3, 9], 50 + V[3, 9 - 3])$
$= \max(V[3, 9], 50 + V[3, 9 - 3])$
$= \max(V[3, 9], 50 + V[3, 6])$
$= \max(50, 50 + 40) = 90$

$V[4, 10] = \max(V[3, 10], v_4 + V[3, 10 – w_4]);$
$= \max(V[3, 10], 50 + V[3, 10 - 3])$
$= \max(V[3, 10], 50 + V[3, 10 - 3])$
$= \max(V[3, 10], 50 + V[3, 7])$
$= \max(70, 50 + 40) = 90$; Keep(4, 10) = 1

https://studyc.info/

Lecture 20 0-1 Knapsack Problem's Algorithm (using Dynamic Programming) and Optimal Weight Triangulation

Optimal Value: Entire Solution

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

Let W = 10

Final Solution: V[4, 10] = 90

Items selected = {2, 4}

V[i, w]	W = 0	1	2	3	4	5	6	7	8	9	10
i = 0	0	0	0	0	0	0	0	0	0	0	0
i = 1	0	0	0	0	0	10	10	10	10	10	10
i = 2	0	0	0	0	40	40	40	40	40	50	50
i = 3	0	0	0	0	40	40	40	40	40	50	70
i = 4	0	0	0	50	50	50	50	90	90	90	90

Constructing Optimal Solution

$V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

i = 4

V[4, 10] = max(70, 50 + 40) = 90; Keep(4, 10) = 1

i = 3

V[3, 10 - 3] = V[3, 7] = max(40, 30) = 40 Keep(3, 7) = 0

i = 2

V[2, 7] = max(10, 40) = 40 Keep(2, 7) = 1

i = 1

V[1, 7-4] = V[1, 3] = 0 Keep(1, 3) = 0

Algorithm: Dynamic Programming

```
KnapSack (v, w, n, W)
for (i = 1 to n), V[i, 0] = 0;
for (j = 0 to W), V[0, j] = 0;
 for (i = 1 to n)
        for (j = 1 to W)
                if (w(i) ≤ j)
                        V[i, j] = max(V[i-1, j], vi + V[i-1, j – wi]);
                else
                        V[i, j] = V[i-1, j];
```

https://studyc.info/

Return V[n, W]

Time Complexity: O(n.W)

Output Elements: Knapsack Algorithm

How do we use all values *keep[i, w],* to determine a subset S of items having the maximum value?

- If *keep[n, w]* is 1, then $n \in S$, and we can repeat for *keep[n-1, W - w_n]*
- If *keep[n, w]* is 0, then $n \notin S$ and we can repeat for *keep[n-1, W]*
- Following is a partial program for this output elements

```
K = W;
for (i = n down to 1)
        if keep[i, K] = = 1
                output i
                K = K – wi
```

Complete: Dynamic Programming Algorithm

```
KnapSack(v, w, n, W)
for (w = 0 to W), V[0, w] = 0;  for (i = 1 to n), V[i, 0] = 0;
for (i = 1 to n)
        for (w = 1 to W)
                if ((w(i) ≤ w) and (vi + V[i-1,w – wi] > V[i-1,w]))
                        V[i, w] = (vi + V[i-1,w – wi];
                        keep[i, w] = 1;
                else
                        V[i, w] = V[i-1,w];
                        keep[i, w] = 0;
K = W;
                for (i = n down to 1)
                        if keep[i, K] = = 1
                                output i
                                K = K – wi
Return V[n, W]
```

1. Generalizations ($x_i \in \{0, 1\}$)

- Common to all versions are a set of n items, with each item $1 \leq j \leq n$ having an associated profit p_j and weight w_j.
- The objective is to pick some of the items, with maximal total profit, obeying that maximum total weight limit W.

https://studyc.info/

- Generally, coefficients are scaled to become integers, and they are almost always assumed to be positive.
- The knapsack problem in its most basic form:

maximize $\sum_{i=1}^{n} p_i x_i$

subject to $\sum_{i=1}^{n} w_i x_i \le W$

$x_i \in \{0, 1\}, \ \forall \ 1 \le i \le n$

2. Specialization (weight = profit)

- If for each item the profit and weight are identical, we get the subset sum problem
- Often called the decision problem

maximize $\sum_{i=1}^{n} p_i x_i$

subject to $\sum_{i=1}^{n} p_i x_i \le W$

$x_i \in \{0, 1\}, \ \forall \ 1 \le i \le n$

3. Generalizations (more than one objects)

- If each item can be chosen multiple times, we get the bounded knapsack problem.
- Suppose, weight of each item is at least 1 unit, then we can never choose an item more than W times.
- This is another variation in the basic form
- Now the problem will become

maximize $\sum_{i=1}^{n} p_i x_i$

subject to $\sum_{i=1}^{n} w_i x_i \le W$

$x_i \in \{0, 1, \ \ldots, W\}, \ \forall \ 1 \le i \le n$

4. Generalizations (K Classes)

- If the items are subdivided into k classes denoted N_i
- And exactly one item must be taken from each class
- We get the multiple choice knapsack problem
- In this case our optimized mathematical model is

maximize $\sum_{i=1}^{k} \sum_{j \in N_i} p_{ij} x_{ij}$

subject to $\sum_{i=1}^{n} \sum_{j \in N_i} w_{ij} x_{ij} \le W$ where $\sum_{j \in N_i} x_{ij} = 1$

$\forall \ 1 \le i \le k$

$x_{ij} \in \{0, 1\}, \ \forall \ 1 \le i \le k, j \in N_i$

5. Generalizations (more than one knapsacks)

https://studyc.info/

- If there are n items, m knapsacks with capacities W_i
- We get the multiple knapsack problem

$$\text{maximize } \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij}$$

$$\text{subject to } \sum_{j=1}^{n} w_j x_{ij} \leq W_i \qquad 1 \leq i \leq m$$

$$1 \leq j \leq n$$

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad 1 \leq i \leq n$$

$$x_{ij} \in \{0, 1\}, \; 1 \leq i \leq m \text{ and } 1 \leq j \leq n$$

6. Item's Different Weight in Different Knapsack)

- If in the multiple knapsack problem, the weights are not the same in every container
- We are allowed to choose each item multiple times, we get multiple constrained knapsack problem

$$\text{maximize } \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to } \sum_{j=1}^{n} w_{ij} x_j \leq W_i \qquad 1 \leq i \leq m$$

$$x_j \geq 0, \;\; x_j \in Z$$

Optimal Weight Triangulation

Why Polygon Triangulation?

- Finite element method is a technique for solving numerical problems e.g. stress or heat flow simulations of any kind of systems
- It involves dividing a shape into simple elements for example triangles
- Then formulating a set of linear equations describing relations between simulated quantities in each element, and solving these equations.
- The time and accuracy both, in solution, depend on the quality of dividing into triangles
- Generally, it is desired that triangles must be as close to equilateral as possible

Similarity: Optimal Polygon Triangulation, other Problem

- Optimal triangulation problem is very similar to matrix chain multiplication
- It is an excellent approach to make one to one corresponding between two problems and
- Then solving one problem based on the approach already used in the solution of the other problem

https://studyc.info/

- This is what we are going to do in solving an optimal solution of the triangulation problem which is very popular in computational geometry
- Applications of this problem can be observed in many other areas where division of structures is required before performing computation over it.

Basic Concepts

Polygon: A set of finite piecewise-linear, closed curve in a plane is called a polygon

Sides: The pieces of the polygon are called its sides

Vertex: A point joining two consecutive sides is called a vertex

Interior: The set of points in the plane enclosed by a simple polygon forms interior of the polygon

Boundary: The set of point on the polygon forms its boundary

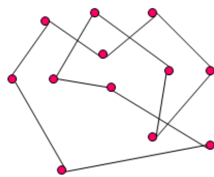
Exterior: The set of points surrounding the polygon form its exterior

Simple Polygon: A polygon is simple if it does not cross itself, i.e., if its sides do not intersect one another except for two consecutive sides sharing a common vertex.

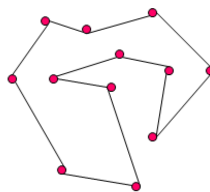
Subdivision of Polygon: A simple polygon subdivides the plane into its interior, its boundary and its exterior.

Convex Polygon: A simple polygon is convex if given any two points on its boundary or in its interior, all points on the line segment drawn between them are contained in the polygon's boundary or interior.

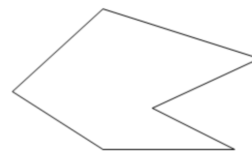
Polygons



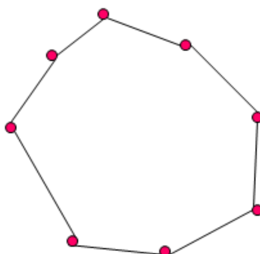
Polygon

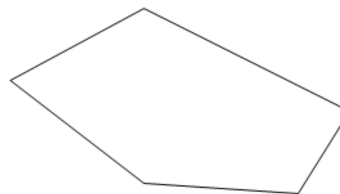


Simple Polygon



Simple Polygon





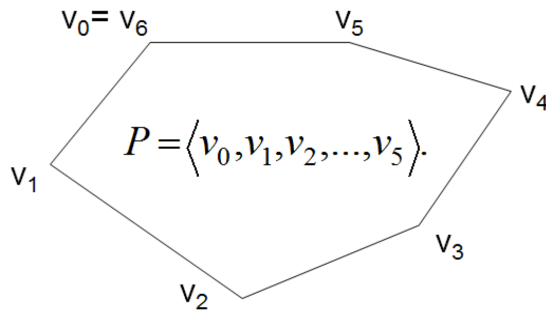
Convex Polygons

https://studyc.info/

Labeling Convex Polygons

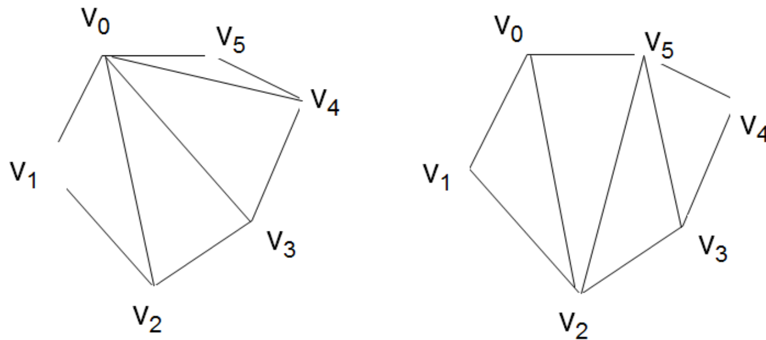
For a convex polygon, it is assumed that its vertices are labeled in counterclockwise order
$P = \langle v_0, v_1, v_2, \ldots, v_{n-1} \rangle$

We assume that indexing is done modulo *n*, so $v_0 = v_n$ and the above polygon P has *n* number of vertices



Chords in Polygons

- Given two non-adjacent vertices v_i, v_j of a convex polygon ($i < j$), the line segment v_iv_j is called a chord.
- For two non-adjacent vertices v_i and v_j of a simple polygon ($i < j$), line segment v_iv_j is a chord if interior of the segment lies entirely in the interior of polygon
- Any chord subdivides a polygon into two polygons



Optimal Weight Triangulation Problem

- A triangulation of a convex polygon is a maximal set T of pair-wise non-crossing chords, i.e., every chord not in T intersects the interior of some chord in T
- It is easy to see that such a set subdivides interior of polygon into a collection of triangles, pair-wise disjoint

Problem Statement

https://studyc.info/

Given a convex polygon, determine a triangulation that minimizes sum of the perimeters of its triangles

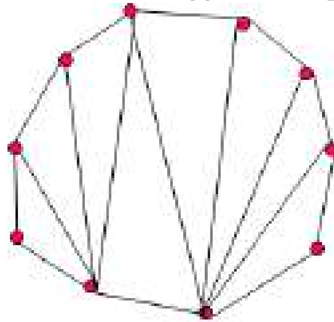
Analysis

- Given three distinct vertices, v_i, v_j and v_k.
- Define a weight of associated triangle by a function
  $w(v_i, v_j, v_k) = |v_i\, v_j| + |v_j\, v_k| + |v_k\, v_i|$,

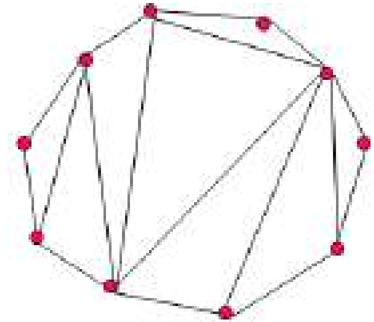
where $|v_i\, v_j|$ denotes length of line segment (v_i, v_j).

Brute Force Triangulation: Triangulation

- In general, given a convex polygon, there are exponential number of possible triangulations.
- There are many criteria that are used depending on application. For example, you have to minimize the value of cable in designing such triangulation
- This suggests the optimal solution to this problem

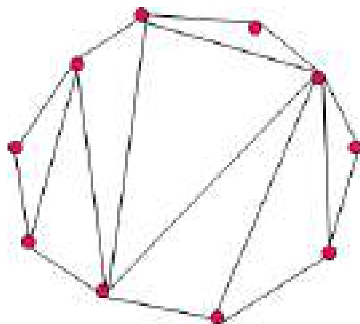


Triangulation

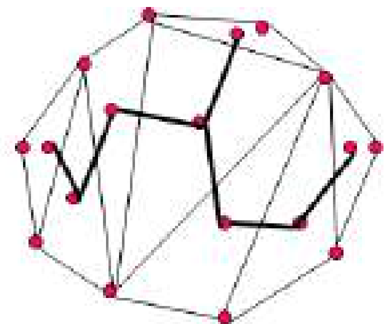


Dual Graphs

- Dual graph of a triangulation is a graph whose vertices are the triangles, and in which two vertices are adjacent if the corresponding both triangles share a common chord
- It is to be noted that dual graph is a tree. And hence algorithms for traversing trees can be used for traversing the triangles of that triangulation



Dual Graph



Observations in Dual Graph

- Each internal node corresponds to one triangle
- Each edge between internal nodes corresponds to one chord of triangulation.
- Now for given n-vertex polygon
  - n-2 internal nodes which are in fact triangles and

https://studyc.info/

- n-3 edges which are chords

Proof of Lemmas

Lemma 1: A triangulation of a simple polygon, with n vertices, has n-2 number of triangles.

Proof: Proof is done using mathematical induction

Basis Step

Suppose that there three vertices, polygon will be a triangle, i.e. there are 3 - 2 = 1 number of triangles. Hence statement is true for n = 3

If there are 4 vertices, polygon will be in fact a rectangle, divide it into two triangles. The result is true. Hence statement is true for n = 4

Inductive Hypothesis

Let us suppose that statement is true for n = k, i.e., if there are k vertices then there are k-2 number of triangles

Claim: Now we have to prove that if there are k+1 vertices there must be k+1-2 = k-1, number of triangles.

Since for k vertices there are k-2 triangles. Insert one more point at boundary of polygon

In fact point will be inserted at boundary of one of the triangles. So the triangle will become rectangle. Divide it into two triangles. It will increase one more triangle in the division. Hence it becomes; k – 2 + 1 = k - 1, number of triangles.

It proves the claim. Hence by mathematical induction it proves that for n number of vertices there are n - 2 number of triangles.

Lemma 2: A triangulation of a simple polygon, with n vertices, has n-3 chords.

Proof

If there are three points, it will be triangle. To make a chord in a polygon, it requires at least four points. So we have to give proof for $n \geq 4$.

Basis Step

Suppose that there are four number of vertices, in this case polygon will be a rectangle, there must be 4 - 3 = 1 number of chords. Hence statement is true for n = 4

Inductive Hypothesis

Let us suppose that the statement is true for n = k, i.e., if there are k vertices then there are k - 3 number of chords of the polygon.

https://studyc.info/

Claim

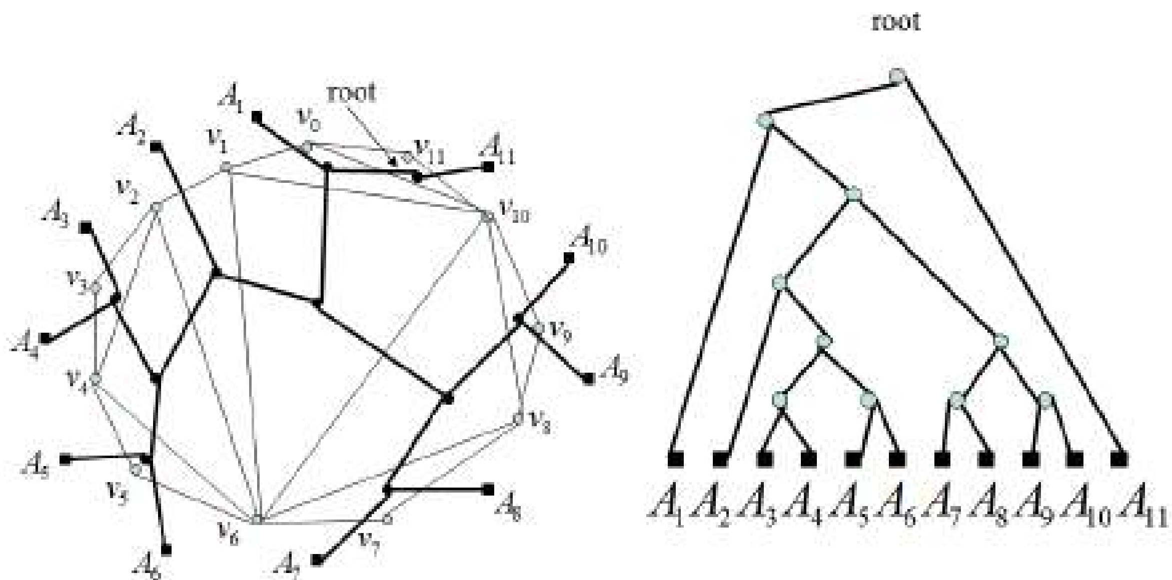
Now we have to prove that if there are k+1 vertices there must be k+1-3 = k-2, number of chords.

Since for k vertices there are k-3 chords. Insert one more point at boundary of polygon

In fact point will be inserted at boundary of one of the triangles. So the triangle will become rectangle. Divide it into two triangles. It will increase one more chord in the division. Hence it becomes k – 3 + 1 = k - 2, number of chords. Proved.

Correspondence to Binary Trees

Relationship between optimal triangulation and chain matrix multiplication problem



- In chain matrix multiplication, associated binary tree is the evaluation tree for the multiplication, where the leaves of the tree correspond to the matrices, and each node of the tree is associated with a product of a sequence of two or more matrices.
- Now let us consider an (n+1) sided convex polygon, P = <v_0, v_1, . . ,v_n> and fix one side of it as (v_0 ,v_n)
- Consider a rooted binary tree whose:
  - root node = is the triangle containing side (v_0 ,v_n),
  - internal nodes = are nodes of the dual tree, and
  - leaves = are remaining sides of the tree.
- This partitioning of polygon is equivalent to a binary tree with n-1 leaves, and vice versa.

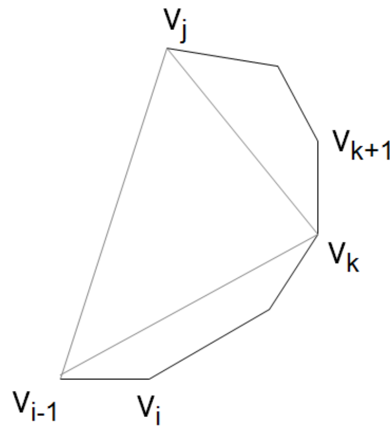
Dynamic Programming Solution

- Let t[i, j] = minimum weight triangulation for the sub-polygon <$v_{i-1}, v_i, \ldots, v_j$>, for $1 \leq i \leq j \leq n$

- We have start with v_{i-1} rather than v_i, to keep the structure as similar as matrix chain multiplication
- It is to be noted that if we can compute $t[i, j]$ for all i and j $(1 \le i \le j \le n)$, then the weight of minimum weight triangulation of the entire polygon will be $t[1, n]$. Hence it is our objective function.
- For the base case $t[i, i] = 0$, for line (v_{i-1}, v_i).

Optimal Substructure

t[i, j] = weight of an optimal triangulation of polygon $<v_{i-1}, v_i, \ldots, v_j>$.
$t[i, j] = \min_k \{ t[i, k] + t[k+1, j] + w(\Delta v_{i-1} v_k v_j) \}, i < j$
$t[i, i] = 0$
$i \le k \le j-1$



- In general, to compute $t[i, j]$, consider the sub-polygon $<v_{i-1}, v_i, \ldots, v_j>$, where $i \le j$.
- One of the chords of this polygon is the side (v_{i-1}, v_j).
- We may split this sub-polygon by introducing a triangle whose base is this chord, and whose third vertex is any vertex v_k, where $i \le k \le j-1$.
- This subdivides the polygon into 2 sub-polygons $<v_{i-1}, \ldots v_k>$ and $<v_{k+1}, \ldots v_j>$, whose minimum weights are $t[i, k]$ and $t[k+1, j]$.
- It leads to following recursive rule computing $t[i, j]$
  $t[i, i] = 0$
  $t[i, j] = \min_{i \le k \le j-1} (t[i, k] + t[k+1, j] + w(v_{i-1} v_k v_j))$ for $i < j$

Algorithm

t[1,1]	t[1,2]	. . .	t[1,n]
	t[2,2]	. . .	t[2,n]
		. . .	. . .
			t[n, n]

$t[i, j] = \min_{i < k < j} (t[i, k] + t[k, j] + w(v_i v_j v_k))$ if $i < j$;
$t[i, j] = 0$ if $i = j$.
function min_weight_tri(p[ ], n)
1. **for** i ←□ 1 **to** n **do**

```
2.          t[i, i] ← 0;
3.  for l ← 2 to n do
4.          for i ← 1 to n – l+1 do
5.                  j ← i + l-1;
6.                  t[i, j] ← ∞;
7.                  for k ← i  to j - 1 do
8.                          q ← t[i, k] + t[k+1, j] + w(vi vj vk);
9.                          if (q < t[i, j]) then
10.                                 t[i, j] ← min(t[i, j], q);
11.                                 v[i, j] ← k;
12.  return( t(1, n) );
```

Computational Cost

$$T(n) = n + \sum_{i=1}^{n} \sum_{j=i+1}^{n} (j - i) = \sum_{i=1}^{n} \sum_{k=1}^{n-i} k$$

$$T(n) = n + \sum_{i=1}^{n} \frac{(n-i)(n-i+1)}{2}$$

$$T(n) = n + \frac{1}{2}\sum_{i=1}^{n}(n^2 + n + 1) - \frac{1}{2}\sum_{i=1}^{n}((1+2n)i - i^2)$$

$$= n + \frac{1}{2}(n^2 + n + 1)\sum_{i=1}^{n} 1 - \frac{1}{2}(1+2n)\sum_{i=1}^{n} i + \frac{1}{2}\sum_{i=1}^{n} i^2$$

$$= \Theta(n^3)$$

https://studyc.info/

Lecture 21 Optimal Weight Triangulation

Longest Common Subsequence Problem

In biological applications, we often want to compare the DNA of two (or more) different organisms. A part of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine. Represent each of the bases by their initial letters. A part of DNA can be expressed as a string over the finite set {A, C, G, T}.

For example, the DNA of one organism may be

S1= CCGGTCGAGTGCGCGGAAGCCGGCCGAA, while the DNA of another organism may be

S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA.

One goal of comparing two parts of DNA is to determine how “similar” two parts are, OR measure of how closely related two organisms are. As we know that similarity is an ambiguous term and can be defined in many different ways. Here we give some ways of defining it with reference to this problem.

An Introduction: Similarity

For example, we can say that two DNA parts are similar if one is a substring of the other. In our case, neither *S*1 nor *S*2 is a substring of the other. This will be discussed in string matching. Alternatively, we could say two parts are similar if changes needed to turn one to other is small. Another way to measure similarity is by finding third part *S*3 in which bases in *S*3 appear in both *S*1, *S*2. Bases must preserve order, may not consecutively. Longer *S*3 we can find, more similar *S*1 and *S*2 are. In above, *S*3 is GTCGTCGGAAGCCGGCCGAA.

What is a Subsequence?

In mathematics, a subsequence of some sequence is a new sequence which is formed from original one by deleting some elements without disturbing the relative positions of the remaining elements.

Examples: < B,C,D,B > is a subsequence of
< A,C,B,D,E,G,C,E,D,B,G > , with corresponding index sequence <3,7,9,10>.
    < D, E, E, B > is also a subsequence of the same < A,C,B,D,E,G,C,E,D,B,G > ,
    with corresponding index sequence <4,5,8,10>.

Longest Common Subsequence

The sequence Z = (B, C, A) is a subsequence of X = (A, B, C, B, D, A, B).
The sequence Z = (B, C, A) is also a subsequence of Y = (B, D, C, A, B, A).
Of course, it is a common subsequence of X and Y. But the above sequence is not a longest common subsequence This is because the sequence Z’ = (B, D, A, B) is a longer subsequence of X = (A, B, C, B, D, A, B) and Y = (B, D, C, A, B, A)

https://studyc.info/

Statement:

In the longest-common-subsequence (LCS) problem, we are given two sequences

$X = <x_1, x_2, \ldots, x_m>$ and

$Y = <y_1, y_2, \ldots, y_n>$

And our objective is to find a maximum-length common subsequence of X and Y.

Note:

This LCS problem can be solved using brute force approach as well but using dynamic programming it will be solved more efficiently.

Brute Force Approach

First we enumerate all the subsequences of $X = <x_1, x_2, \ldots, x_m>$. There will be 2^m such subsequences. Then we check if a subsequence of X is also a subsequence of Y. In this way, we can compute all the common subsequences of X and Y. Certainly, this approach requires exponential time, making it impractical for long sequences.

Note: Because this problem has an optimal sub-structure property, and hence can be solved using approach of dynamic programming

Dynamic Programming Solution

Towards Optimal Substructure of LCS: Prefixes

As we shall see, the natural classes of sub-problems correspond to pairs of "prefixes" of the two input sequences.

To be precise, given a sequence $X = <x_1, x_2, ..., x_m>$, we define the ith prefix of X, for i = 0, 1, ..., m, as $X_i = <x_1, x_2, ..., x_i>$.

Examples:

If X = <A, B, C, B, D, A, B> then

X_4 = <A, B, C, B> and

X_0 is the empty sequence = < >

If $X = (x_1, x_2, \ldots, x_m)$, and $Y = (y_1, y_2, \ldots, y_n)$ be sequences and let us suppose that $Z = (z_1, z_2, \ldots, z_k)$ be a longest common sub-sequence of X and Y

Let, $X_i = (x_1, x_2, \ldots, x_i)$, $Y_j = (y_1, y_2, \ldots, y_j)$ and $Z_l = (z_1, z_2, \ldots, z_l)$ are prefixes of X, Y and Z respectively.

1. if $x_m = y_n$, then $z_k = x_m$ and Z_{k-1} is LCS of X_{m-1}, Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is LCS of X_{m-1} and Y
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is LCS of X and Y_{n-1}

Proof of Theorem

Case 1

On contrary suppose that $x_m = y_n$ but $z_k \neq x_m$,

https://studyc.info/

Then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length k + 1, contradicting the supposition that Z is a LCS of X and Y.
Thus, we must have $z_k = x_m = y_n$.
Now, the prefix Z_{k-1} is a length-(k - 1) common subsequence of X_{m-1} and Y_{n-1}.
Now we wish to show that it is an LCS.
Suppose, there is a common subsequence W of X_{m-1} and Y_{n-1} with length greater than k - 1. Appending $x_m = y_n$ to W gives common subsequence of X and Y whose length is greater than k, a contradiction.

<u>Case 2</u>

If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y.
If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

<u>Case 3</u>

The proof is symmetric to (2)

https://studyc.info/

Lecture 22 Review of Lectures 01-21

https://studyc.info/

Lecture 23 Longest Common Subsequence (Dynamic Algorithm) & Optimal Binary Search Trees

Theorem: Optimal Substructure of an LCS

If $X = (x_1, x_2, \ldots, x_m)$, and $Y = (y_1, y_2, \ldots, y_n)$ be sequences and let us suppose that $Z = (z_1, z_2, \ldots, z_k)$ be a longest common sub-sequence of X and Y

1. if $x_m = y_n$, then $z_k = x_m$ and Z_{k-1} is LCS of X_{m-1}, Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is LCS of X_{m-1} and Y
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is LCS of X and Y_{n-1}

$$c(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ OR } j = 0 \\ c(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c(i-1, j), c(i, j-1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Problem

If X = <A, B, C, B, D, A, B>, and Y = <B, D, C, A, B, A> are two sequences then compute a maximum-length common subsequence of X and Y.

Solution:

Let c(i, j) = length of LCS of X_i and Y_j, now we have to compute c(7, 6).

The recursive mathematical formula computing LCS is given below

$$c(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ OR } j = 0 \\ c(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c(i-1, j), c(i, j-1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

If X = <A, B, C, B, D, A, B>, Y = <B, D, C, A, B, A>

c(1, 1) = max (c(0, 1), c(1, 0)) = max (0, 0) = 0 ; b[1, 1] = ←

c(1, 2) = max (c(0, 2), c(1, 1)) = max (0, 0) = 0 ; b[1, 2] = ←

c(1, 3) = max (c(0, 3), c(1, 2)) = max (0, 0) = 0 ; b[1, 3] = ←

c(1, 4) = c(0, 3) + 1 = 0 + 1 = 1 ; b[1, 4] = ↖

c(1, 5) = max (c(0, 5), c(1, 4)) = max (0, 1) = 1 ; b[1, 5] = ←

c(1, 6) = c(0, 5) + 1 = 0 + 1 = 1 ; b[1, 6] = ↖

c(2, 1) = c(1, 0) + 1 = 0 + 1 = 1 ; b[2, 1] = ↖

c(2, 2) = max (c(1, 2), c(2, 1)) = max (0, 1) = 1 ; b[2, 2] = ←

c(2, 3) = max (c(1, 3), c(2, 2)) = max (0, 1) = 1 ; b[2, 3] = ←

c(2, 4) = max (c(1, 4), c(2, 3)) = max (1, 1) = 1 ; b[2, 4] = ←

c(2, 5) = c(1, 4) + 1 = 1 + 1 = 2 ; b[2, 5] = ↖

c(2, 6) = max (c(1, 6), c(2, 5)) = max (1, 2) = 2 ; b[2, 6] = ←

c(3, 1) = max (c(2, 1), c(3, 0)) = max (1, 0) = 1 b[3, 1] = ←

c(3, 2) = max (c(2, 2), c(3, 1)) = max (1, 1) = 1 ; b[3, 2] = ←

c(3, 3) = c(2, 2) + 1 = 1 + 1 = 2 ; b[3, 3] = ↖

c(3, 4) = max (c(2, 4), c(3, 3)) = max (1, 2) = 2 ; b[3, 4] = ←

c(3, 5) = max (c(2, 5), c(3, 4)) = max (2, 2) = 2 ; b[3, 5] = ←

c(3, 6) = max (c(2, 6), c(3, 5)) = max (2, 2) = 2 ; b[2, 6] = ←

c(4, 1) = c(3, 0) + 1 = 0 + 1 = 1 ; b[4, 1] = ↖

c(4, 2) = max (c(3, 2), c(4, 1)) = max (1, 1) = 1 ; b[4, 2] = ←

c(4, 3) = max (c(3, 3), c(4, 2)) = max (2, 1) = 2 ; b[4, 3] = ←

c(4, 4) = max (c(3, 4), c(4, 3)) = max (2, 2) = 2 ; b[4, 4] = ←

c(4, 5) = c(3, 4) + 1 = 2 + 1 = 3 ; b[4, 5] = ↖

c(4, 6) = max (c(3, 6), c(4, 5)) = max (2, 3) = 3 ; b[4, 6] = ←

c(5, 1) = max (c(4, 1), c(5, 0)) = max (1, 0) = 1 ; b[5, 1] = ↑

c(5, 2) = c(4, 1) + 1 = 1 + 1 = 2 ; b[5, 2] = ↖

c(5, 3) = max (c(4, 3), c(5, 2)) = max (2, 2) = 2 ; b[5, 3] = ←

c(5, 4) = max (c(4, 4), c(5, 3)) = max (2, 2) = 2 ; b[5, 4] = ←

c(5, 5) = max (c(4, 5), c(5, 4)) = max (3, 2) = 3 ; b[5, 5] = ↑

c(5, 6) = max (c(4, 6), c(5, 5)) = max (3, 3) = 3 ; b[5, 6] = ←

c(6, 1) = max (c(5, 1), c(6, 0)) = max (1, 0) = 1 ; b[6, 1] = ↑

c(6, 2) = max (c(5, 2), c(6, 1)) = max (2, 1) = 2 ; b[6, 1] = ↑

c(6, 3) = max (c(5, 3), c(6, 2)) = max (2, 2) = 2 ; b[6, 3] = ←

c(6, 4) = c(5, 3) + 1 = 2 + 1 = 3 ; b[6, 4] = ↖

c(6, 5) = max (c(5, 5), c(6, 4)) = max (2, 3) = 3 ; b[6, 5] = ←

https://studyc.info/

c(6, 6) = c(5, 5) + 1 = 3 + 1 = 4 ; b[6, 6] = ↖

c(7, 1) = c(6, 0) + 1 = 0 + 1 = 1 ; b[7, 1] = ↖

c(7, 2) = max (c(6, 2), c(7, 1)) = max (2, 1) = 2 ; b[7, 2] = ↑

c(7, 3) = max (c(6, 3), c(7, 2)) = max (2, 2) = 2 ; b[7, 3] = ←

c(7, 4) = max (c(6, 4), c(7, 3)) = max (3, 2) = 3 ; b[7, 4] = ↑

c(7, 5) = c(6, 4) + 1 = 3 + 1 = 4 ; b[7, 5] = ↖

c(7, 6) = max (c(6, 6), c(7, 5)) = max (4, 4) = 4 ; b[7, 6] = ←

Results:

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	•	•	•	•	•	•	•
1	A	•	←	←	←	↖	←	↖
2	B	•	↖	←	←	←	↖	←
3	C	•	↑	←	↖	←	←	←
4	B	•	↖	←	↑	←	↖	←
5	D	•	↑	↖	←	←	↑	←
6	A	•	↑	↑	←	↖	←	↖
7	B	•	↖	↑	←	↑	↖	←

Computable Tables:

https://studyc.info/

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	•	•	•	•	•	•	•
1	A	•	←	←	←	↖	←	↖
2	B	•	↖	←	←	←	↖	←
3	C	•	↑	←	↖	←	←	←
4	B	•	↖	←	↑	←	↖	←
5	D	•	↑	↖	←	←	↑	←
6	A	•	↑	↑	←	↖	←	↖
7	B	•	↖	↑	←	↑	↖	←

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

	j	0	1	2	3	4	5	6
i		y_j	**B**	**D**	C	**A**	**B**	A
0	x_i	•	•	•	•	•	•	•
1	A	•	←	←	←	↖	←	↖
2	B	•	↖	←	←	←	↖	←
3	C	•	↑	←	↖	←	←	←
4	**B**	•	↖	←	↑	←	↖	←
5	**D**	•	↑	↖	←	←	↑	←
6	**A**	•	↑	↑	←	↖	←	↖
7	**B**	•	↖	↑	←	↑	↖	←

https://studyc.info/

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Table size: O(n.m)
Every entry takes O(1) time to compute.
The algorithm takes O(n.m) time and space.
The space complexity can be reduced to
$2 \cdot \min(m, n) + O(1)$.

Longest Common Subsequence Algorithm

c[i, j] = c(i-1, j-1) + 1 if xi = yj;
c[i, j] = max(c(i-1, j), c(i, j-1)) if xi ≠ yj;
c[i, j] = 0 if (i = 0) or (j = 0).

```
function LCS(X, Y)
1  m ←   length [X]
2  n ←   length [Y]
3  for i ← 1 to m
4     do c[i, 0] ← 0;
5  for j ← 1 to n
6     do c[0, j] ← 0;
7  for i ← 1 to m
8     do for j ← 1 to n
9        do if (xi = = yj)
10          then c[i, j] ← c[i-1, j-1] + 1
11                b[i, j] ← " ↖ "
12          else if c[i-1, j] ≥ c[i, j-1]
13                then c[i, j] ← c[i-1, j]
14                     b[i, j] ← "↑"
15                else  c[i, j] ← c[i, j-1]
16                     b[i, j] ← "←"
17 Return c and b;
```

Construction of Longest Common Subsequence

c[i, j] = c(i-1, j-1) if xi = yj;
c[i, j] = min(c(i-1, j), c(i, j-1)) if xi ≠ yj;
c[i, j] = 0 if (i = 0) or (j = 0).

procedure PrintLCS(b, X, i, j)

https://studyc.info/

```
1.  if (i == 0) or (j == 0)
2.     then return
3.  if b[i,  j] == " ↖"
4.     then PrintLCS(b, X, i-1, j-1)
5.            Print xi
6.  else if b[i,  j] ← "↑"
7.     then PrintLCS(b, X, i-1, j)
8.  else PrintLCS(b, X, i, j-1)
```

Relationship with shortest common supper-sequence

Shortest common super-sequence problem is closely related to longest common subsequence problem

Shortest common super-sequence

Given two sequences: $X = < x_1,...,x_m >$ and $Y = < y_1,...,y_n >$

A sequence $U = < u_1,...,u_k >$ is a common super-sequence of X and Y if U is a super-sequence of both X and Y. The shortest common supersequence (scs) is a common supersequence of minimal length.

Problem Statement

The two sequences X and Y are given and task is to find a shortest possible common supersequence. Shortest common supersequence is not unique. Easy to make SCS from LCS for 2 input sequences.

Example:

- X[1..m] = abcbdab
  Y[1..n] = bdcaba
  LCS = Z[1..r] = bcba
- Insert non-lcs symbols preserving order, we get
  SCS = U[1..t] = abdcabdab.

Optimal Binary Search Trees

Binary search tree (BST) is a binary data structure which has the following properties:

- Each node has a value.
- An order is defined on these values.
- Left sub-tree of node contains values less than node value
- Right sub-tree of a node contains only values greater than or equal to the node's value.

Optimal Binary Search Trees

Example: A translator from English to, say, Urdu.

https://studyc.info/

- Use a binary search tree to store all the words in our dictionary, together with their translations.
- The word “the” is much more likely to be looked up than the word “ring”
- So we would like to make the search time for the word “the” very short, possibly at the expense of increasing the search time for the word “ring.”

Problem Statement: We are given a probability distribution that determines, for every key in the tree, the likelihood that we search for this key. The objective is to minimize the expected search time of the tree.

We need what is known as an optimal binary search tree.

Formally, we are given a sequence $K = \langle k_1, k_2, ..., k_n \rangle$ of n distinct keys in sorted order i.e. $k_1 < k_2 < \cdots < k_n$, and we wish to build a binary search tree from these keys. For each k_i, we have a probability p_i that search is for k_i. Some searches may not be successful, so we have n + 1 “dummy keys” $d_0, d_1, d_2, ..., d_n$ representing values not in K.

In particular

d_0 = represents all values less than k_1,

d_n = represents all values greater than k_n, and

d_i = represents all values between k_i and k_{i+1},

$$\forall\ i = 1, 2, ..., n-1$$

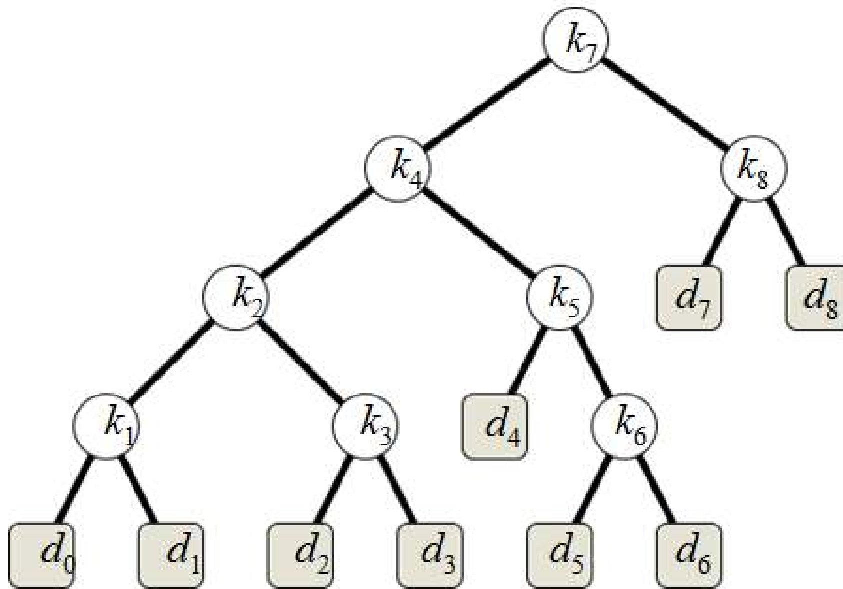
For each dummy key d_i, we have a probability q_i that a search will correspond to d_i.

Each k_i is an internal node, each dummy key d_i is a leaf.

Every search is either successful (finding some key k_i) or failure (finding some dummy key d_i), and so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

https://studyc.info/



Total Cost: Optimal Binary Search Trees

$$\text{Total cost} = \sum_{i=1}^{n} p_i(\text{depth}((k_i)+1) + \sum_{i=0}^{n} q_i(\text{depth}((d_i)+1)$$

$$= \sum_{i=1}^{n} p_i.\text{depth}(k_i) + \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i\text{depth}(d_i) + \sum_{i=0}^{n} q_i$$

Since we know that: $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$

Hence Total Cost $= \sum_{i=1}^{n} p_i.\text{depth}(k_i) + \sum_{i=0}^{n} q_i\text{depth}(d_i) + 1$

Example:

Let

p_i = probability of searching k_1

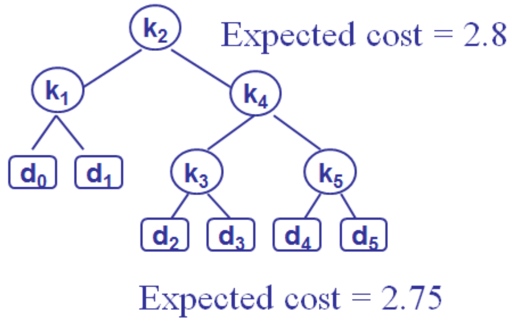
q_i = probability representing d_i

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

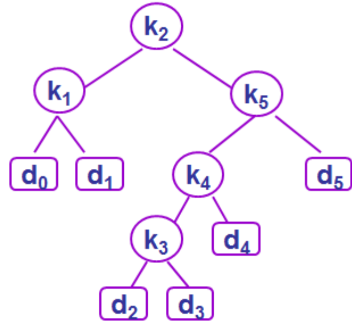
Define cost of a search is as number of nodes examined in a search

Cost of a key = depth of the key + 1

https://studyc.info/

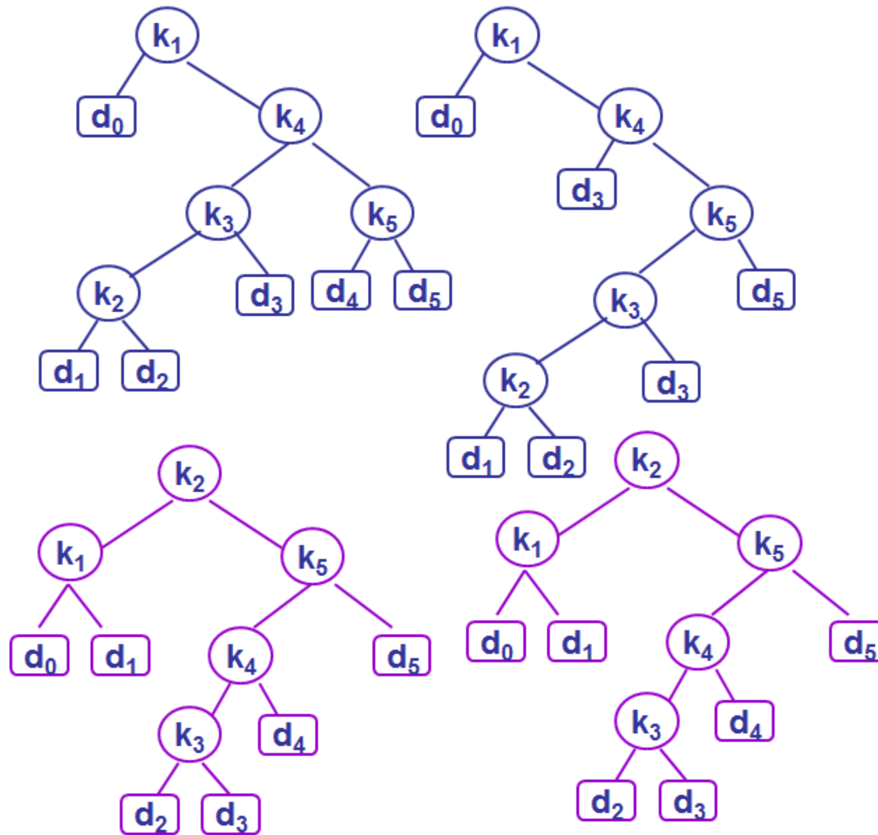


Expected cost = 2.75

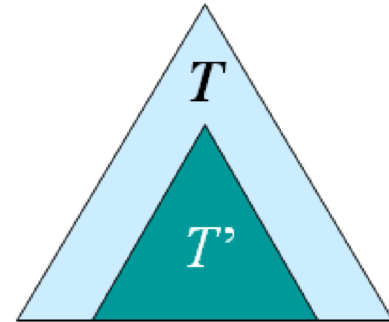


Brute Force Solution

Total number of binary trees will be exponential as in case of chain matrix problem. Brute force approach is not economical.



https://studyc.info/



Constructing Dynamic Programming

Optimal Substructure

Observation: Any subtree of a BST contains keys range k_i, ..., k_j for some $1 \le i \le j \le n$.

Lemma

If T is an optimal BST and T contains subtree T' with keys k_i, ... ,k_j , then T' must be an optimal BST for keys k_i, ..., k_j.

Proof**:**

Cut and paste method.

Limitations of Dynamic Programming

- Dynamic programming can be applied to any problem that observes the principle of optimality.
- Generally, it means that partial solutions can be optimally extended with regard to the state after the partial solution instead of the partial solution itself.
- The biggest limitation using dynamic programming is number of partial solutions we must keep track of
- For all examples we have seen, partial solutions can be described by stopping places in the input.
- This is because combinatorial objects e.g. strings, numerical sequences, and polygons etc., all have an implicit order defined upon their elements.
- This order cannot be changed without completely changing the original problem.
- Once order fixed, there are relatively few possible stopping places, and we get an efficient algorithms.
- If objects are not firmly ordered then we have an exponential number of possible partial solutions
- And we get an infeasible amount of memory resulting an infeasible solution.

https://studyc.info/

Lecture 24 Optimal Binary Search Trees (Constructing Dynamic Programming)

Construction: Optimal Substructure

- One of the keys in $k_i, \ldots, k_j$, say k_r, $i \le r \le j$, must be the root of an optimal subtree for these keys.
- Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
- Right subtree of k_r contains $k_{r+1}, \ldots, k_j$.
- To find an optimal BST:
  - Examine all candidate roots k_r, for $i \le r \le j$
  - Determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$
- Find optimal BST for $k_i, \ldots, k_j$, where $i \ge 1, j \le n, j \ge i-1$
- When $j = i-1$, the tree is empty.
- Define $e[i, j]$ = expected search cost of optimal BST for $k_i, \ldots, k_j$.
- If $j = i-1$, then $e[i, j] = q_{i-1}$.
- If $j \ge i$,
  - Select a root k_r, for some $i \le r \le j$.
  - Recursively make an optimal BSTs
    - for $k_i, \ldots, k_{r-1}$ as the left subtree, and
    - for $k_{r+1}, \ldots, k_j$ as the right subtree.

Lemma: Prove that when OPT tree becomes a sub-tree of a node then expected search cost increases by $w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$

Proof:

$Total\ cost$ when a tree becomes subtree

$$= \sum_{l=i}^{j} p_l (depth((k_l) + 1 + 1) + \sum_{l=i-1}^{j} q_l (depth((d_l) + 1 + 1)$$

$$= \sum_{l=i}^{j} p_l.(depth(k_l) + 1) + \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l (depth(d_l) + 1) + \sum_{l=i-1}^{j} q_l$$

$$= \sum_{l=i}^{j} p_l.(depth(k_l) + 1) + \sum_{l=i-1}^{j} q_l (depth(d_l) + 1) + \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

$$= total \text{ cost when tree was not subtree } + \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

- When OPT subtree becomes a subtree of a node:
  - Depth of every node in OPT subtree goes up by 1.
  - Expected search cost increases by $w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$
  - If k_r is the root of an optimal BST for $k_i, \ldots, k_j$
  - $e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$
  - $= e[i, r-1] + e[r+1, j] + w(i, j)$.
  - But, we don't know k_r. Hence,

https://studyc.info/

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j \end{cases}$$

Algorithm: Optimal Binary Search

```
OPTIMAL-BST(p, q, n)
for i ← 1 to n + 1
   do e[i, i– 1] ← qi-1
        w[i, i– 1] ← qi-1.
for l ← 1 to n
   do for i ← 1 to n–l + 1
        do j ←i + l–1
           e[i, j ]←∞
           w[i, j ] ← w[i, j–1] + pj + qj
           for r ←i to j
               do t ← e[i, r–1] + e[r + 1, j ] + w[i, j ]
                   if t < e[i, j ]
                       then e[i, j ] ← t
                                root[i, j ] ←r
return e and root
```

Greedy Algorithms

Why Greedy Algorithm?

The algorithms we have studied in dynamic programming are relatively inefficient, for example

- Cost of 0-1 knapsack problem: O(nW)
- Cost of matrix chain multiplication: $O(n^3)$
- Cost in longest common subsequence: O(mn)
- Optimal binary search trees: $O(n^3)$

This is because

- We have many choices computing optimal solution.
- We check all of them in dynamic programming.
- We must think which choice is the best or
- At least restrict the choices we have to try.

What are Greedy Algorithms?

- In greedy algorithms, we do the same thing
- Mostly optimization algorithms go through a sequence of steps, with a set of choices at each step.
- In dynamic programming best choices is ignored
- Sometimes a simpler and efficient algorithm required.
- Greedy algorithms make best choice at a moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms do not always yield an optimal solutions, but mostly they do
- They tend to be easier to implement.

https://studyc.info/

Where Greedy Algorithms do not work?

- Greedy choice:
  - We make the choice that looks best at a moment.
  - Every time make a choice, greedily maximize value
- An example where this does not work
  - Find longest monotonically increasing subsequence
  - Given sequence <3 4 5 17 7 8 9>
  - Longest such subsequence is <3 4 5 7 8 9>.
  - The greedy choice after choosing <3 4 5> is to choose 17, which is an unwanted element, results in the sequence <3 4 5 17> which is suboptimal.

Activity Selection Problem

Some Definitions:

Closed Interval = $[a, b] = \{x \in R \mid a \leq x \leq b\}$

Open Interval = $(a, b) = \{x \in R \mid a < x < b\}$

Left Semi Open = $(a, b] = \{x \in R \mid a < x \leq b\}$

Right Semi Open = $[a, b) = \{x \in R \mid a \leq x < b\}$

Activity Selection Problem

The problem involves scheduling of several competing activities that require exclusive use of common resource

Problem Statement

- The problem can be stated as, suppose we have a set: $S = \{a_1, a_2, ..., a_n\}$ of n proposed activities.
  - Each activity wishes to use a resource which can be used by only one activity at a time.
  - Each activity a_i has starting time s_i, finishing time f_i

    where, $0 \leq s_i < f_i < \infty$
- Objective in activity-selection problem is to select a maximum-size subset of mutually compatible activities.

Steps in Designing Activity Selection Algorithm

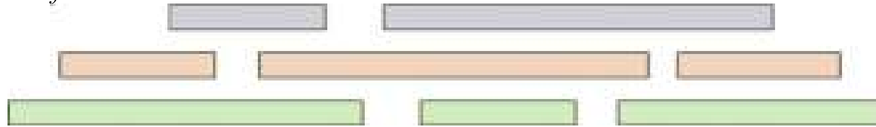
Steps to solve the problem

- We will formulate the dynamic-programming solution in which
  - Combine optimal solutions to two subproblems to form an optimal solution to original problem
  - Check several choices when determining which subproblems to use in an optimal solution
- Then needed to make a greedy choice in which
  - One of the subproblems guaranteed empty, so that only one nonempty subproblem remains
- Then a recursive greedy algorithm is developed and converted to an iterative one

https://studyc.info/

Application: Scheduling Problem

- A classroom can be used for one class at a time.
- There are n classes that want to use the classroom.
- Every class has a corresponding time interval $I_j = [s_j, f_j)$ during which the room would be needed for this class.
- Our goal is to choose a maximal number of classes that can be scheduled to use the classroom without two classes ever using the classroom at the same time.
- Assume that the classes are sorted according to increasing finish times; that is, $f_1 < f_2 < \ldots < f_n$.



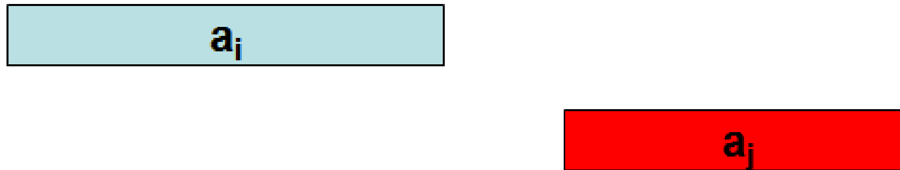
Designing Activity Selection Problem

Compatible Activity

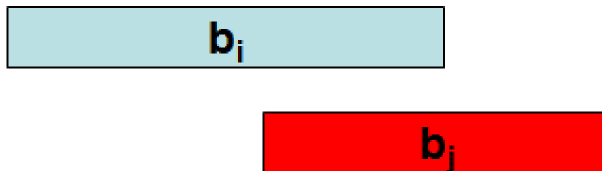
If a selected activity a_i *is required to* take place during the half-open time interval $[s_i, f_i)$. And activity a_j *is required to* take place during the half-open time interval $[s_j, f_j)$. Then the activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap i.e

$s_i \geq f_j$ or $s_j \geq f_i$

Compatible Activities



Not Compatible Activities



Compatible not Maximal

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

https://studyc.info/

A set S of activities sorted in increasing order of finish time

- The subset consisting of mutually compatible activities are
  - $\{a_3, a_9, a_{11}\}$ but it is not a maximal subset,
  - $\{a_1, a_4, a_8, a_{11}\}$ is larger.
  - $\{a_2, a_4, a_9, a_{11}\}$ is another largest subset.

Optimal Substructure of Activity Selection Problem

The first step is to find optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.

Let us start by defining sets

$$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$$

S_{ij} is the subset of activities in S that can start after activity a_i finishes and finish before activity a_j starts.

The fictitious activities a_0 and a_{n+1} are added and adopt the conventions that $f_0 = 0$ and $s_{n+1} = \infty$. Then $S = S_{0.n+1}$, and the ranges for i and j are given by

$$0 \le i, j \le n + 1$$

Let us assume that A_{ij} *is a solution to* S_{ij}

Let us assume that the activities are sorted in monotonically increasing order of finish times of the activities:

$$f_0 \le f_1 \le f_2 \le \dots \le f_n < f_{n+1}$$

Assuming that we have sorted set of activities, our space of subproblems is

- to select a maximum-size subset of mutually compatible activities from S_{ij} S_{ij}, for $0 \le i < j \le n + 1$,
- knowing that all other S_{ij} are empty.

Decomposition: Optimal Substructure of Problem

Suppose that a solution to S_{ij} *is non-empty and* includes some activity a_k, so that $f_i \le s_k < f_k \le s_j$

After choosing activity a_k, *it* decomposes S_{ij} into two subproblems, S_{ik} and S_{kj}

S_{ik} = activities that start after a_i finishes and finish before a_k starts and

S_{kj} = activities that start after a_k finishes and finish before a_j starts,

Each of S_{ik} and S_{kj} are subset of the activities in S_{ij}

Solution to $S_{i.j}$: Optimal Substructure of Problem

https://studyc.info/

Our solution to S_{ij} is the union of the solutions to S_{ik} and S_{kj}, along with the activity a_k. Thus, the number of activities in our solution to S_{ij} is the size of our solution to S_{ik}, plus the size of our solution to S_{kj} , plus a_k.

Solution(S_{ij}) = Solution(S_{ik}) $\cup$ Solution(S_{kj}) $\cup \{a_k\}$

$A_{ij} = A_{ik} \cup A_{kj} \cup \{a_k\}$

Suppose we now that an optimal solution A_{ij} *includes* activity a_k, then the solutions A_{ik} and A_{kj} used within this optimal solution must be optimal.

Why A_{ik} and A_{kj} are Optimal?

Proof: Why A_{ik} and A_{kj} are Optimal?

If we had a solution A'_{ik} to S_{ik} that included more activities than A_{ik} , we could cut out A_{ik} from A_{ij} and paste A'_{ik} in A_{ij} , thus producing another solution A'_{ij} to S_{ij} with more activities than A_{ij} .

Because we assumed that A_{ij} is an optimal solution, we have derived a contradiction.

Similarly, if we had a solution A'_{kj} to S_{kj} with more activities than A_{kj} , we could replace A_{kj} by A'_{kj} to produce solution to S_{ij} with more activities than A_{ij}

Further Decomposition to Find $S_{0, n+1}$

Now we can build a maximum-size subset of mutually compatible activities in S_{ij}

- by splitting the problem into two subproblems, mutually compatible activities in S_{ik} and S_{kj}
- finding maximum-size subsets A_{ik} and A_{kj} of these activities for these subproblems and then
- forming maximum-size subset A_{ij} as

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

Optimal solution to entire problem is: $A_{0,n+1}$

A Recursive Solution

Let c[i, j] = number of activities in a maximum-size subset of mutually compatible activities in S_{ij}

We have c[i, j] = 0 whenever $S_{ij} = Ø$;

In particular we have, c[i, j] = 0 for i ≥ j.

Since $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

Therefore the recurrence relation for the problem is

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

Since value of k varies between i + 1, ..., j – 1, and hence there are (j - 1) - (i - 1) + 1 possible values of k, i.e., j - 1 – i - 1 + 1 = j – i - 1

Since maximum-size subset of S_{ij} must use one of these values for *k*, we check them all to find the best.
Thus, our full recursive definition of *c*[*i, j*] becomes

$$c[i,j]=\begin{cases}0 & if\ S_{ij}=\varphi\\ \max\limits_{i<k<j}\{c[i,k]+c[k,j]+1\} & if\ S_{ij}\neq\varphi\end{cases}$$

Lecture 25 Greedy Algorithms

<u>Statement:</u> Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min\ \{f_k : a_k \in S_{ij}\}$, then

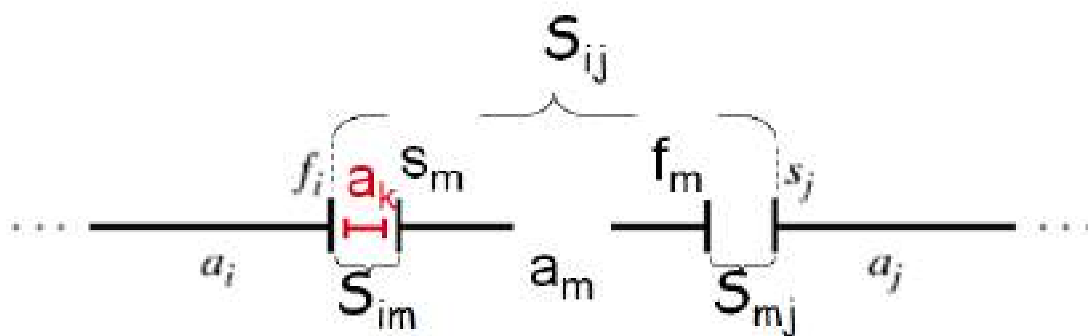
1. Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

<u>Note:</u> After proving these properties, it is guaranteed that the greedy solution to this problem does exist.

<u>Theorem:</u>

<u>Proof (Part B)</u>

First we prove second part because it is bit simpler
Suppose that S_{im} is nonempty
It means there is some activity a_k such that: $f_i \leq s_k < f_k \leq s_m < f_m. \Rightarrow f_k < f_m.$
Then a_k is also in S_{ij} and it has an earlier finish time than a_m, which contradicts our choice of a_m. Hence S_{im} is empty, proved



<u>Part A</u>

To prove first part, suppose that A_{ij} is a maximum-size subset of mutually compatible activities of S_{ij},

Order A_{ij} monotonic increasing order of finish time

Let a_k be the first activity in A_{ij}.

https://studyc.info/

Case 1

If $a_k = a_m$, then we are done, since we have shown that a_m is used in some maximal subset of mutually compatible activities of S_{ij}.

Case 2

If $a_k \neq a_m$, then we construct the subset

$$A'_{ij} = A_{ij} \setminus \{a_k\} \cup \{a_m\}$$

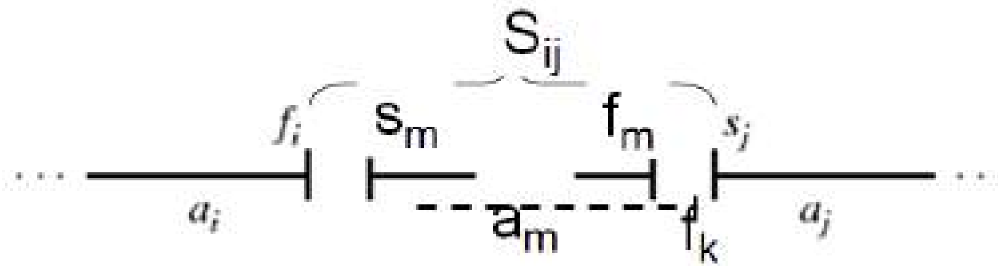
Since activities in A_{ij} are disjoint, so is true for A'_{ij}.

As a_k is first activity in A_{ij} to finish, and $f_m \leq f_k$.

Noting that A'_{ij} has same number of activities as A_{ij}

We see that A'_{ij} is a maximal subset of mutually compatible activities of S_{ij} that includes a_m.

Hence proves the theorem.



Why is this Theorem Useful?

	Dynamic programming	Using the theorem
Number of subproblems in the optimal solution	2 subproblems: S_{ik}, S_{kj}	1 subproblem: S_{mj} $S_{im} = \varnothing$
Number of choices to consider	$j - i - 1$ choices	1 choice: the activity with the earliest finish time in S_{ij}

A Recursive Greedy Algorithm

Recursive-Activity-Selector (*s, f, i, j*)

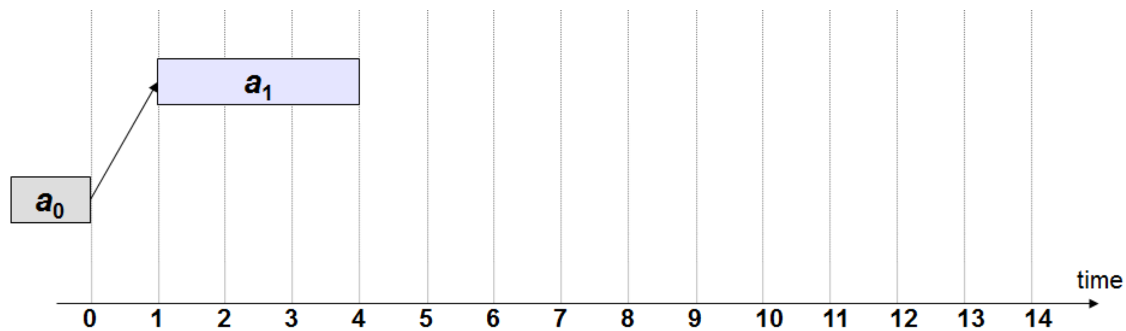
https://studyc.info/

1 $m \leftarrow i + 1$
2 **while** $m < j$ and $s_m < f_i$ Find the first activity in S_{ij}.
3 **do** $m \leftarrow m + 1$
4 **if** $m < j$
5 **then**
return $\{a_m\}$ Recursive-Activity-Selector (s, f, m, j)
6 **else return** Ø

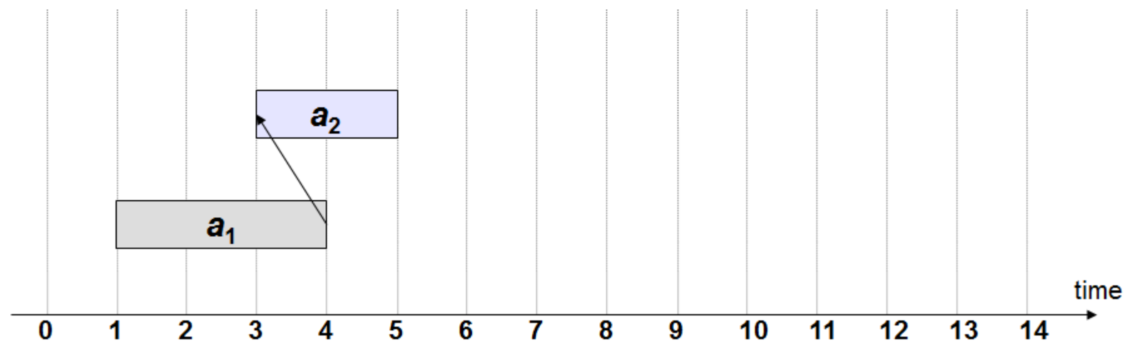
<u>Example: A Recursive Greedy Algorithm</u>

i	0	1	2	3	4	5	6	7	8	9	10	11
s_i	-	1	3	0	5	3	5	6	8	8	2	12
f_i	0	4	5	6	7	8	9	10	11	12	13	14

For the Recursive Greedy Algorithm, the set *S* of activities is sorted in increasing order of finish time



$i = 0$,
$j = n + 1 = 12$
$m \leftarrow i + 1 \leftarrow 0 + 1 = 1$
$m < j$ (1 < 12) and $s_1 < f_0$ (But 1>0)
if $m < j$ (1 < 12)
return $\{a_1\}$ □ Recursive-Activity-Selector ($s, f,$ 1,12)

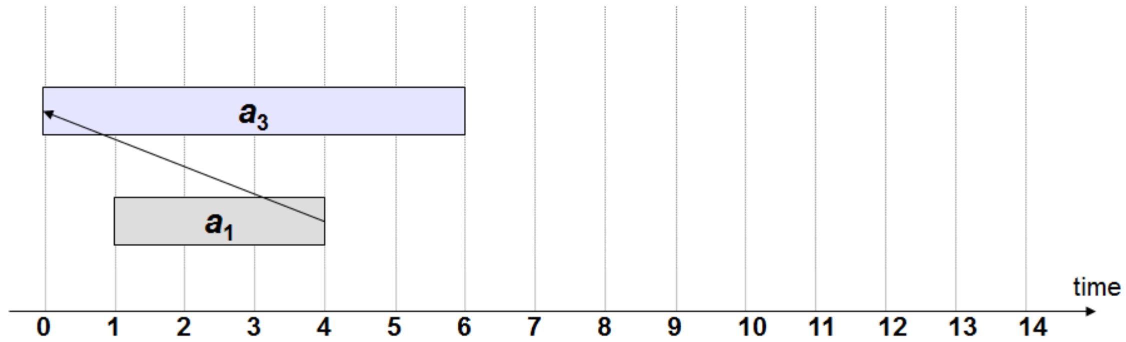


$i = 1$,

$m \leftarrow i + 1 \leftarrow 1 + 1 = 2$

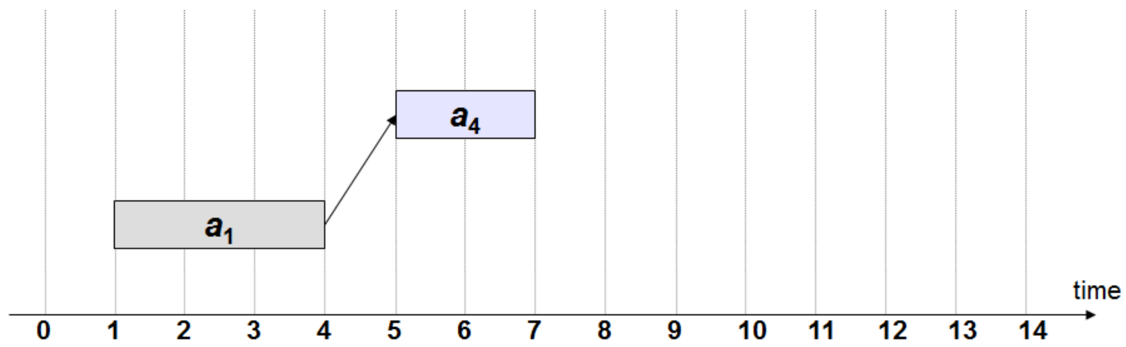
$m < j$ (2 < 12) and $s_2 < f_1$ (3 < 4)

$m \leftarrow m + 1 \leftarrow 2 + 1 = 3$



$m < j$ (3 < 12) and $s_3 < f_1$ (0 < 4)

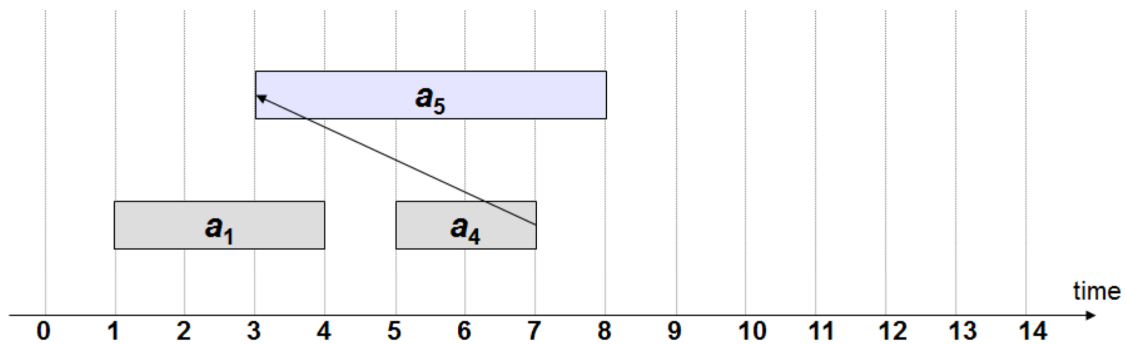
$m \leftarrow m + 1 \leftarrow 3 + 1 = 4$



$m < j$ (4 < 12) and $s_4 < f_1$ (But 5 > 4)

if $m < j$ (4 < 12)

return $\{a_4\}$ □ Recursive-Activity-Selector($s, f,$ 4,12)



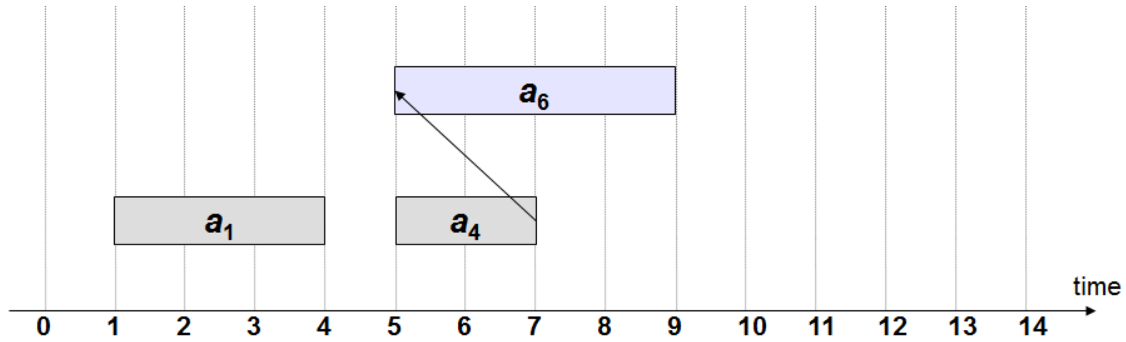
$i = 4$,

https://studyc.info/

$m \leftarrow i + 1 \leftarrow 4 + 1 = 5$

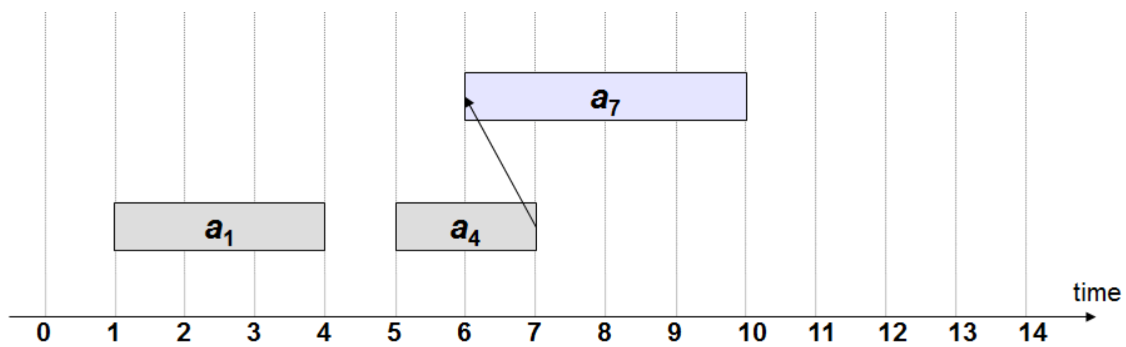
$m < j$ (5 < 12) and $s_5 < f_4$ (3 < 7)

$m \leftarrow m + 1 \leftarrow 5 + 1 = 6$



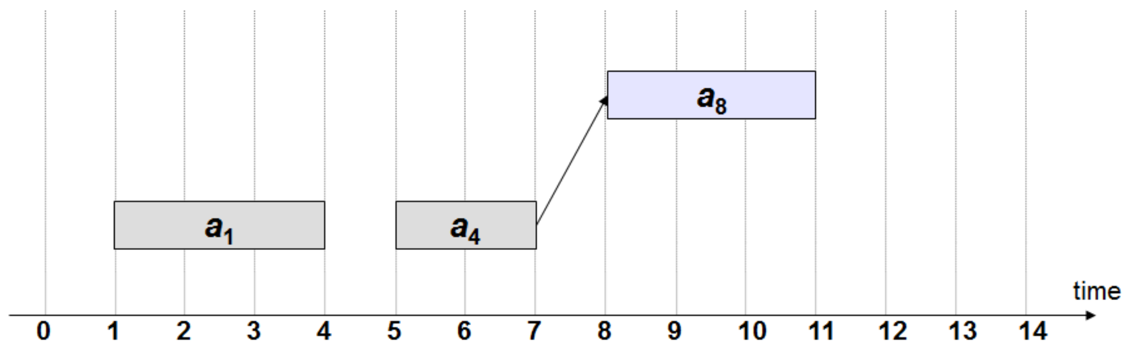
$m < j$ (6 < 12) and $s_6 < f_4$ (5 < 7)

$m \leftarrow m + 1 \leftarrow 6 + 1 = 7$



$m < j$ (7 < 12) and $s_7 < f_4$ (6 < 7)

$m \leftarrow m + 1 \leftarrow 7 + 1 = 8$

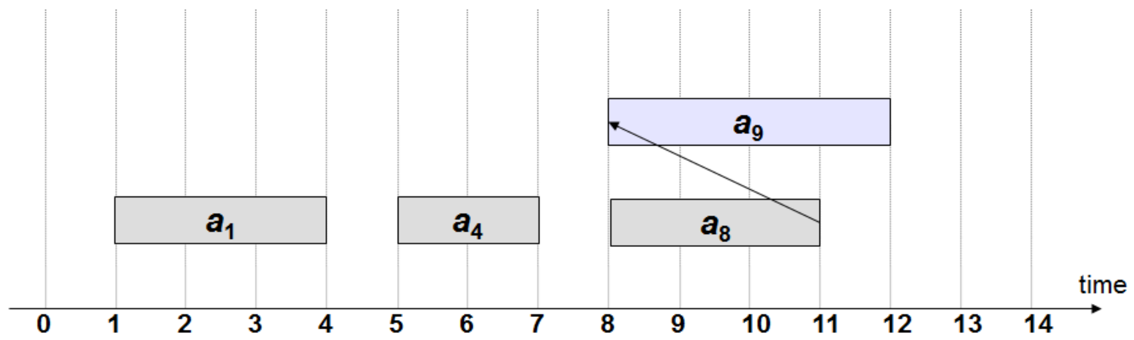


$m < j$ (8 < 12) and $s_8 < f_l$ (But 8 > 7)

if $m < j$ (8 < 12)

https://studyc.info/

return $\{a_8\}$ □ Recursive-Activity-Selector (s, f, 8,12)

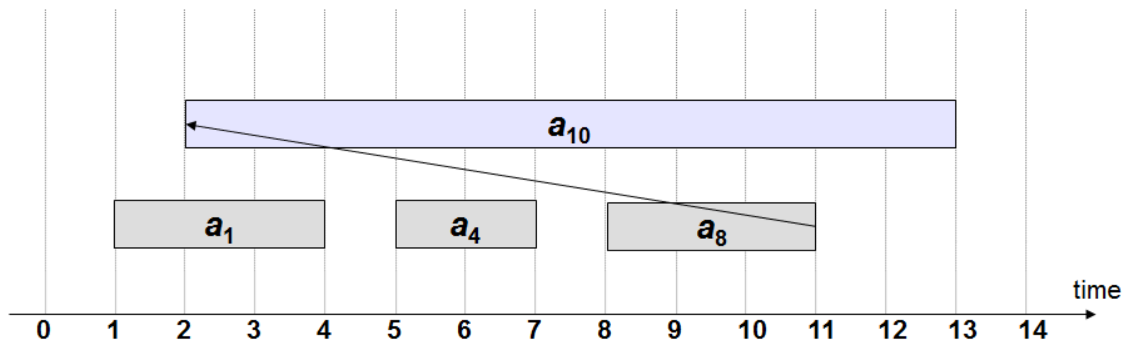


$i = 8$,

$m \leftarrow i + 1 \leftarrow 8 + 1 = 9$

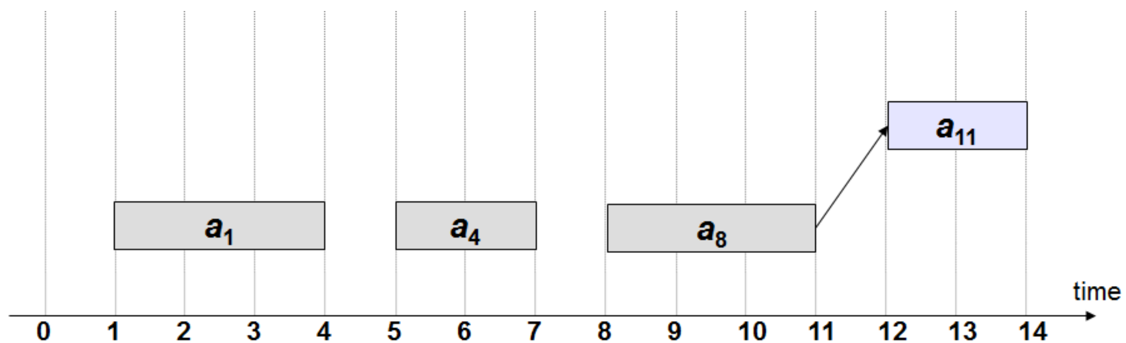
$m < j$ (9 < 12) and $s_9 < f_8$ (8 < 11)

$m \leftarrow m + 1 \leftarrow 9 + 1 = 10$



$m < j$ (10 < 12) and $s_{10} < f_8$ (2 < 11)

$m \leftarrow m + 1 \leftarrow 10 + 1 = 11$

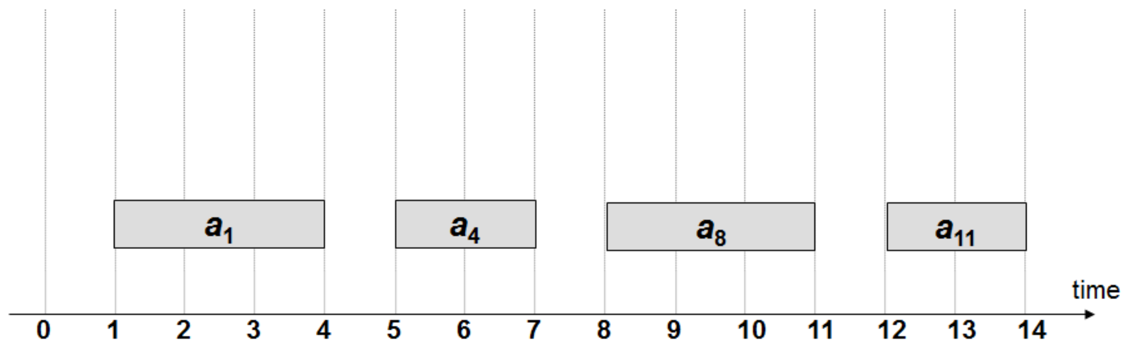


$m < j$ (11 < 12) and $s_{11} < f_8$ (But 12 > 11)

if $m < j$ (11 < 12)

https://studyc.info/

return $\{a_{11}\}$ □ Recursive-Activity-Selector ($s, f, 11,$12)



$i = 11$,

$m \leftarrow i + 1 \leftarrow 11 + 1 = 12$

$m < j$ (But 12 = 12)

An Iterative Greedy Algorithm

Iterative-Activity-Selector (s, f)
1 $n \leftarrow length[s]$
2 $A \leftarrow \{a_1\}$
3 $i \leftarrow 1$
4 **for** m $\leftarrow$ 2 to n
5 **do if** $s_m \geq f_i$
6 **then** A $\leftarrow A \cup \{a_m\}$
7 $i \leftarrow m$
8 **return** A

Summary

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.
- For each decision point in the algorithm, the choice that seems best at the moment is chosen at that time.
- This strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does.
- Now we give a sequence of steps designing an optimal solution of using greedy approach

Summary: Steps Designing Greedy Algorithms

We went through the following steps in the above problem:
1. Determine the suboptimal structure of the problem.
2. Develop a recursive solution.
3. Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.

https://studyc.info/

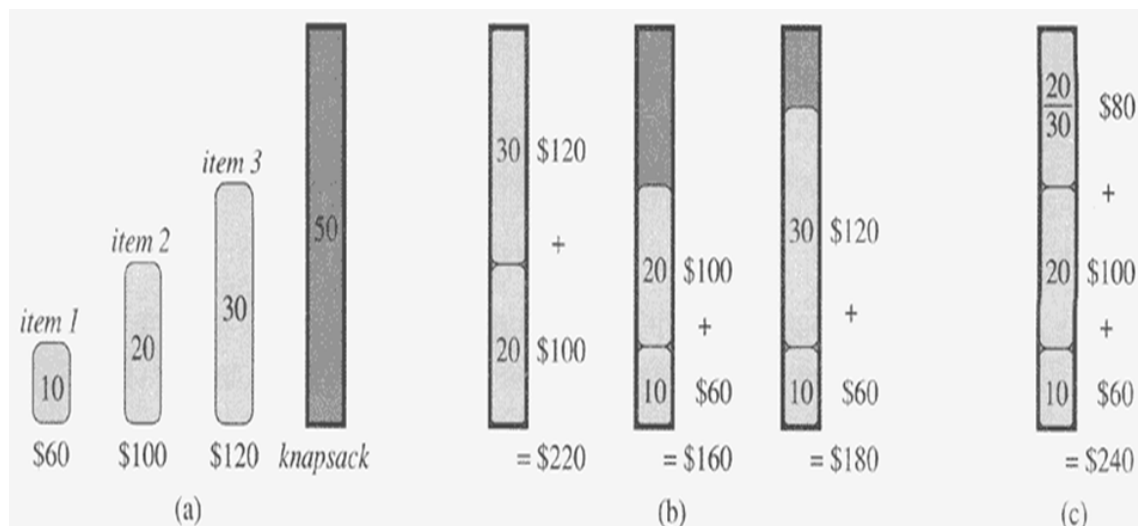
4. Show that all but one of the sub-problems induced by having made the greedy choice are empty.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert this recursive algorithm to an iterative one.

Checks in Designing Greedy Algorithms

- In the beneath every greedy algorithm, there is almost always a dynamic programming solution.
  How can one tell if a greedy algorithm will solve a particular optimization problem?
- There is no way in general, but there are two key ingredients
  - greedy choice property and
  - optimal sub-structure
- If we can demonstrate that the problem has these properties, then we are well on the way to developing a greedy algorithm for it.

The Knapsack Problem

- **The 0-1 Knapsack Problem**
  - A thief robbing a store finds n items: i-th item worth v_i and weight w_i, where v_i and w_i integers
  - The thief can only carry weight W in his knapsack
  - Items must be taken entirely or left behind
  - Which items should the thief take to maximize the value of his load?
- **The Fractional Knapsack Problem**
  - Similar to 0-1 can be solved by greedy approach
  - In this case, the thief can take fractions of items.



(Figure) The greedy strategy does not work for the 0-1 knapsack problem. (a) THe thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though

item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Developing Algorithm: Fractional Knapsack

- Pick the item with the maximum value per pound v_i/w_i
- If the supply of that element is exhausted and the thief can carry more then take as much as possible from the item with the next greatest value per pound
- Continue this process till knapsack is filled
- It is good to order items based on their value per pound

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Algorithm: Fractional Knapsack Problem

Fractional-Knapsack (W, v[n], w[n])

1. While w > 0 and as long as there are items remaining
2. pick item with maximum v_i/w_i
3. $x_i \leftarrow \min(1, w/w_i)$
4. remove item i from list
5. $w \leftarrow w - x_i w_i$

w the amount of space remaining in the knapsack (w = W)

Running time: $\Theta(n)$ if items already ordered; else $\Theta(n \lg n)$

Making Change

Someone comes to your store and makes a purchase of 98.67. He/she gives you 100. You want to give back change using the least number of coins.

- **INPUT:** The values of coins: $C_1, C_2, \ldots, C_k$, and an integer N. Assume that some coin has value 1.
- **GOAL:** To find a multi-set of coins S whose sum is N where the total number of coins is minimized.
- A greedy approach is to add the highest value coin possible.

Making Change (Greedy Algorithm)

Greedy algorithm (C, N)

1. sort coins so $C1 \geq C2 \geq \ldots \geq Ck$
2. $S = \Phi$;
3. Change = 0
4. i = 1 \\Check for next coin
5. **while** Change $\neq$ N do \\all most valuable coins
6. **if** Change + Ci $\leq$ N **then**
7. Change = Change + Ci
8. $S = S \cup \{Ci\}$
9. **else** *i = i+1*

In Pakistan, our currency notes are

https://studyc.info/

$C_1 = 5000, C_2 = 1000, C_3 = 500, C_4 = 100, C_5 = 50, C_6 = 20, C_7 = 10$

Applying above greedy algorithm to N = 13,660, we get

$S = \{C_1, C_1, C_2, C_2, C_2, C_3, C_4, C_5, C_7\}$

Does this algorithm always find an optimal solution? For Pakistani currency.

It does but does not hold always

Dynamic Programming vs. Greedy Algorithms

- Dynamic programming
  - We make a choice at each step
  - The choice depends on solutions to subproblems
  - Bottom up solution, smaller to larger subproblems
- Greedy algorithm
  - Make the greedy choice and THEN
  - Solve subproblem arising after the choice is made
  - The choice we make may depend on previous choices, but not on solutions to subproblems
  - Top down solution, problems decrease in size

https://studyc.info/

Lecture 26 Huffman Coding

Using ASCII Code: Text Encoding

Our objective is to develop a code that represents a given text as compactly as possible. A standard encoding is ASCII, which represents every character using 7 bits

Example

Represent "An English sentence" using ASCII code
1000001 (A) 1101110 (n) 0100000 ( ) 1000101 (E) 1101110 (n) 1100111 (g) 1101100 (l)
1101001 (i) 1110011 (s) 1101000 (h) 0100000 ( ) 1110011 (s) 1100101 (e) 1101110 (n)
1110100 (t) 1100101 (e) 1101110 (n) 1100011 (c) 1100101 (e)

= 133 bits ≈ 17 bytes

Refinement in Text Encoding

Now a better code is given by the following encoding:

‹space› = 000, A = 0010, E = 0011, s = 010, c = 0110, g = 0111, h = 1000, i = 1001, l = 1010, t = 1011, e = 110, n = 111

Then we encode the phrase as

0010 (A) 111 (n) 000 ( ) 0011 (E) 111 (n) 0111 (g) 1010 (l) 1001 (i) 010 (s)
1000 (h) 000 ( ) 010 (s) 110 (e) 111 (n) 1011 (t) 110 (e) 111 (n) 0110 (c)
110 (e)

This requires 65 bits ≈ 9 bytes. Much improvement!

Major Types of Binary Coding

There are many ways to represent a file of information.

Binary Character Code (or **Code**)

- each character represented by a unique binary string.

Fixed-Length Code

- If $\Sigma = \{0, 1\}$ then
- All possible combinations of two bit strings

  $\Sigma \times \Sigma = \{00, 01, 10, 11\}$
- If there are less than four characters then two bit strings enough
- If there are less than three characters then two bit strings not economical
- All possible combinations of three bit strings

  $\Sigma \times \Sigma \times \Sigma = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- If there are less than nine characters then three bit strings enough
- If there are less than five characters then three bit strings not economical and can be considered two bit strings
- If there are six characters then needs 3 bits to represent, following could be one representation.

  a = 000, b = 001, c = 010, d = 011, e = 100, f = 101

Variable Length Code

https://studyc.info/

  - better than a fixed-length code
  - It gives short code-words for frequent characters and
  - long code-words for infrequent characters
- Assigning variable code requires some skill
- Before we use variable codes we have to discuss prefix codes to assign variable codes to a set of given characters

Prefix Code (Variable Length Code)

A prefix code is a code typically a variable length code, with the "prefix property". Prefix property is defined as no codeword is a prefix of any other code word in the set.

Examples

1. Code words {0,10,11} has prefix property
2. A code consisting of {0, 1, 10, 11} does not have, because "1" is a prefix of both "10" and "11".

Other names

Prefix codes are also known as prefix-free codes, prefix condition codes, comma-free codes, and instantaneous codes etc.

Why prefix codes?

- Encoding simple for any binary character code;
- Decoding also easy in prefix codes. This is because no codeword is a prefix of any other.

Example 1

If a = 0, b = 101, and c = 100 in prefix code then the string: 0101100 is coded as 0·101·100

Example 2

In code words: {0, 1, 10, 11}, receiver reading "1" at the start of a code word would not know whether

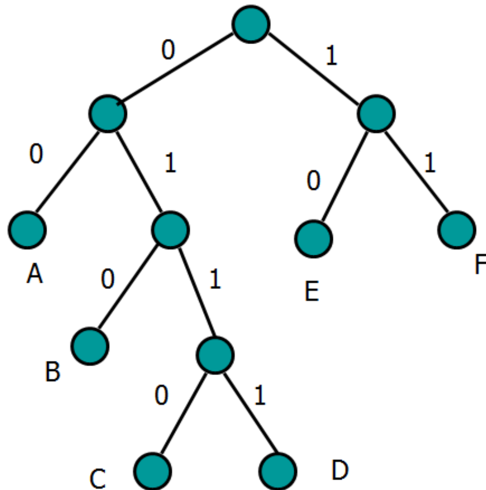
  - that was complete code word "1", or
  - prefix of the code word "10" or of "11"

Prefix codes and Binary Trees

Tree representation of Binary trees

A	00
B	010
C	0110
D	0111
E	10
F	11

https://studyc.info/



Huffman Codes:

- In Huffman coding, variable length code is used
- Data is considered to be a sequence of characters.
- Huffman codes are a widely used and very effective technique for compressing data
  - Savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

Example:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated above.

If each character is assigned a 3-bit fixed-length codeword, the file can be encoded in 300,000 bits.

Using the variable-length code

$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000$ bits

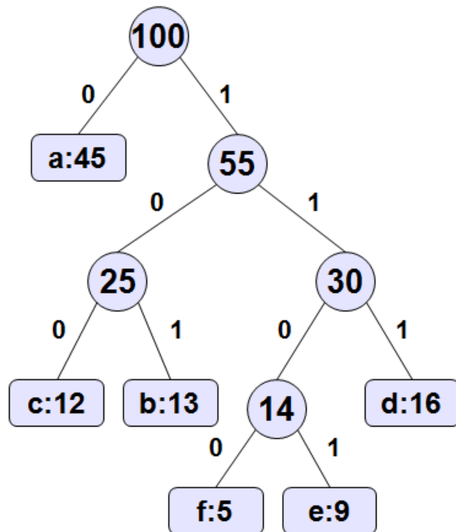
which shows a savings of approximately 25%

https://studyc.info/

Binary Tree: Variable Length Codeword

The tree corresponding to the variable-length code is shown for the data in table.

	Frequency (in thousands)	Variable-length codeword
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



Cost of Tree Corresponding to Prefix Code

- Given a tree T corresponding to a prefix code. For each character c in the alphabet C,
  - let $f(c)$ denote the frequency of c in the file and
  - let $d_T(c)$ denote the depth of c's leaf in the tree.
  - $d_T(c)$ is also the length of the codeword for character c.
  - The number of bits required to encode a file is $B(T) = \sum_{c \in C} f(c) d_T(c)$ which we define as the ***cost*** of the tree T.

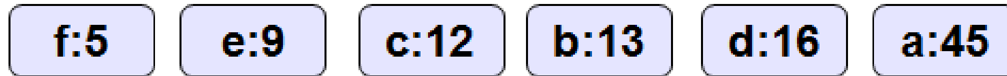
Algorithm: Constructing a Huffman Codes

Huffman (C)
1 $n \leftarrow |C|$

https://studyc.info/

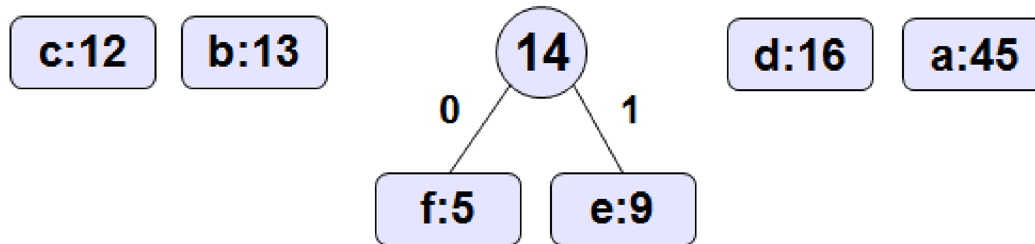
2 $Q \leftarrow C$
3 **for** $i \leftarrow 1$ **to** n - 1
4 **do** allocate a new node z
5 $left[z] \leftarrow x \leftarrow$ Extract-Min (Q)
6 $right[z] \leftarrow y \leftarrow$ Extract-Min (Q)
7 $f[z] \leftarrow f[x] + f[y]$
8 Insert (Q, z)
9 **return** Extract-Min(Q) t Return root of the tree.

Example:

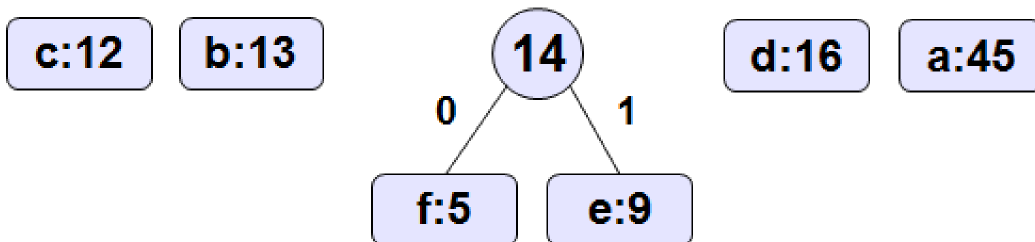


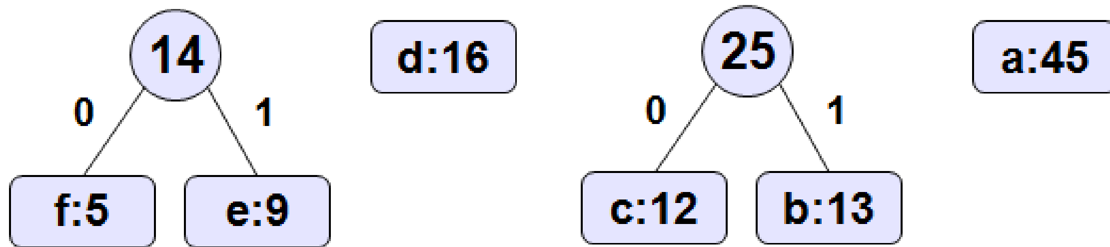
The initial set of n = 6 nodes, one for each letter.

Number of iterations of loop are 1 to n-1 (6-1 = 5)



for i ← 1
Allocate a new node z
$left[z] \leftarrow x \leftarrow$ Extract-Min (Q) = f:5
$right[z] \leftarrow y \leftarrow$ Extract-Min (Q) = e:9
$f[z] \leftarrow f[x] + f[y]$ (5 + 9 = 14)
Insert (Q, z)





for i ← 2

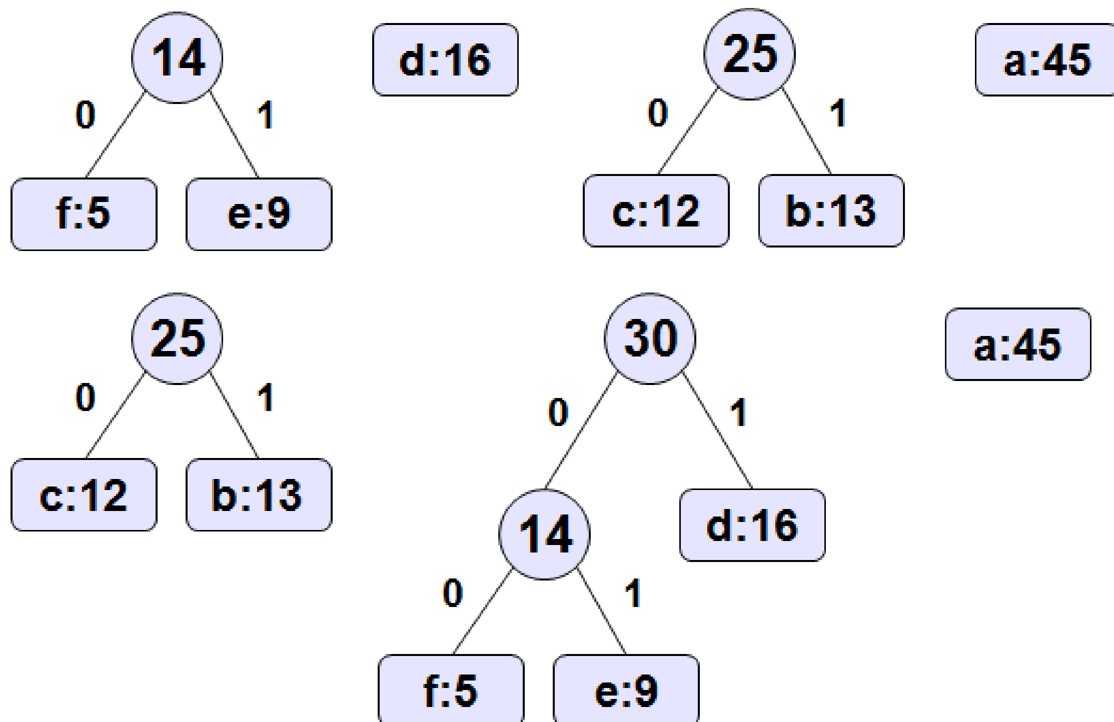
Allocate a new node z

left[*z*] ← *x* ← Extract-Min (*Q*) = c:12

right[*z*] ← *y* ← Extract-Min (*Q*) = b:13

f [*z*] ← *f* [*x*] + *f* [*y*] (12 + 13 = 25)

Insert (*Q*, *z*)



for i ← 3

Allocate a new node z

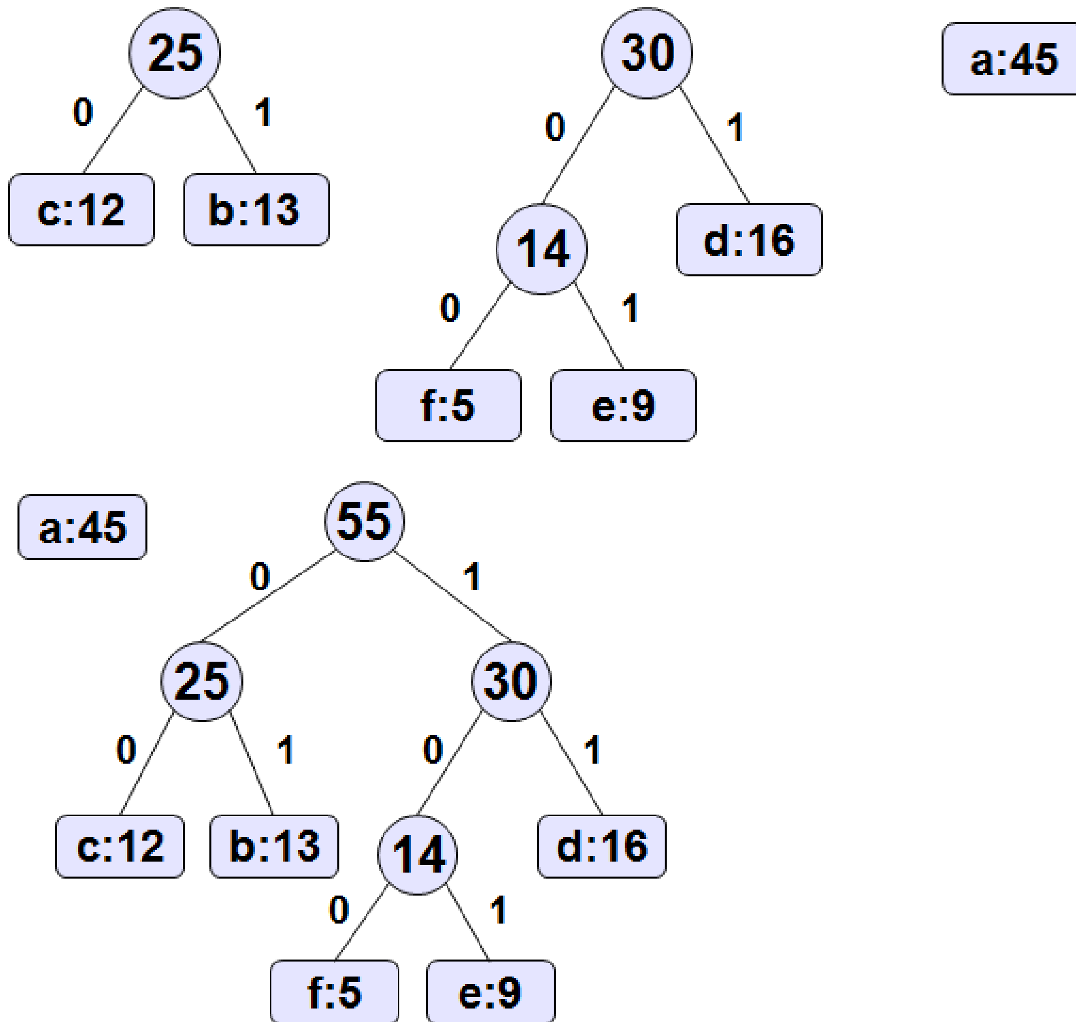
left[*z*] ← *x* ← Extract-Min (*Q*) = z:14

right[*z*] ← *y* ← Extract-Min (*Q*) = d:16

f [*z*] ← *f* [*x*] + *f* [*y*] (14 + 16 = 30)

Insert (*Q*, *z*)

https://studyc.info/



for i ← 4

Allocate a new node z

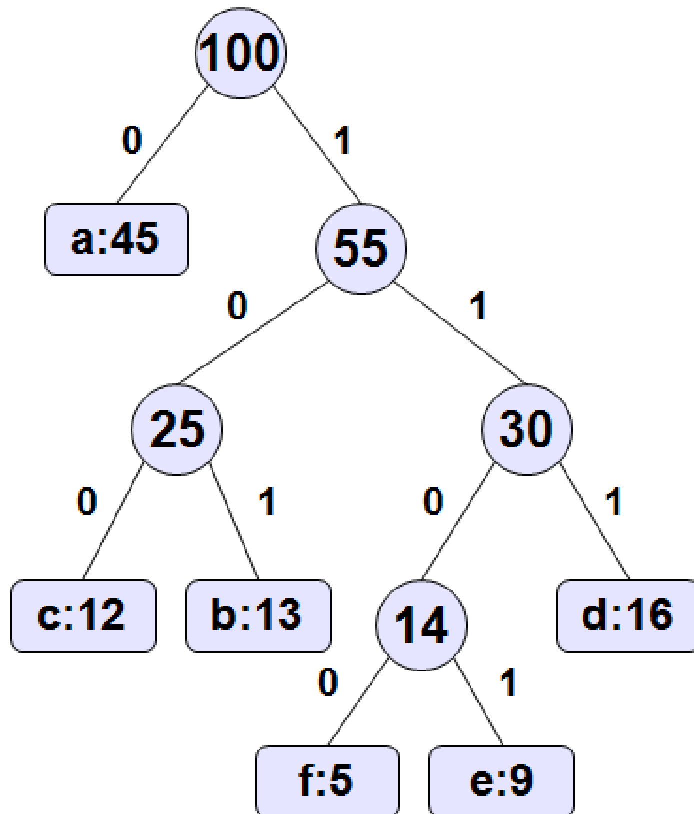
left[*z*] ← *x* ← Extract-Min (*Q*) = z:25

right[*z*] ← *y* ← Extract-Min (*Q*) = z:30

f [*z*] ← *f* [*x*] + *f* [*y*] (25 + 30 = 55)

Insert (*Q*, *z*)

https://studyc.info/



for i ← 5

Allocate a new node z

left[*z*] ← *x* ← Extract-Min (*Q*) = a:45

right[*z*] ← *y* ← Extract-Min (*Q*) = z:55

$f[z] \leftarrow f[x] + f[y]$ (45 + 55 = 100)

Insert (*Q, z*)

Lemma 1: Greedy Choice

There exists an optimal prefix code such that the two characters with smallest frequency are siblings and have maximal depth in T.

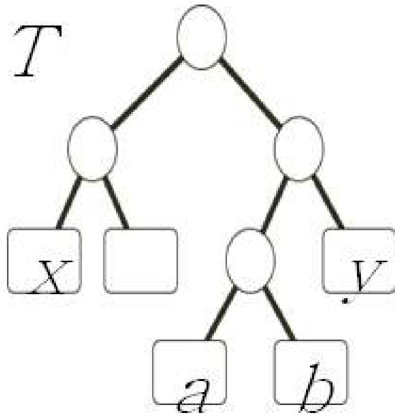
Proof:
Let x and y be two such characters, and let T be a tree representing an optimal prefix code.

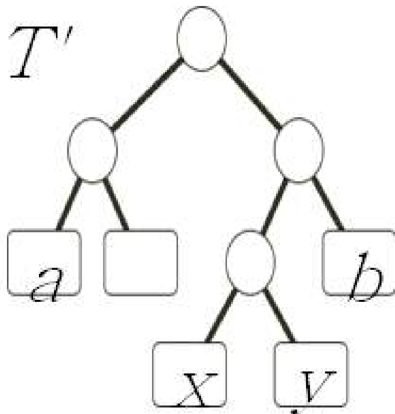
Let a and b be two sibling leaves of maximal depth in T, and assume without loss of generality that $f(x) \leq f(y)$ and $f(a) \leq f(b)$.

This implies that $f(x) \leq f(a)$ and $f(y) \leq f(b)$.

Let T' be the tree obtained by exchanging a and x and b and y.

https://studyc.info/





The cost difference between trees *T* and *T′* is

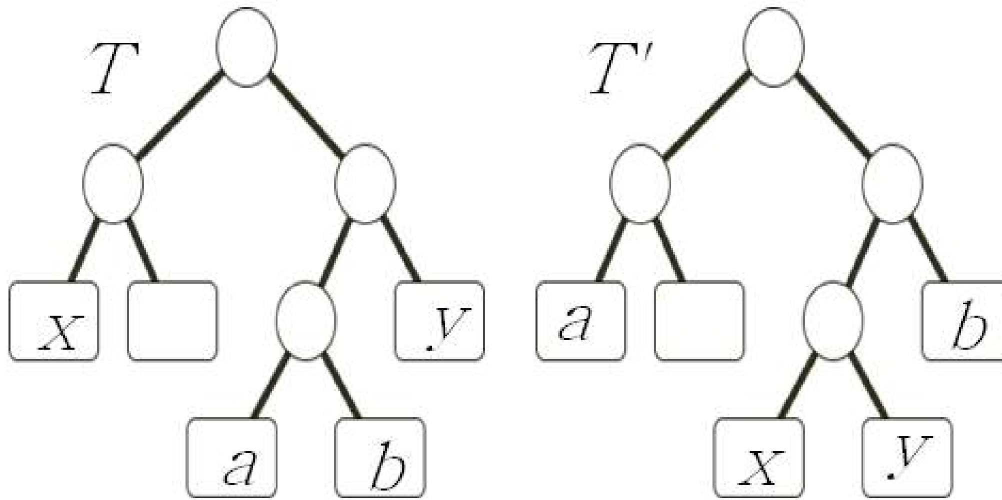
$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(a) d_T(a) - f(x) d_{T'}(x) - f(a) d_{T'}(a) + \\
&\quad f(y) d_T(y) + f(b) d_T(b) - f(y) d_{T'}(y) - f(b) d_{T'}(b) \\
&= \big(f(a) - f(x)\big)\big(d_T(a) - d_T(x)\big) + \big(f(b) - f(y)\big)\big(d_T(b) - d_T(y)\big) \\
&\geq 0
\end{aligned}
$$

Hence B(T’) ≤ B(T)

Since B(T) ≤ B(T’)

Hence B(T) = B(T’)

https://studyc.info/



https://studyc.info/

Lecture 27 Huffman Coding Problem and Graph Theory

Algorithm: Constructing a Huffman Codes

Huffman (C)
1 $n \leftarrow |C|$
2 $Q \leftarrow C$
3 **for** $i \leftarrow 1$ **to** n - 1
4 **do** allocate a new node z
5 $left[z] \leftarrow x \leftarrow$ Extract-Min (Q)
6 $right[z] \leftarrow y \leftarrow$ Extract-Min (Q)
7 $f[z] \leftarrow f[x] + f[y]$
8 Insert (Q, z)
9 **return** Extract-Min(Q) t Return root of the tree.

Lemma 2: Optimal Substructure Property

Let C be a given alphabet with frequency $f[c]$ defined for each character $c \in C$.
Let x, y (characters) $\in C$ with minimum frequency.
Let C' be alphabet C with characters x, y removed, new character z added, so that $C' = C - \{x, y\} \cup \{z\}$;
Define f for C' as for C, except that $f[z] = f[x] + f[y]$.
Let T' be any tree representing an optimal prefix code for the alphabet C'.
Then tree T, obtained from T' by replacing leaf node for z with an internal node having x, y as children, represents an optimal prefix code for alphabet C.

Proof:
Since $C' = C - \{x, y\} \cup \{z\}$; where $f(z) = f(x) + f(y)$,
We are give that T' is an optimal tree for C'.
Let T be tree obtained from T' by making x, y children of z.
We prove: $B(T) = B(T') + f(x) + f(y)$

$$
\begin{aligned}
B(T) &= \sum_{c \in C} f(c) d_T(c) \\
&= \sum_{c \in C \setminus \{x,y\}} f(c) d_T(c) + f(x) d_T(x) + f(y) d_T(y) \\
&= \sum_{c \in C' \setminus \{z\}} f(c) d_{T'}(c) + \big(f(x) + f(y)\big)\big(d_{T'}(z) + 1\big) \\
&= \sum_{c \in C' \setminus \{z\}} f(c) d_{T'}(c) + f(z) d_{T'}(z) + f(x) + f(y) \\
&= \sum_{c \in C' \setminus \{z\}} f(c) d_{T'}(c) + f(x) + f(y) \\
&= B(T') + f(x) + f(y)
\end{aligned}
$$

$B(T') = B(T) - f(x) - f(y)$

If T' is optimal for C', then T is optimal for C?

Assume on contrary that there exists a better tree T'' for C, such that $B(T'') < B(T)$
Assume without loss of generality T'' has siblings x and y.

https://studyc.info/

Let tree T''' be a tree T'' with the common parent of x and y replaced by vertex z with frequency $f[z] = f[x] + f[y]$. Then

$$B(T''') = B(T'') - f(x) - f(y)$$
$$< B(T) - f(x) - f(y) \qquad \text{Since, } B(T'') < B(T)$$
$$= B(T').$$

This contradicts the optimality of $B(T')$.
Hence, T must be optimal for C.

Road Trip Problem

You purchase a new car. On your semester break, you decide to take a road trip from Peshawar to Karachi. Your car has a tank of some capacity such that only a distance k km can be traveled before refilling the tank.
Suppose there are filling stations at distances of $d_0 < d_1 < d_2 < ... < d_n$ where d_n is the total distance of your trip.
Your goal is to find the smallest number of stops required i.e. shortest subsequence of $< d_0 \cdots d_n >$, given that you start at d_0 and end at d_n.

INPUT**:** The max distance k, along with the distances: $d_0, d_1, ..., d_n$.

GOAL**:** To find a smallest sub sequence of $d_0, d_1, ..., d_n$ so that you can start from d_0 and end at d_n.

Note:

- Greedy approach is considered each d_i in order.
- We stop to refuel at d_i only if the tank will finish before we get to d_{i+1}.

Road Trip Problem (Greedy algorithm)

```
1.      for i = 1 to n do
2.          if di – di-1 > k then "do not use this car"
3.      S = d0
4.      last = d0 (the distance of the last item in S)
5.      dn+1 = ∞ (forces dn to be in S)
6.      for i = 1 to n do
7.          if di+1 - last > k then
8.              S := S ∪ di
9.              last := di
```

Graph Theoretic Concepts

Definitions:

Graph *G* consists of two finite sets *V(G)* and *E(G)*

Endpoints a set of one or two vertices of an edge

https://studyc.info/

Loop an edge with just one endpoint

Edge-endpoint function: End-Point-Function: E→ Set of V

Parallel edges two distinct edges with same endpoints

Adjacent vertices vertices connected by an edge

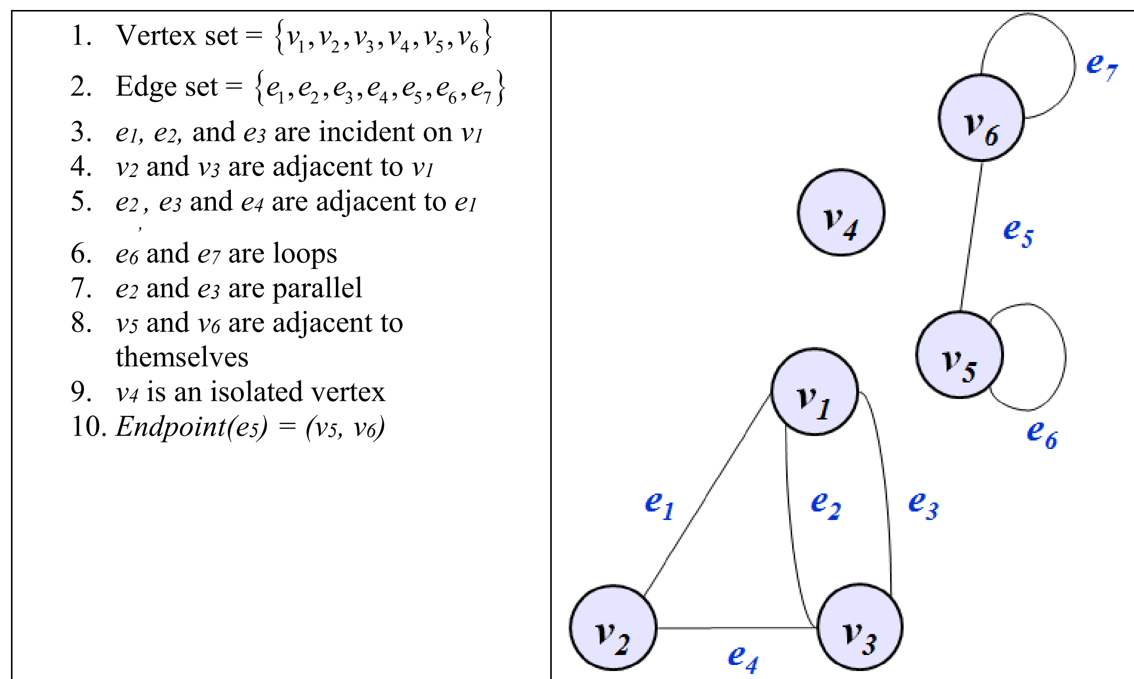
Vertex adjacent to itself vertex endpoint of a loop

Adjacent edges two edges incident on the same endpoint

Isolated a vertex on which no edge is incident

Empty a graph with no vertices, otherwise **nonempty**.

Examples:



Directed, Simple and Complete Graph

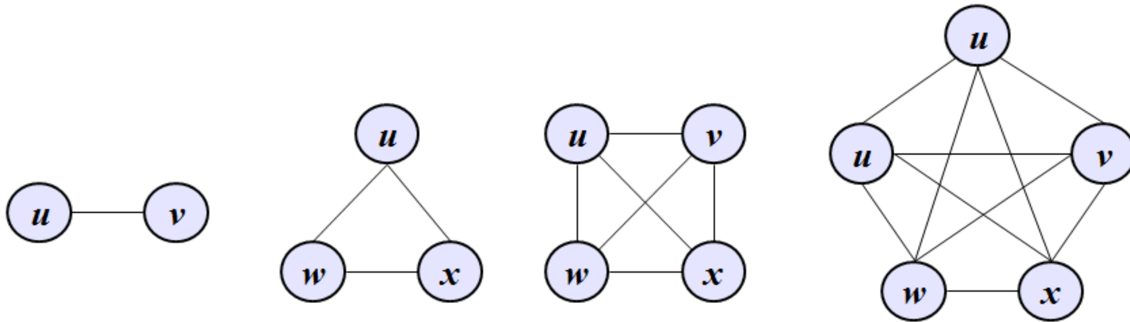
Directed graph (digraph) in which each edge is associated with an **ordered pairs** of vertices

Simple graph does not have any loop or parallel edge

Subgraph H is subgraph of G if and only if, every vertex in H is also vertex in G, and every edge in H has the same endpoints as in G.

Complete graph on n vertices, (K_n) is a simple graph in which each pair of vertices has exactly one edge.

https://studyc.info/



Complete Graph

Example 1: Find a recursive formula to compute number of edges in a **complete graph on *n* vertices**.

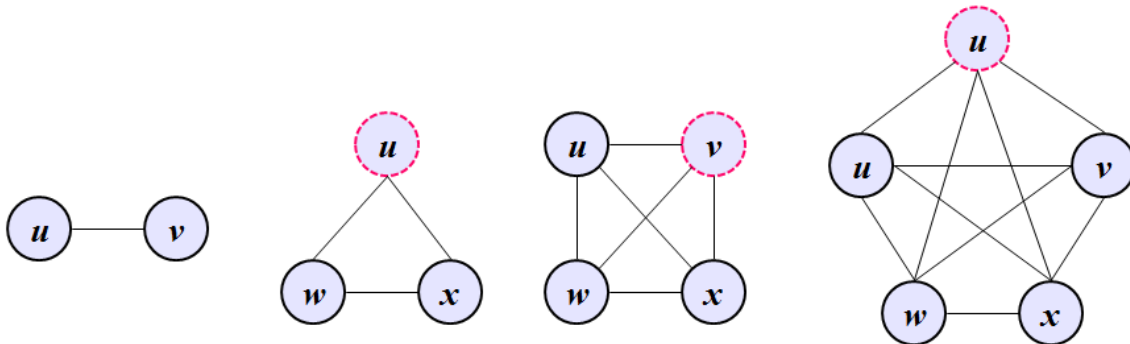
Solution:

Let S_k = total number of edges of a complete graph with k vertices

S_k = total number of edges of a complete graph with k - 1 vertices + total number of edges connected k^{th} node

= total number of edges of a complete graph with k - 1 vertices + k-1 number of edges

$S_k = S_{k-1} + (k - 1)$



Example 2:

Find an **explicit** formula to compute number of edges in **complete graph on *n* vertices**.

Solution:

Since, $S_k = S_{k-1} + (k - 1) \Rightarrow S_{k-1} = S_{k-2} + (k - 2)$ and so on

By back substitution

$S_k = S_{k-2} + (k - 2) + (k - 1) = S_{k-3} + (k - 3) + (k - 2) + (k - 1)$

$S_k = S_1 + 1 + 2 + \ldots + (k - 2) + (k - 1) = (k-1)k/2$

$S_k = (k-1)k/2$

$S_n = (n-1)n/2$

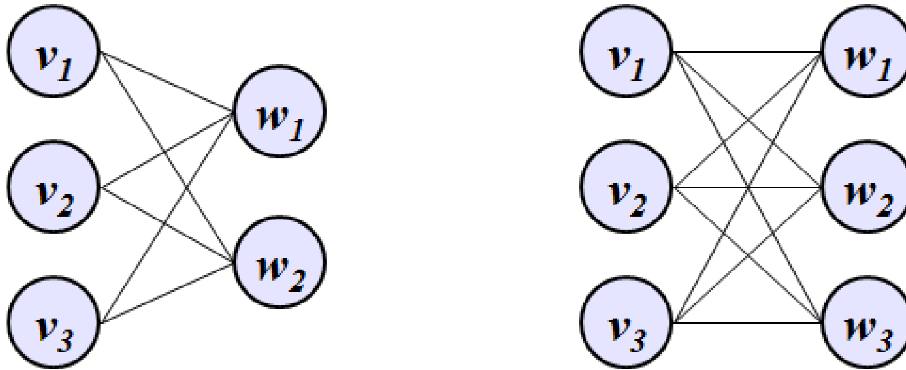
https://studyc.info/

Complete Bipartite Graph

A **complete bipartite** graph on (m, n) vertices, denoted $K_{m,n}$, is a simple graph with vertices $v_1, v_2, \ldots, v_n$ and $w_1, w_2, \ldots, w_n$ that satisfies the following properties:

for all $i, k = 1, 2, \ldots, m$ and for all $j, l = 1, 2, \ldots, n$

1. there is an edge from each vertex v_i to each vertex w_j ;
2. there is not an edge from any vertex v_i to any other vertex v_k ;
3. there is not an edge from any vertex w_j to any other vertex w_l



Degree of Graph:

Definition degree of v is number of edges incident on *v*

Theorem:

In a graph G, sum of degrees of all vertices equals twice the number of edges of *G*. Specifically, if the vertices of *G* are $v_1, v_2, \ldots, v_n$ where *n* is a positive integer, then

Total degree of $G = \deg(v_1) + \deg(v_2) + \ldots + \deg(v_n)$

$= 2$. (the number of edges of *G*)

Note: Total degree of an undirected graph, G, is even.

Proposition

In a graph there are even numbers of vertices of odd degree.

Solution:

Let us suppose that: V = all vertices of a graph

$V_1 = \{v_1, v_2, \ldots, v_k\}$ = set of all vertices of odd degree

$V_2 = \{w_1, w_2, \ldots, w_m\}$ = set of all vertices of even degree

Now we have to prove that k is even?

On contrary, suppose that k is odd

Degree (graph) = $\deg(v_1) + \deg(v_2) + , \ldots, \deg(v_k) + \deg(V_2)$

= odd + even = odd, contradiction.

Hence k must be even.

https://studyc.info/

That is there even number of vertices of odd degree.

Walk of Graph

Walk from *v* to *w* is a finite alternating sequence of adjacent vertices and edges of *G*. It has the form $v = v_0 e_1 v_1 e_2 \ldots v_{n-1} e_n v_n = w$,

for all $i = 1, 2, \ldots n$, v_{i-1} and v_i are endpoints of e_i .
Trivial walk from *v* to *v* consists of single vertex *v*.

Closed walk starts and ends at the same vertex

Path a walk that does not contain a repeated edge.

$$v = v_0 e_1 v_1 e_2 \ldots v_{n-1} e_n v_n = w,$$

where all the e_i are distinct (that is, $e_i \neq e_k$ for any $i \neq k$).
Simple path a path that does not contain a repeated vertex. Thus a simple path is a walk of the form

$$v = v_0 e_1 v_1 e_2 \ldots v_{n-1} e_n v_n = w,$$

*a*ll e_i and v_j are distinct ($v_i \neq v_j$, $e_i \neq e_j$ for any $i \neq j$).

Circuit of Graph

A **circuit** is a closed walk that does not contain a repeated edge. Thus a circuit is a walk of the form $v_0 e_1 v_1 e_2 \ldots v_{n-1} e_n v_n$

where $v_0 = v_n$ and all the e_i are distinct.
A **simple circuit** is a circuit that does not have any other repeated vertex except the first and last. Thus a simple circuit is walk of the form $v_0 e_1 v_1 e_2 \ldots v_{n-1} e_n v_n$

where all the e_i are distinct and all the v_j are distinct except that $v_0 = v_n$

Euler Circuit

An **Euler circuit** for *G* is a circuit that contains every vertex and every edge of *G*. That is, an Euler circuit is a sequence of adjacent vertices and edges in *G* that starts and ends at the same vertex, uses every vertex of *G* at least once, and every edge exactly once.

Theorem If a graph has an Euler circuit, then every vertex of the graph has even degree.

Theorem A graph *G* has an Euler circuit if, and only if, *G* is connected and every vertex of *G* has even degree.

Euler Path

Let *G* be a graph and let *v* and *w* be two vertices of *G*. An **Euler path** from *v* to *w* is a sequence of adjacent edges and vertices that starts at *v*, ends at *w*, passes through every vertex of *G* at least once, and traverses every edge of *G* exactly once.

https://studyc.info/

Corollary
Let *G* be a graph and let *v* and *w* be two vertices of *G*. There is an **Euler path** from *v* to *w* if, and only if, *G* is connected, *v* and *w* have odd degree, and all other vertices of *G* have even degree.

Hamiltonian Circuit

Given a graph *G*, a **Hamiltonian circuit** for *G* is a simple circuit that includes every vertex of *G*. That is, a Hamiltonian circuit of *G* is a sequence of adjacent vertices and distinct edges in which every vertex of *G* appears exactly once.

If a graph *G* has a Hamiltonian circuit then *G* has a sub-graph *H* with the following properties:

1. *H* contains every vertex of *G;*
2. *H* is connected;
3. *H* has the same number of edges as vertices;
4. every vertex of *H* has degree 2.

Connected Graph

Two vertices v and w of G are **connected** if, and only if, there is a walk from *v* to *w*.

***G* is connected** if, and only if, for *any* two vertices *v* and *w* in *G*, there is a walk from *v* to *w*. symbolically:

G is connected Û A *v*, *w e V(G), E* a walk from *v* to *w*

Lemma

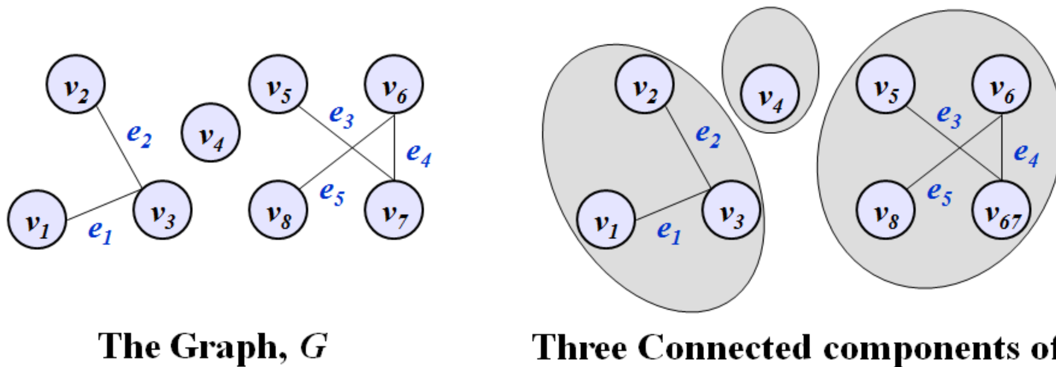
a. If *G* is connected, then any two distinct vertices of *G* can be connected by a simple path.
b. If *v, w* are in circuit and one edge is removed from the circuit, then there still exists a path from *v* to *w*
c. If *G* is connected and contains a circuit, then an edge of circuit can be removed without disconnecting *G*.

Connected Component

A graph *H* is a **connected component** of a graph *G* if, and only if,

1. *H* is a subgraph of *G;*
2. *H* is connected;
3. No connected subgraphs of *G* has *H* as a subgraph and contains vertices or edges that are not in *H*.

https://studyc.info/



The Graph, *G* **Three Connected components of *G***

Isomorphism

Let *G* and *G'* be graphs with vertex sets *V(G)* and *V(G')* and edge sets *E(G)* and *E(G')*, respectively. ***G* is isomorphic to *G'*** if, and only if, there exist one-to-one correspondence *g : V(G) f V(G')* and *h : E(G) f E(G')* that preserve the edge-endpoint functions of *G* and *G'* in the sense that

for all *v e V(G)* and *e e E(G)* ,

v is an endpoint of *e Û g(v)* is an endpoint of *h(e)*

Isomorphism Invariants

A property *P* is called an **isomorphic invariant** if, and only if, given any graphs *G* and *G'*, if *G* has property *P* and *G'* is isomorphic to *G*, then *G'* has property *P*.

If *G* and *G'* are simple graphs then ***G* is isomorphic to *G'*** if, and only if, there exists a one-to-one correspondence *g* from the vertex set *V(G)* of *G* to the vertex set *V(G')* of *G'* that preserves the edge-endpoint functions of *G* and *G'* in the sense that

for all vertices *u* and *v* of *G*,

{u, v} is an edge in *G* Û*{g(u), g(v)}* is an edge in *G'*

Theorem
Each of following properties is an invariant for graph isomorphism, *n, m,* and *k* are all nonnegative integers:

1. has *n* vertices;
2. has *m* edges;
3. has a vertex of degree *k;*
4. has *m* vertices of degree *k;*
5. has a circuit of length *k;*
6. has a simple circuit of length *k;*
7. has *m* simple circuits of length *k*;
8. is connected;
9. has an Euler circuit;
10. has a Hamiltonian circuit.

https://studyc.info/

Trees

- Graph is **circuit-free** ⇔ it has no nontrivial circuits.
- A graph is a **tree** ⇔ it is circuit-free and connected.
- A **trivial tree** is a graph that consists of a single vertex
- **Empty tree** that does not have any vertices or edges.
- **Forest** a graph if circuit-free.
- **Terminal vertex (Leaf)** a vertex of degree 1in T
- **Internal vertex** a vertex of degree greater than 1 in T

Lemma 1 Any tree having more than one vertices has at least one vertex of degree 1.

Lemma 2 If G is any connected graph, C is any nontrivial circuit in G, and one of the edges of C is removed form G', then the graph that remains is connected.

Theorem For any positive integer n, if G is a connected graph with n vertices and $n - 1$ edges, then G is a tree

Theorem

Statement

For positive integer n, any tree with n vertices has $n - 1$ edges.

Solution

We prove this theorem by mathematical induction

Basis

n = 1, tree has no edge and hence true

Inductive hypothesis

Suppose that id n = k then tree has k – 1 edges

Claim

Now if we add one more vertex to the tree then exactly one edge will be added otherwise it will not remain tree. And hence it will become k edges in the tree. Proved!

Rooted Trees

Rooted tree a distinguished vertex

Level of a vertex is the number of edges along the unique path between it and the root.

Height of a rooted tree is maximum level to any vertex

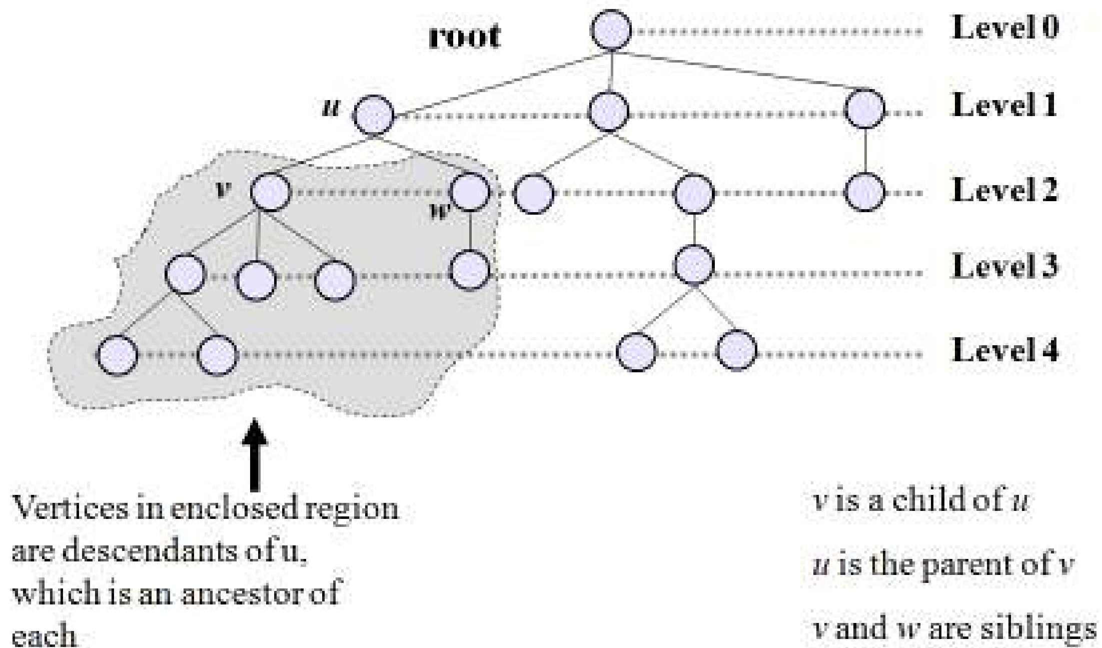
Children of v are all those vertices that are adjacent to v and are one level farther away from the root than v.

Parent if w is child of v, then v its parent

Siblings vertices that are children of same parent

Ancestor and **Descendent** given vertices v and w, if v lies on the unique path between w and the root, then v is an **ancestor** of w and w is a **descendent** of v

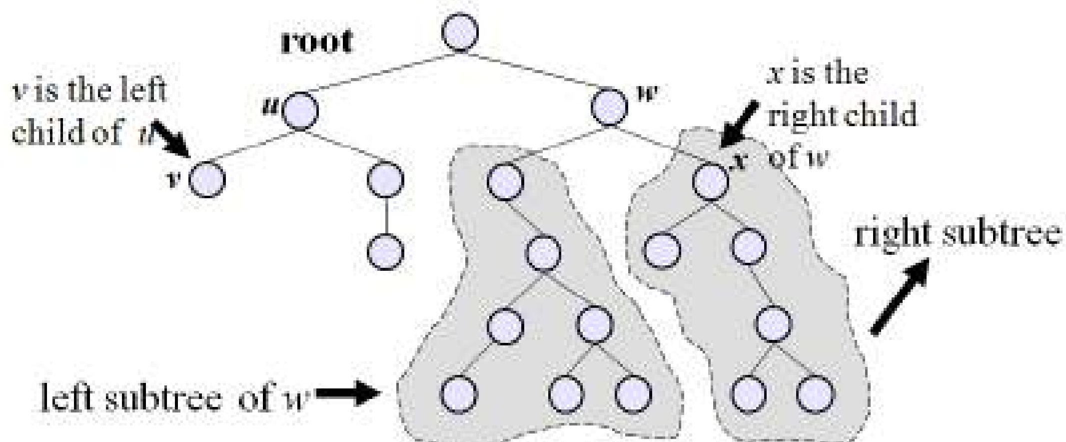
https://studyc.info/



<u>**Binary Trees**</u>

A **binary tree** is a rooted tree in which every internal vertex has at most two children. Each child in a binary tree is either **left child** or a **right child** (but not both), an internal vertex has at most one left and one right child.

Full binary tree is a binary tree in which each internal vertex has exactly two children.



Theorem
If k is a positive integer and T is a full binary tree with k internal vertices, the T has a total of $2k + 1$ vertices and has $k + 1$ terminal vertices.

Theorem

If T is a binary tree that has t number of terminal vertices and height is h, then $t \leq 2_k$ OR
$\log_2 t \leq h$

https://studyc.info/

Spanning Trees

A **spanning tree** for a graph *G* is a subgraph of *G* that contains every vertex of *G* and is a tree.

Proposition

1. Every connected graph has a spanning tree.
2. Any two spanning trees for a graph have the same number of edges.

Minimal Spanning Trees

A **weighted graph** is a graph for which each edge has an associated real number **weight**. The sum of the weights of all the edges is the **total weight** of the graph.

A **minimal spanning tree** for a weighted graph is a spanning tree that has the least possible total weight compared to all other spanning trees for the graphs.

If G is a weighted graph and e is an edge of G then $w(e)$ denotes the weight of e and $w(G)$ denotes the total weight of G.

https://studyc.info/

Lecture 28 Breadth First Search

Representations of Graphs

- Two standard ways to represent a graph
  - Adjacency lists,
  - Adjacency Matrix
- Applicable to directed and undirected graphs.

Adjacency lists

- A compact way to represent **sparse graphs**.
  - $|E|$ is much less than $|V|^2$
- Graph $G(V, E)$ is represented by array Adj of $|V|$ lists
- For each $u \in V$, the adjacency list $Adj[u]$ consists of all the vertices adjacent to u in G
- The amount of memory required is: $(V + E)$

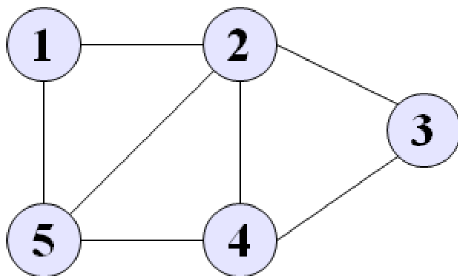
Adjacency Matrix

- A graph $G(V, E)$ assuming the vertices are numbered 1, 2, 3, … , $|V|$ in some arbitrary manner, then representation of G consists of: $|V| \times |V|$ matrix A = (a_{ij}) such that

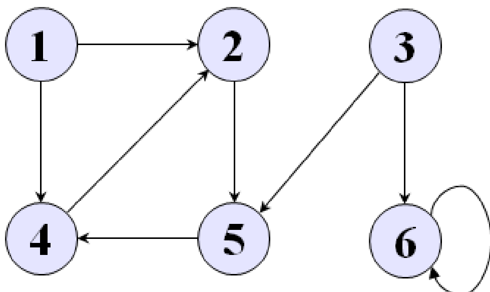
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Preferred when graph is **dense**
  - $|E|$ is close to $|V|^2$

Adjacency matrix of undirected graph



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

https://studyc.info/

The amount of memory required is $\Theta(V^2)$

For undirected graph to cut down needed memory only entries on and above diagonal are saved. In an undirected graph, *(u, v)* and *(v, u)* represents the same edge, adjacency matrix A of an undirected graph is its own **transpose A = A^T**

It can be adapted to represent **weighted graphs**.

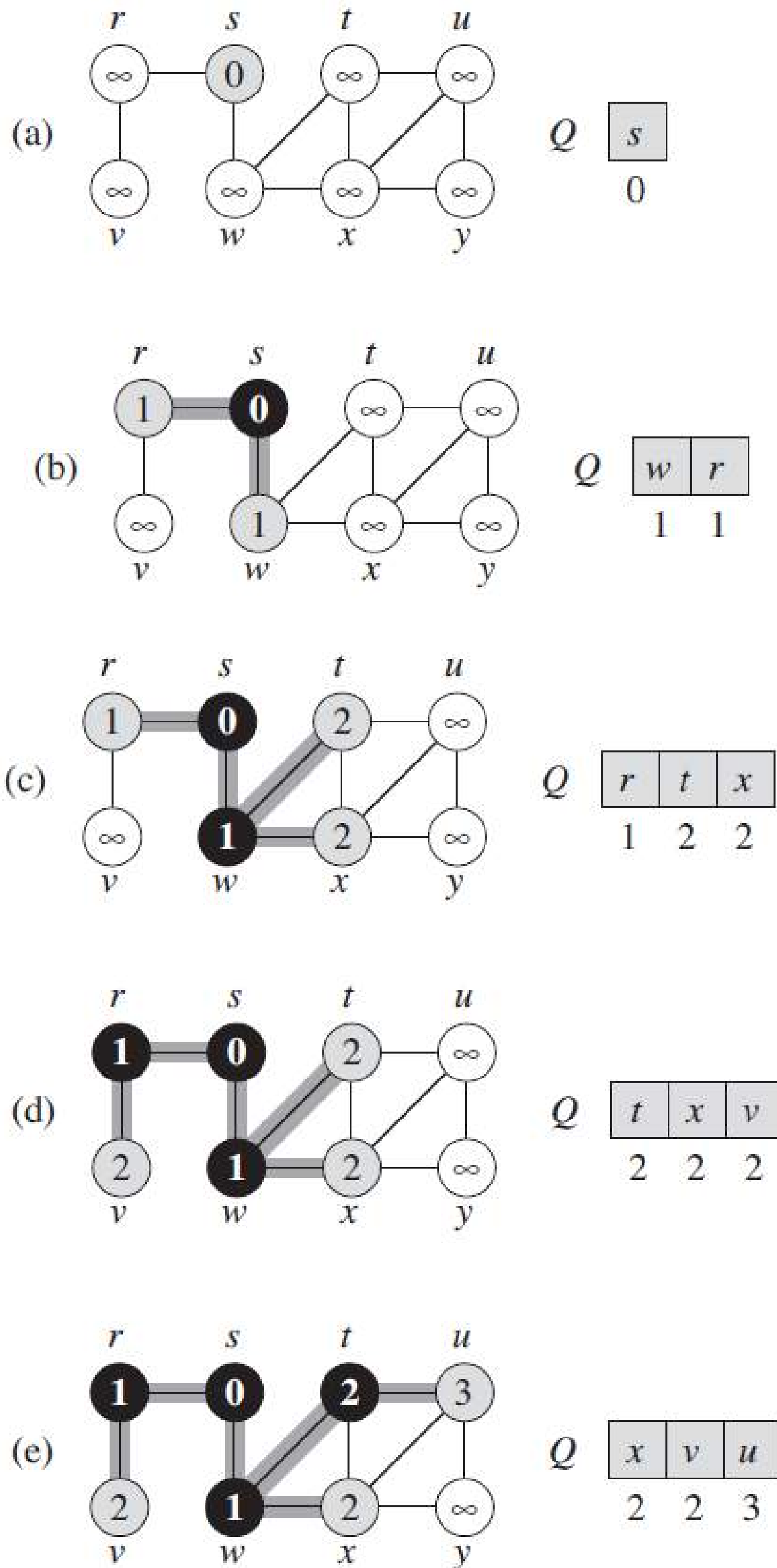
Breadth First Search

- One of **simplest** algorithm searching graphs
- A vertex is **discovered** first time, encountered
- Let *G (V, E)* be a graph with **source** vertex *s*, BFS
  - discovers every vertex reachable from *s*.
  - gives distance from *s* to each reachable vertex
  - produces BF tree root with *s* to reachable vertices
- To keep track of progress, it colors each vertex
  - vertices start **white,** may later **gray,** then **black**
  - Adjacent to black vertices have been discovered
  - Gray vertices may have some adjacent white vertices
- It is assumed that input graph *G (V, E)* is represented using adjacency list.
- Additional structures maintained with each vertex $v \in V$ are
  - *color[u]* – stores color of each vertex
  - *π[u]* – stores predecessor of *u*
  - *d[u]* – stores distance from source *s* to vertex *u*

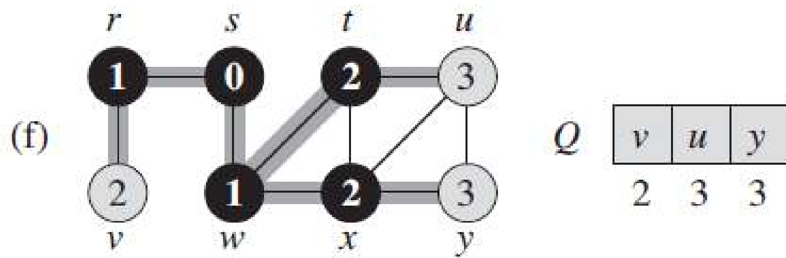
Algorithm

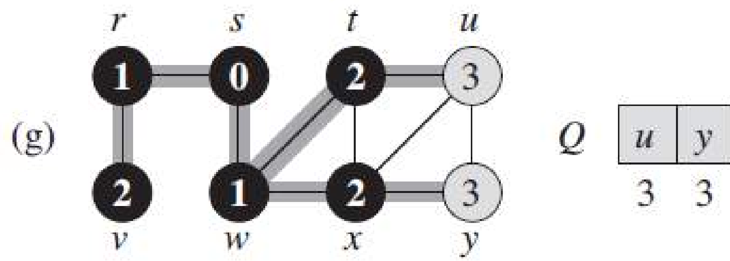
```
BFS(G, s)
 1   for each vertex u ← V [G] – {s}
 2      do color [u] ← WHITE
 3         d [u] ← ∞
 4         π[u] ← NIL
 5   color[s] ← GRAY
 6   d [s] ← 0
 7   π[s] ← NIL
 8   Q ← Ø                    /* Q always contains the set of GRAY vertices */
 9   ENQUEUE (Q, s)
10   while Q ≠ Ø
11      do u ← DEQUEUE (Q)
12         for each v ∈ Adj [u]
13             do if color [v] = WHITE            /* For undiscovered vertex. */
14                   then color [v] ← GRAY
15                        d [v] ← d [u] + 1
16                        π[v] ← u
17                        ENQUEUE(Q,  v)
18         color [u] ← BLACK
```

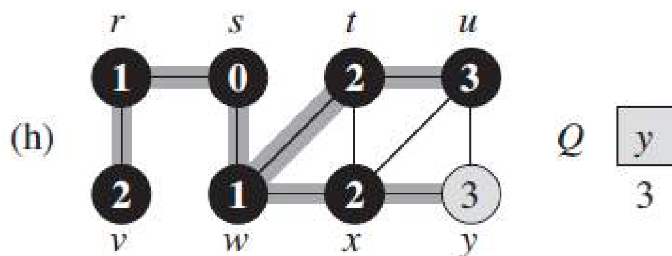
https://studyc.info/

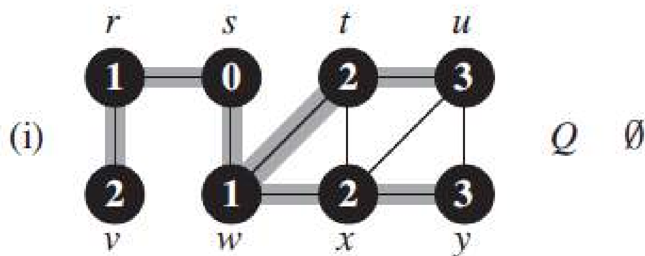


https://studyc.info/









- Each vertex is enqueued and dequeued atmost once
  - Total time devoted to queue operation is $O(V)$
- The sum of lengths of all adjacency lists is $\Theta(E)$
  - Total time spent in scanning adjacency lists is $O(E)$
- The overhead for initialization $O(V)$

Total Running Time of BFS = $O(V+E)$

From CLRS Textbook: The operations of enqueuing and dequeuing take O(1) time, and so the total time devoted to queue operations is O(V). Because the procedure scans the adjacency list

https://studyc.info/

of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$ and thus the total running time of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Shortest Paths

- The **shortest-path-distance** $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex *s* to vertex *v.*
    - if there is no path from *s* to *v,* then $\delta(s, v) = \infty$
- A path of length $\delta(s, v)$ from *s* to *v* is said to be a **shortest path** from *s* to *v*.
- Breadth First search finds the distance to each reachable vertex in the graph $G(V, E)$ from a given source vertex $s \in V$.
- The field *d,* for distance, of each vertex is used

Algorithm

BFS-Shortest-Paths (G, s)

```
1   ∀ v ∈ V
2      d [v] ← ∞
3   d [s] ← 0
4   ENQUEUE (Q, s)
5   while Q ≠ φ
6        do v ← DEQUEUE(Q)
7           for each w in Adj[v]
8               do if d [w] = ∞
9                    then d [w] ← d [v] +1
10                        ENQUEUE (Q, w)
```

Lemma 1: Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$\delta(s, v) \le \delta(s, u) + 1$

Proof: If *u* is reachable from s, then so is *v*. In this case, the shortest path from s to *v* cannot be longer than the shortest path from *s* to *u* followed by the edge (u, v), and thus the inequality holds. If *u* is not reachable from *s*, then $\delta(s,u) = \infty$, and the inequality holds.

Lemma 2: Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $d[v] \ge \delta(s, v)$.

Proof:

- Induction on number of ENQUEUE operations.
    - To prove $d[v] \ge \delta(s, v)$ for all $v \in V$

- The basis of the induction is situation immediately after s is enqueued in line 9 of BFS algorithm.
- Base case holds, because $d[s] = 0 = \delta(s, s)$ and $d[v] = \infty \geq \delta(s, v)$ for all $v \in V - \{s\}$
- Inductive hypothesis: $d[u] \geq \delta(s, u)$. Here white vertex v is discovered during search from a vertex u.
- By line 15 of BFS we have $d[v] = d[u] + 1$ (1)
- By Inductive hypothesis $d[u] \geq \delta(s, u)$ (2)
- By previous Lemma, $\delta(s, u) + 1 \geq \delta(s, v)$ (3)
- Now by (1), (2) and (3),

$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$

- *Hence* $d[v] \geq \delta(s, v)$. Vertex v is then enqueued, and never enqueued again because it is also grayed.
- Hence it prove the theorem

Lemma 3: Suppose that during execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $<v_1, v_2, \ldots, v_r>$, *where* v_1 is the head of Q and v_r is the tail. Then, $d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_{i+1}]$ for $i = 1, 2, \ldots, r - 1$

Proof

- Proof is done by induction on queue operations
- Initially, when queue contains s, lemma holds.
- For inductive step, we must prove that lemma holds after dequeuing and enqueuing a vertex.

Dequeuing a vertex.

- If head v_1 of queue is dequeued, v_2 becomes new head. (If queue is empty, lemma holds vacuously.)
- *Now* $d[v_1] \leq d[v_2]$ (by inductive hypothesis)
- *To prove that* $d[v_r] \leq d[v_2] + 1$
- We have $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$
- And remaining inequalities are unaffected.

Enqueuing a vertex.

- When we enqueue vertex v_{r+1}
- At time of enqueuing v_{r+1}, let u was removed. Hence, by inductive hypothesis, $d[v_1] \geq d[u]$ i.e.

  $d[u] \leq d[v_1]$. (1)

- *Since* v_{r+1} *is adjacent to u*

$d[v_{r+1}] = d[u] + 1$ (2)

- By (1) and (2), $d[v_{r+1}] = d[u] + 1 \leq d[v_1] + 1$
- By inductive hypothesis we have $d[v_r] \leq d[u] + 1$

https://studyc.info/

- Now $d[v_r] \leq d[u] + 1 = d[v_{r+1}]$, and the remaining inequalities are unaffected.
- Thus, the lemma is proved when v_{r+1} is enqueued

Corollary: Suppose that vertices v_i and v_j are enqueued during execution of BFS, and that v_i is enqueued before v_j. Then $d[v_i] \leq d[v_j]$ at the time that v_j *is* enqueued.

Proof:

- Immediate from above Lemma and
- the property that each vertex receives a finite d value at most once during the course of BFS

Theorem (Correctness of BFS)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination

$$d[v] = \delta(s, v) \text{ for all } v \in V.$$

Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by edge $(\pi[v], v)$.

Proof:

- Assume, for the purpose of contradiction, that some vertex receives a d value not equal to its shortest path distance.
- Let v be the vertex with minimum $\delta(s, v)$ that receives such an incorrect d value; clearly $v \neq s$.
- By Lemma 22.2, $d[v] \geq \delta(s, v)$, and thus we have that $d[v] > \delta(s, v)$. Vertex v must be reachable from s, for if it is not, then $\delta(s, v) = \infty \geq d[v]$.
- Let u be the vertex immediately preceding v on a shortest path from s to v, so that $$\delta(s, v) = \delta(s, u) + 1.$$
- Because $\delta(s, u) < \delta(s, v)$, and because of how we chose v, we have $d[u] = \delta(s, u)$.
- Putting these properties together, we have $$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 \qquad (22.1)$$
- Now consider the time when BFS chooses to dequeue vertex u from Q in line 11.
- At this time, vertex v is, white, gray, or black.
- We shall show that in each of these cases, we derive a contradiction to inequality (22.1).
- If v is white, then line 15 sets $d[v] = d[u] + 1$, contradicting inequality (22.1).
- If v is black, then it was already removed from the queue and, by Corollary 22.4, we have $d[v] \leq d[u]$, again contradicting inequality (22.1).
- If v is gray, then it was painted gray upon dequeuing some vertex w, which was removed from Q earlier than u and, $d[v] = d[w] + 1$.
- By Corollary 22.4, however, $d[w] \leq d[u]$, and so we have $d[v] \leq d[u] + 1$, once again contradicting inequality (22.1).

https://studyc.info/

- Thus we conclude that $d[v] = \delta(s, v)$ for all $v \in V$. All vertices reachable from s must be discovered, if they were not, they would have infinite d values.
- To conclude the proof of the theorem, observe that if $\pi[v] = u$, then $d[v] = d[u] + 1$.
- Thus, we can obtain a shortest path from s to v by taking a shortest path from s to $\pi[v]$ and then traversing the edge $(\pi[v], v)$

Lemma: When applied to a directed or undirected graph $G = (V, E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

Proof:

- Line 16 of BFS sets $\pi[v] = u$ if and only if $(u\ v) \in E$ and $\delta(s, v) < \infty$ that is, if v is reachable from s and thus V_π consists of the vertices in V reachable from s.
- Since G_π forms a tree, it contains a unique path from s to each vertex in V_π .
- By applying previous Theorem inductively, we conclude that every such path is a shortest path.
- The procedure in upcoming slide prints out the vertices on a shortest path from s to v, assuming that BFS has already been run to compute the shortest-path tree.

Print Path

PRINT-PATH (G, s, v)

```
1  if v = s
2    then print s
3    else if π[v] = NIL
4           then print “no path from s to v exists
5           else PRINT-PATH (G, s, π[v])
6                print v
```

https://studyc.info/

Lecture 29 Proof (Breadth First Search Algorithm) and Depth First Search

Theorem (Correctness of BFS)

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $d[v] = \delta(s,v)$ for all $v \in V$.

Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by edge $(\pi[v], v)$.

Proof:

Assume, for the purpose of contradiction, that some vertex receives a d value not equal to its shortest-path distance. Let v be the vertex with minimum $\delta(s, v)$ that receives such an incorrect d value; clearly $v \neq s$. By Lemma 22.2, $d[v] \geq \delta(s, v)$, and thus we have that $d[v] > \delta(s, v)$. Vertex v must be reachable from s, for if it is not, then $\delta(s, v) = \infty \geq d[v]$.

Let u be the vertex immediately preceding v on a shortest path from s to v, so that

$\delta(s, v) = \delta(s, u) + 1$.

Because $\delta(s, u) < \delta(s, v)$, and because of how we chose v, we have $d[u] = \delta(s, u)$.

Putting these properties together, we have

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 \qquad (22.1)$$

Now consider the time when BFS chooses to dequeue vertex u from Q in line 11.

At this time, vertex v is, white, gray, or black.

We shall show that in each of these cases, we derive a contradiction to inequality (22.1).

If v is white, then line 15 sets $d[v] = d[u] + 1$, contradicting inequality (22.1).

If v is black, then it was already removed from the queue and, by Corollary 22.4, we have $d[v] \leq d[u]$, again contradicting inequality (22.1).

If v is gray, then it was painted gray upon dequeuing some vertex w, which was removed from Q earlier than u and, $d[v] = d[w] + 1$.

By Corollary 22.4, however, $d[w] \leq d[u]$, and so we have $d[v] \leq d[u] + 1$, once again contradicting inequality (22.1).

Thus we conclude that $d[v] = \delta(s, v)$ for all $v \in V$. All vertices reachable from s must be discovered, if they were not, they would have infinite d values.

To conclude the proof of the theorem, observe that if $\pi[v] = u$, then $d[v] = d[u] + 1$.

https://studyc.info/

Thus, we can obtain a shortest path from s to v by taking a shortest path from s to $\pi[v]$ and then traversing the edge $(\pi[v], v)$

Depth First Search

- The predecessor subgraph of a depth-first search forms a **depth-first forest** composed of several **depth-first trees** defined as
  $G_\pi = (V_\pi, E_\pi)$, where
  $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$
  the edges in E_π are called **tree edges.**
- Each vertex is initially white
  - It is grayed when it is **discovered** in the search, and
  - It is blackened when it is **finished**, that is, when its adjacency list has been examined completely.

Discovery and Finish Times

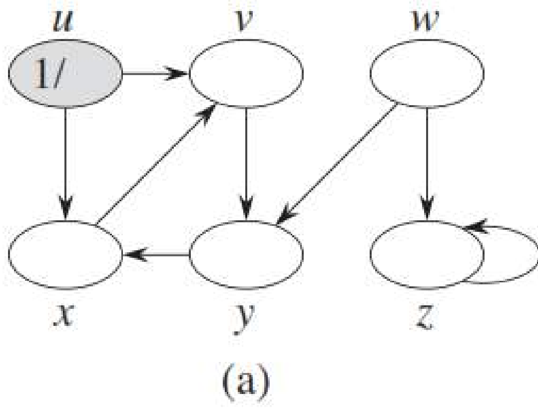
- It guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.
- It **timestamps** each vertex
  - the first timestamp $d[v]$ records when v is first discovered (and grayed), and
  - the second timestamp $f[v]$ records when the search finishes examining v's adjacency list (and blackens v).
- For every vertex u $\quad d[u] < f[u]$

Algorithm: Depth First Search

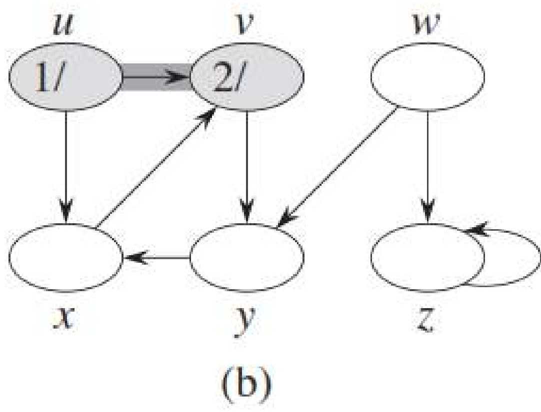
DFS(G)	DFS-Visit(u)
1 **for** each vertex $u \in V[G]$	1 $color[u] \leftarrow$ GRAY
2 **do** $color[u] \leftarrow$ WHITE	2 $time \leftarrow time + 1$
3 $\pi[u] \leftarrow$ NIL	3 $d[u] \leftarrow time$
4 $time \leftarrow 0$	4 **for** each $v \in Adj[u]$
5 **for** each vertex $u \in V[G]$	5 **do if** $color[v]$ = WHITE
6 **do if** $color[u]$ = WHITE	6 **then** $\pi[v] \leftarrow u$
7 **then** DFS-Visit (u)	7 DFS-Visit (v)
	8 $color[u] \leftarrow$ BLACK
	9 $f[u] \leftarrow time \leftarrow time + 1$

Running Time: $\Theta(V + E)$

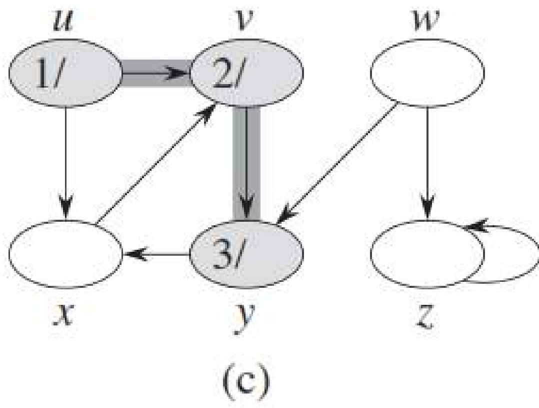
https://studyc.info/



(a)

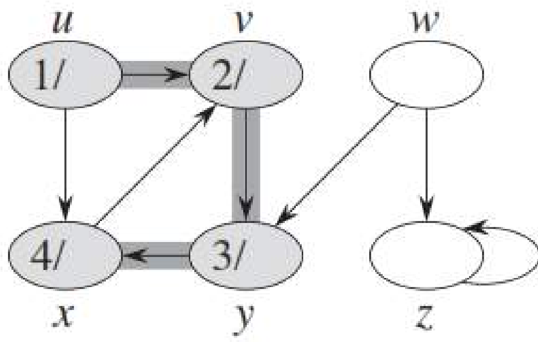


(b)

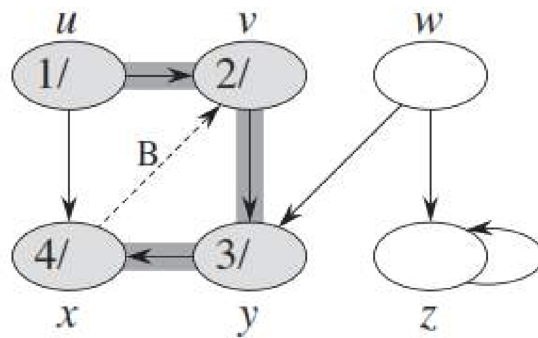


(c)

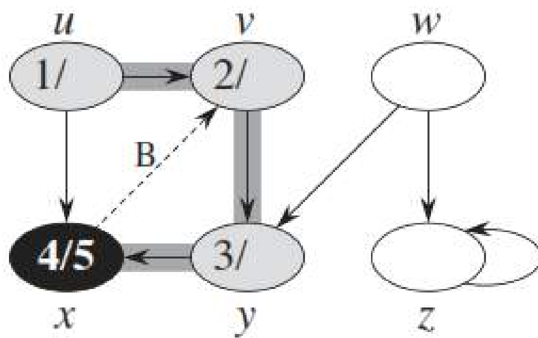
https://studyc.info/



(d)

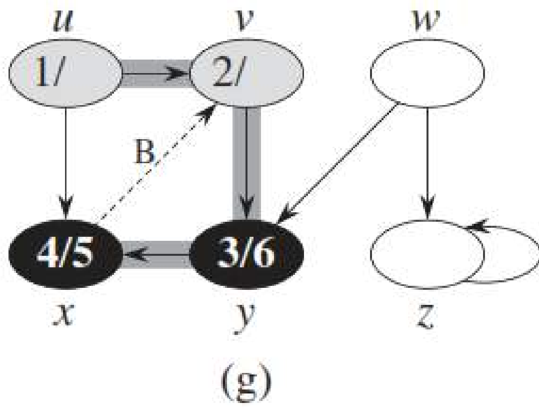


(e)

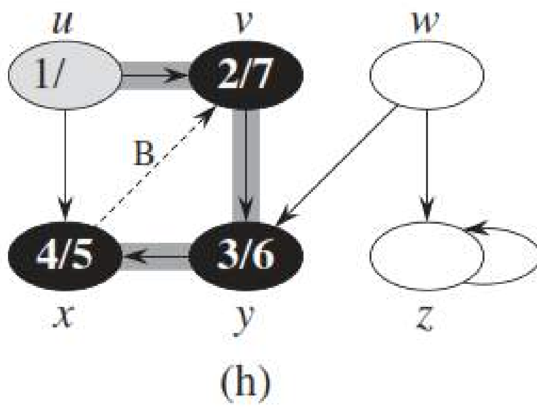


(f)

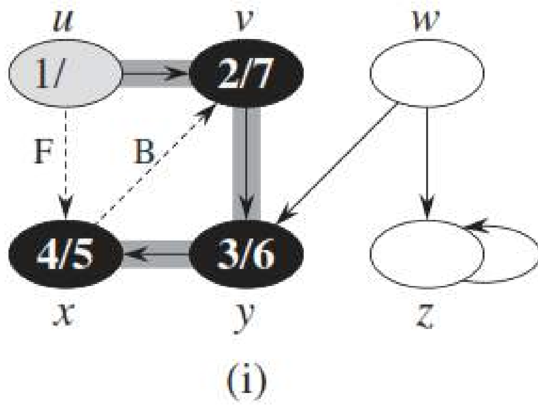
https://studyc.info/



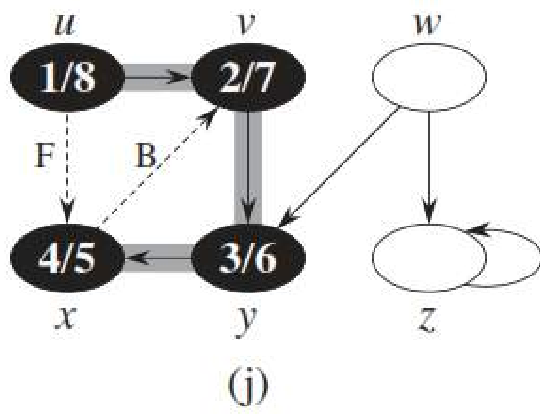
(g)



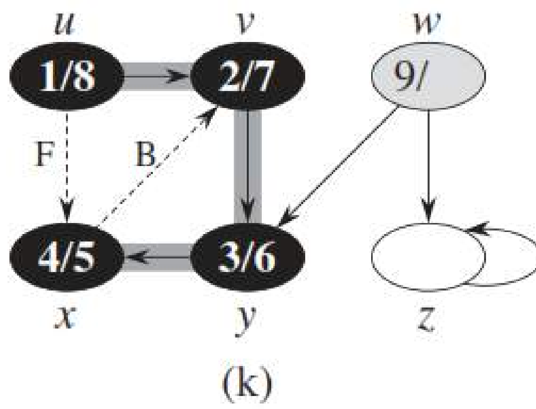
(h)



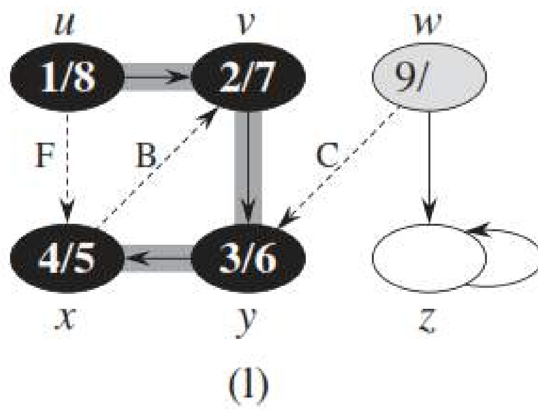
(i)



(j)

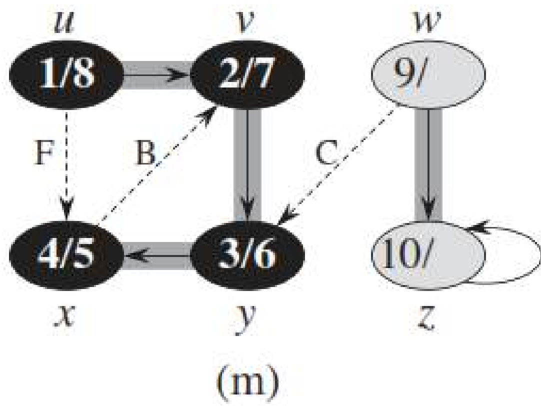


(k)

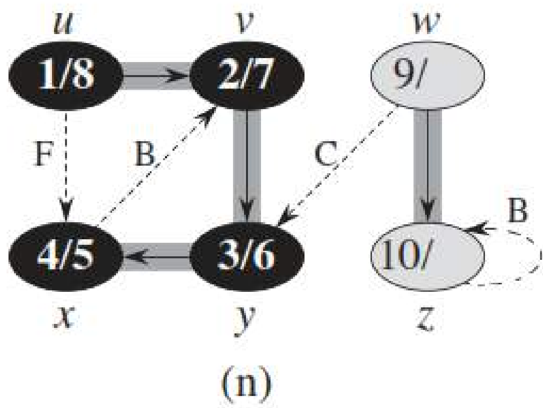


(l)

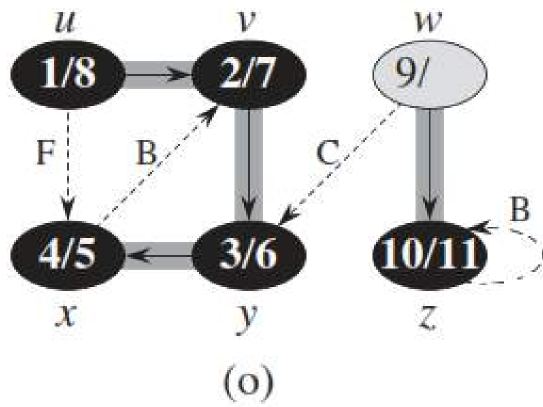
https://studyc.info/



(m)

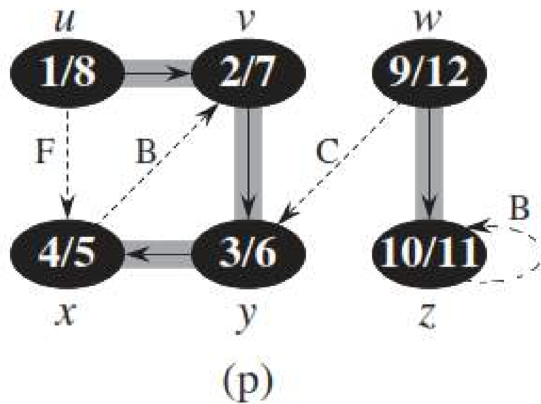


(n)



(o)

https://studyc.info/

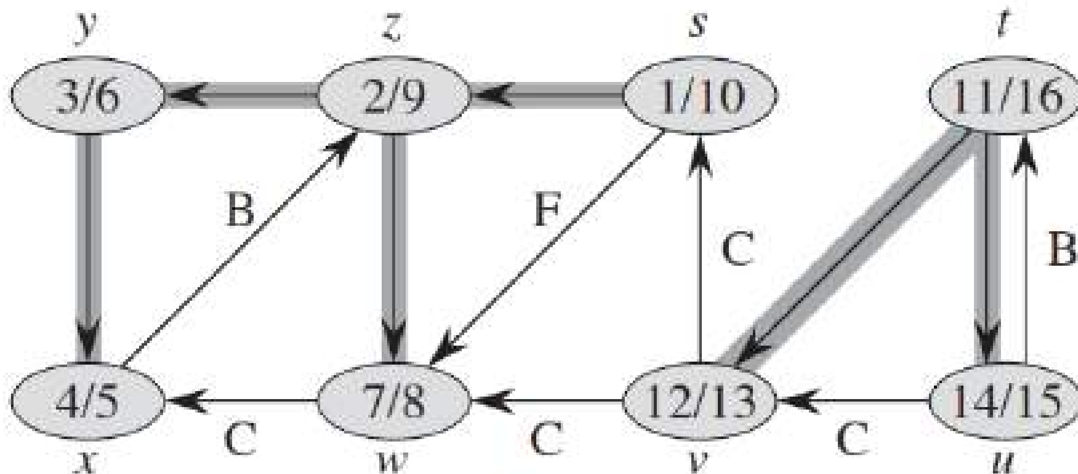


(p)

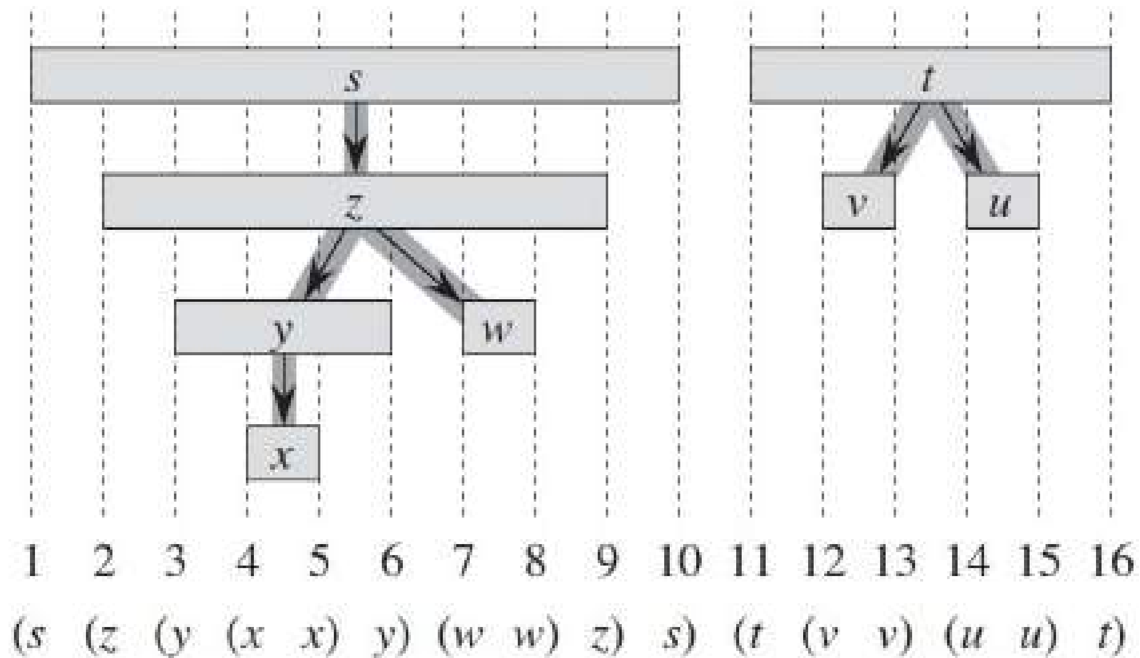
Properties of Depth First Search

- It yields valuable information about structure of a graph.
  - Predecessor subgraph G_π does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT.
- Discovery and finishing times have **parenthesis structure**.
  - If we represent the discovery of vertex *u* with a left parenthesis "(*u*" and represent its finishing by a right parenthesis "*u*)", then
  - history of discoveries and finishing makes well-formed expression in a sense that parentheses properly nested.

Parenthesis Structure



https://studyc.info/



Parenthesis Theorem

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

1. the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
2. the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in a depth-first tree, or
3. the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in a depth-first tree.

Proof

- We begin with case in which $d[u] < d[v]$.
- There are two sub-cases, either

  $d[v] < f[u]$ or $d[v] > f[u]$.

Case 1

- $d[v] < f[u] \Rightarrow v$ discovered while u was still gray.
- This means v is a descendant of u.
- Since v was discovered more recently than u, all of its outgoing edges are explored, and v is finished, before search finishes u.
- *Hence* $d[u] < d[v] <$ f(v) < f(u) (part 3 is proved)

https://studyc.info/

Case 2

- $d[u] < d[v]$ (supposed)
- and $f[u] < d[v]$ (by case 2)
- Hence intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ disjoint.
- Because intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.
- Now if we suppose $d[v] < d[u]$, then again either
- Intervals will be disjoint OR
- Interval of v will contain interval of u.

Corollary (Nesting of Descendants' Intervals)

Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $d[u] < d[v] < f[v] < f[u]$

Proof

- Immediate from the above Theorem

Classification of Edges

The depth-first search can be used to classify the edges of the input graph $G = (V, E)$.

1. **Tree edges**
   - These are edges in the depth-first forest G_π.
   - Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. **Back edges**
   - those edges (u, v) connecting a vertex u to an ancestor v in a depth first tree.
   - Self-loops, which may occur in directed graphs, are considered to be back edges.
3. **Forward edges**
   - Those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. **Cross edges**
   - These are all other edges.
   - They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or
   - they can go between vertices in different depth-first trees.

Theorem

https://studyc.info/

In a depth-first search of an undirected graph *G*, every edge of *G* is either a tree edge or back edge.

<u>Proof</u>

- Let (u, v) be an arbitrary edge of G, and suppose without loss of generality that d[u] < d[v].
- Then, *v* must be discovered and finished before we finish *u* (while *u* is gray), since *v* is on *u*'s adjacency list.
- If the edge (*u*, *v*) is explored first in direction from *u* to *v*, then *v* is undiscovered (white) until that time, for otherwise we would have explored this edge already in the direction from *v* to *u*.
- Thus, (*u*, *v*) becomes a tree edge.
- If (*u*, *v*) is explored first in the direction from *v* to *u*, then (*u*, *v*) is a back edge, since *u* is still gray at the time the edge is first explored.

https://studyc.info/

Lecture 30 Proof (White Path Theorem) & Applications of Depth First Search

Algorithm: Depth First Search

DFS(G)	DFS-Visit(u)
1 **for** each vertex $u \in V[G]$	1 $color[u] \leftarrow$ GRAY
2 **do** $color[u] \leftarrow$ WHITE	4 $time \leftarrow time + 1$
3 $\pi[u] \leftarrow$ NIL	5 $d[u] \leftarrow time$
4 $time \leftarrow 0$	4 **for** each $v \in Adj[u]$
5 **for** each vertex $u \in V[G]$	5 **do if** $color[v]$ = WHITE
6 **do if** $color[u]$ = WHITE	6 **then** $\pi[v] \leftarrow u$
8 **then** DFS-Visit (u)	7 DFS-Visit (v)
	8 $color[u] \leftarrow$ BLACK
	9 $f[u] \leftarrow time \leftarrow time + 1$

Running Time: $\Theta(V + E)$

Theorem: White-Path Theorem

In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $d[u]$ that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

Proof:

- Assume that v is a descendant of u.
- Let w be any vertex on the path between u and v in depth-first tree, so that w is a descendant of u.
- As $d[u] < d[w]$, and so w is white at time $d[u]$.

Second part is proved by contradiction

- Suppose that vertex v is reachable from u along a path of white vertices at time d[u], but v does not become a descendant of u in the depth-first tree.
- Without loss of generality, assume that every other vertex along the path becomes a descendant of u.
- (Otherwise, let v be the closest vertex to u along the path that doesn't become a descendant of u.)
- Let w be predecessor of v in the path, so that w is a descendant of u (w, u may be same) by Corollary above

  $f[w] \leq f[u]$. (1)
- Note v must be discovered after u is discovered,

$d[u] < d[v]$ (2)

- but v must be discovered before w is finished.

$d[v] < f[w]$ (3)

- Therefore, by (1), (2) and (3)

$d[u] < d[v] < f[w] \leq f[u]$.

https://studyc.info/

- Above Theorem implies that interval $[d[v], f[v]]$ is contained entirely within interval $[d[u], f[u]]$.
- By Corollary above, v must be a descendant of u.

Topological Sort

- A Topological Sort of a directed acyclic graph, or a "dag" $G = (V, E)$ is a linear ordering of all its vertices such that
  - if G contains an edge (u, v), then u appears before v in the ordering.
- It is ordering of its vertices along a horizontal line so that all directed edges go from left to right
- The depth-first search can be used to perform a topological sort of a dag.

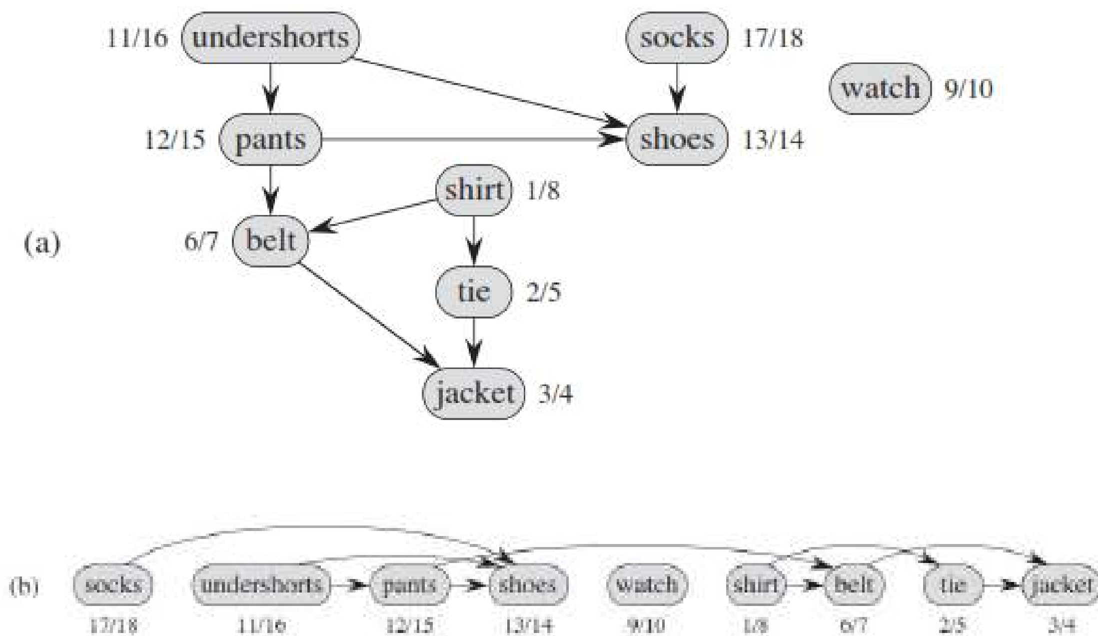
Algorithm

TOPOLOGICAL-SORT (*G*)

1. Call DFS(G) to compute $f[v]$ of each vertex $v \in V$.
2. Set an empty linked list $L = \emptyset$.
3. When a vertex v is colored black, assign it $f(v)$.
4. Insert v onto the front of the linked list, $L = \{v\}.L$.
5. **return** the linked list.
6. The rank of each node is its position in the linked list started from the head of the list.

Running Time: $\Theta(V + E)$

Example:



Lemma: A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

https://studyc.info/

Proof

⇦: G is acyclic.

- Suppose that there is a back edge (u, v).
- Then, vertex v is an ancestor of u in DF forest.
- There is thus a path from v to u in G, and the back edge (u, v) completes a cycle.
- G is cyclic and hence a contradiction,
- Our supposition is wrong and
- Hence G has no back edge

⇦ : If DFS yields no back edges G has no cycle

We prove it by contra positive

- We prove that if G contains a cycle c *the* DFS of G yields a back edge.
- Let G has a cycle c.
- Let v be the first vertex to be discovered in c, and let (u, v) be the preceding edge in c.
- At time $d[v]$, the vertices of c form a path of white vertices from v to u.
- By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest. Therefore, (u, v) is a back edge.

Theorem

TOPOLOGICAL-SORT (G) produces a topological sort of a directed acyclic graph G .

Proof

- Let DFS is run on G to determine finishing times.
- It sufficient to show that for any two distinct $u, v \in V$, if there is an edge in G from u to v, then $f[v] < f[u]$
- Consider any edge (u, v) explored by DFS(G).
- When (u, v) is explored, v *is* gray, white or black

Case 1

- v is gray. v is ancestor of u. (u, v) would be a back edge. It contradicts the above Lemma.

Case 2

- If v is white, it becomes a descendant of u, and hence $f[v] < f[u]$.

Case 3

- If v is black, it has already been finished, so that $f[v]$ has already been set.
- Because we are still exploring from u, we have yet to assign a timestamp to $f[u]$ to u, and so once we do, we will have $f[v] < f[u]$ as well.

https://studyc.info/

Thus, for any edge (u, v) in the dag, we have $f[v] < f[u]$. It proves the theorem.
SCC

Strongly Connected Components

A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have

- $u \leadsto v$, *v is* reachable from u.
- $v \leadsto u$; *u is* reachable from v.
- The depth-first search can be used in decomposing a directed graph into its strongly connected components.

Transpose of a graph

- The **strongly connected components** of a graph $G = (V, E)$ uses the transpose of G, which is defined as

  $G^T = (V, E^T)$, where

  $E^T = \{(u, v) : (v, u) \in E\}$

  E^T consists of the edges of G with reversed directions.

- G and G^T have exactly the same strongly connected components
  - u and v are reachable from each other in G if and only if they are reachable from each other in G^T.

Algorithm: Strongly connected components

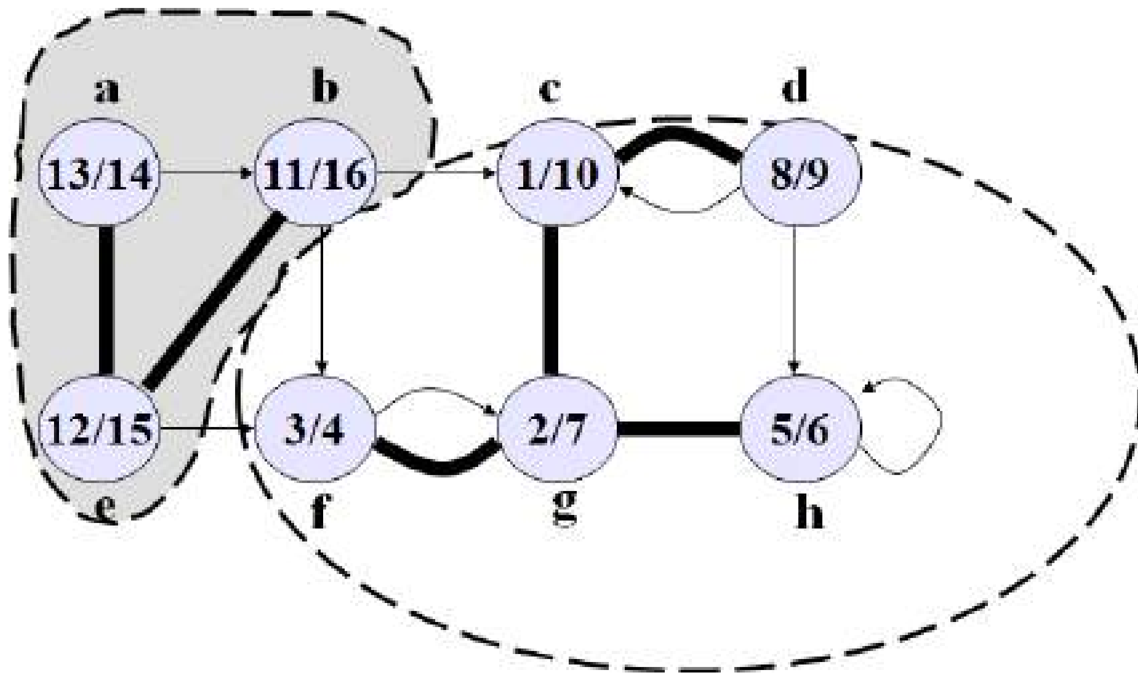
STRONGLY-CONNECTED-COMPONENTS (*G*)

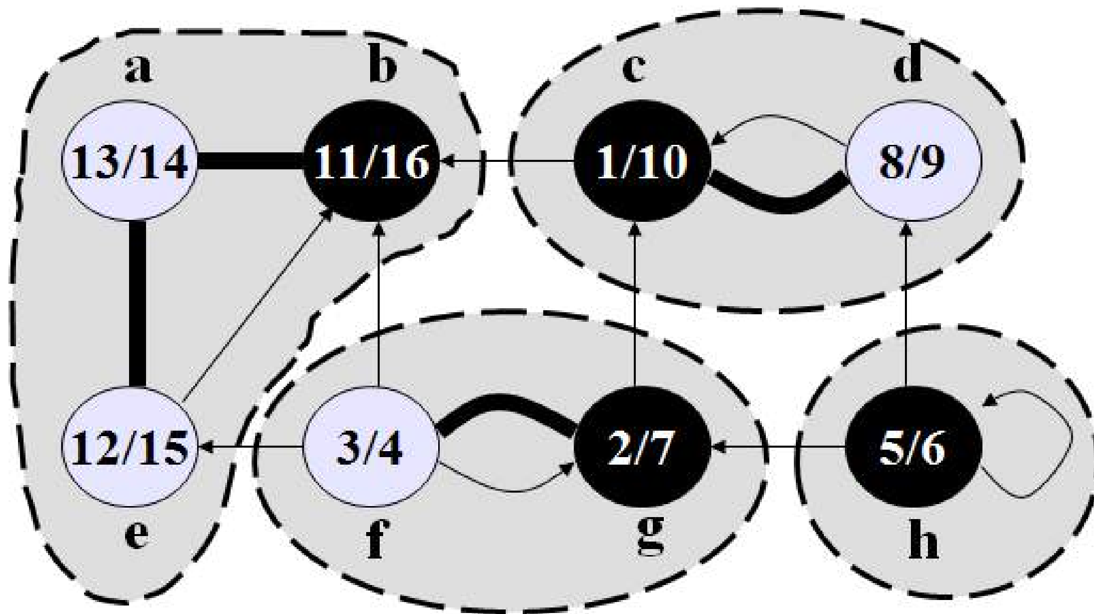
1. call DFS(G), to compute the finish time $f[u]$ of each vertex u
2. compute G^T.
3. call DFS (G^T), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$. (as computed in line 1)
4. Output of the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component.

Running Time: $\Theta(V + E)$

Example: Strongly connected components

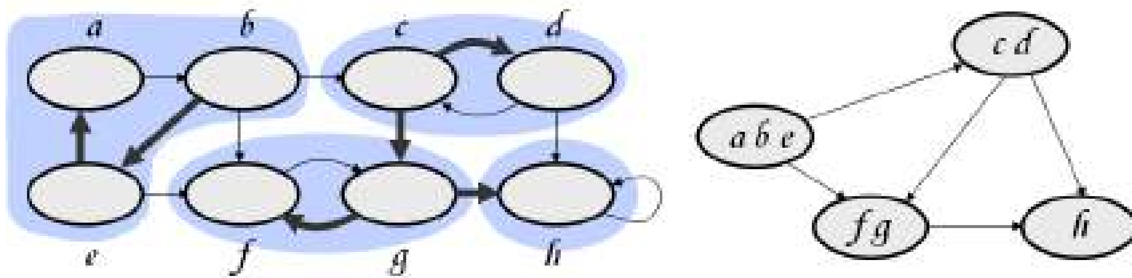
https://studyc.info/


a
b
c
d
13/14
11/16
1/10
8/9
12/15
3/4
2/7
5/6
e
f
g
h


a
b
c
d
13/14
11/16
1/10
8/9
12/15
3/4
2/7
5/6
e
f
g
h

Lecture 31 Backtracking and Branch & Bound Algorithms

Component Graph



The **component graph** $G^{SCC} = (V^{SCC}, E^{SCC})$

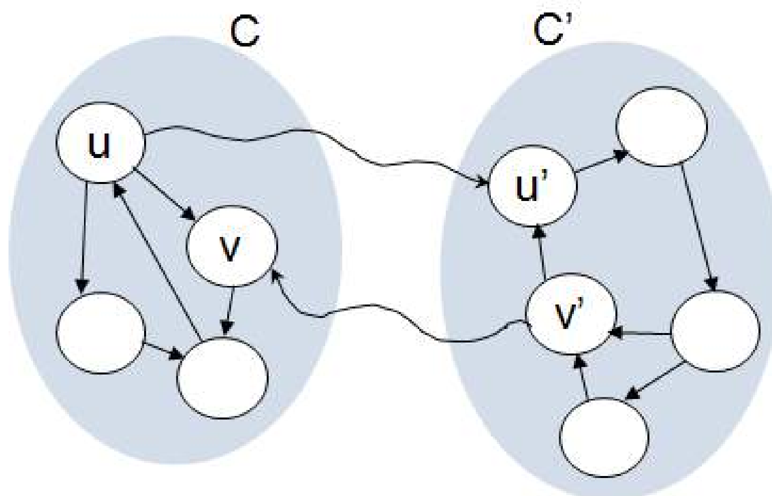
$V^{SCC} = \{v_1, v_2, \ldots, v_k\}$, where v_i corresponds to each strongly connected component C_i

There is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and $y \in C_j$

The component graph is a DAG

Lemma 1: Let C and C' be distinct SCC's in G
Let $u, v \in C$, and $u', v' \in C'$
Suppose there is a path $u \rightsquigarrow u'$ in G
Then there cannot also be a path $v' \rightsquigarrow v$ in G.

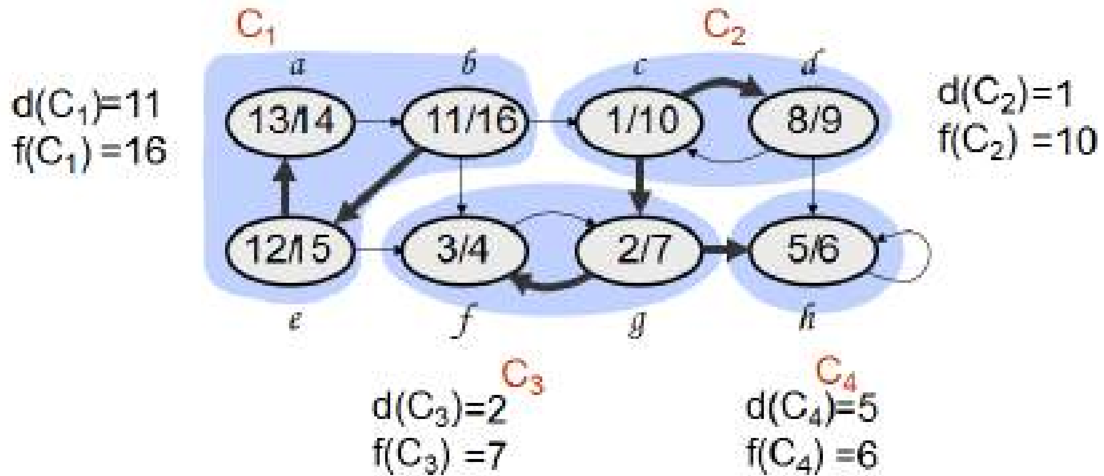
Proof:
Suppose there is path $v' \rightsquigarrow v$
There exists $u \rightsquigarrow u' \rightsquigarrow v'$
There exists $v' \rightsquigarrow v \rightsquigarrow u$
u and v' are reachable from each other, so they are not in separate SCC's: contradiction!



Notations: Vertices to SCC

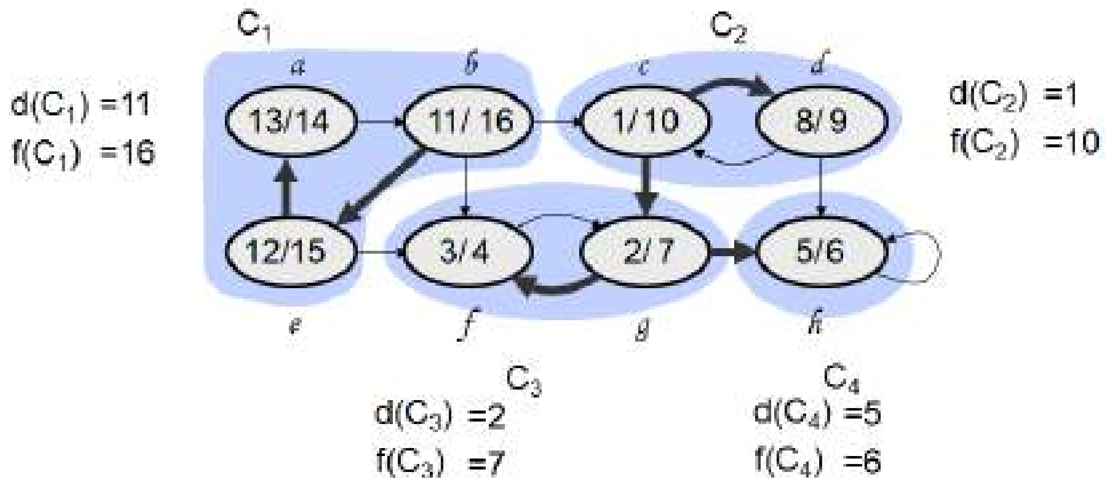
https://studyc.info/

- d and f times of vertices of SCC
- Let $U \subseteq V$, a SCC
  - $d(U) = \min_{u \in U} \{ d[u] \}$ (earliest discovery time)
  - $f(U) = \max_{u \in U} \{ f[u] \}$ (latest finishing time)



Lemma 2:

- Let C and C' be distinct SCCs in a directed graph G = (V, E). If there is an edge (u, v) ∈ E, where u ∈ C and v ∈ C' then f(C) > f(C').



Proof

- Consider C1 and C2, connected by edge (u, v)
- There are two cases, depending on which strongly connected component, *C* or *C′*, had the first discovered vertex during the depth-first search

Case 1

https://studyc.info/

- If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $d[x]$, all vertices in C and C' are white.
- There is a path in G from x to each vertex in C consisting only of white vertices.
- Because $(u, v) \in E$, for any vertex $w \in C'$, there is also a path at time $d[x]$ from x to w in G consisting only of white vertices: $x \rightsquigarrow u \rightsquigarrow v \rightsquigarrow w$.
- By the white-path theorem, all vertices in C and C' become descendants of x in the depth-first tree. By Corollary, $f[x] = f(C) > f(C')$.

Case 2

- $d(C) > d(C')$ (supposition)
- Now $(u, v) \in E$, where $u \in C$ and $v \in C'$ (given)
- Let y be the first vertex discovered in C'.
- At time $d[y]$, all vertices in C' are white and there is a path in G from y to each vertex in C' consisting only of white vertices.
- By the white-path theorem, all vertices in C' become descendants of y in the depth-first tree, and by Corollary, $f[y] = f(C')$.
- At time $d[y]$, all vertices in C are white. Since there is an edge (u, v) from C to C', Lemma implies that there cannot be a path from C' to C.
- Hence, no vertex in C is reachable from y.
- At time $f[y]$, therefore, all vertices in C are still white.
- Thus, for any vertex $w \in C$, we have $f[w] > f[y]$, which implies that $f(C) > f(C')$.

Corollary

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$

Proof

- Since $(u, v) \in E^T$, we have $(v, u) \in E$.
- Since strongly connected components of G and G^T are same, Lemma implies that $f(C) < f(C')$.

Theorem: Correctness of SCC Algorithm

STRONGLY-CONNECTED-COMPONENTS (G) correctly computes SCCs of a directed graph G.

Proof

- We argue by induction on number of DF trees of G^T that "vertices of each tree form a SCC".
- The basis for induction, when $k = 0$, is trivial.
- Inductive hypothesis is that, first k trees produced by DFS of G^T are strongly connected components.
- Now we prove for $(k+1)^{st}$ tree produced from G^T, i.e. vertices of this tree form a SCC.
- Let root of this tree be u, which is in SCC C.
- Now, $f[u] = f(C) > f(C')$, $\forall$ C' yet to be visited and $\neq C$
- By inductive hypothesis, at the time search visits u, all other vertices of C are white.

https://studyc.info/

- By white-path theorem, all other vertices of C are descendants of u in its DF tree.
- Moreover, by inductive hypothesis and by Corollary above, any edges in G^T, that leave *C* must be, to SCCs that have already been visited.
- Thus, no vertex in any SCC other than *C* will be a descendant of *u* during the DFS of G^T.
- Thus, vertices of DF tree in G^T rooted at *u* form exactly one SCC.

Why BackTracking?

When the graph is too large then Depth and breadth-first techniques are infeasible.

In this approach if node searched for is found out that cannot exist in the branch then return back to previous step and continue the search to find the required node.

What is backtracking?

- Backtracking is refinement of Brute Force approach
- It is a technique of constraint satisfaction problems
- Constraint satisfaction problems are with complete solution, where elements order does not matter.
- In backtracking, multiple solutions can be eliminated without examining, by using specific properties
- Backtracking closely related to combinatorial search
- There must be the proper hierarchy in produces
- When a node is rejected, whole sub-tree rejected, and we backtrack to the ancestor of node.
- Method is not very popular, in the worst case, it takes an exponential amount of time to complete.

Solution Spaces

- Solutions are represented by vectors $(v_1, ..., v_m)$ of values. If S_i is the **domain** of v_i, then $S_1 \times ... \times S_m$ is the **solution space** of the problem.

- **Approach**

  – It starts with an empty vector.

  – At each stage it extends a partial vector with a new value

  – Upon reaching a partial vector $(v_1, ..., v_i, v)$ which can't represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values.

General Algorithm: Solution Spaces

ALGORITHM try(v_1,...,v_i)
    IF (v_1,...,v_i) is a solution
        THEN RETURN (v_1,...,v_i)
    FOR each v DO

https://studyc.info/

IF $(v_1,...,v_i,v)$ is acceptable vector
THEN
sol = try$(v_1,...,v_i,v)$
THEN RETURN sol

Knapsack: Feasible Solutions

Partial solution is one in which only first *k* items have been considered.

- Solution has form $S_k = \{x_1, x_2, \ldots, x_k\}$, $1 \leq k < n$.
- The partial solution S_k is feasible if and only if

$$\sum_{i=1}^{k} w_i x_i \leq C$$

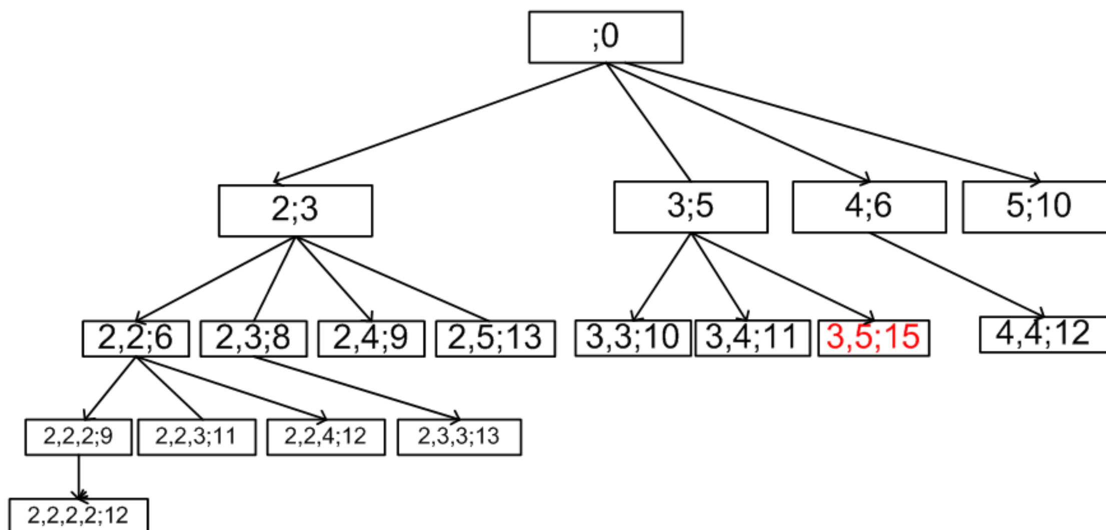
- If S_k is infeasible, then every possible complete solution containing S_k is also infeasible.

Knapsack Example: Backtracking

Maximum Capacity = 8

i	1	2	3	4
v_i	3	5	6	10
w_i	2	3	4	5

(2,2,3;11) means that two elements of each weight 2 and one element of weight 3 is with total value 11



Knapsack Algorithm: Backtracking

https://studyc.info/

```
BackTrack(i, r)                    \\ BackTrack(1, C)
b ← 0
{try each kind of item in tern}
for k ← i to n
        do
        if w(k) ≤ r then
                b ← max (b, v[k] + BackTrack(k, r - w[k]))
return b
```

https://studyc.info/

Lecture 32 Minimal Spanning Tree Problem

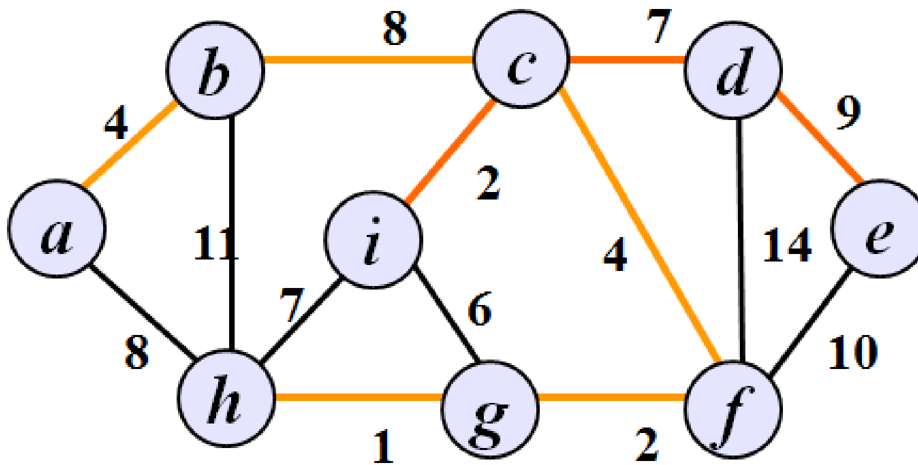
Minimum Spanning Tree

Given a graph G = (V, E) such that

- G is connected and undirected
- $w(u, v)$ weight of edge (u, v)
- T is a **Minimum Spanning Tree (MST)** of G if
- T is acyclic subset of E (T $\subseteq$ E)
- It connects all the vertices of G and
- Total weight, $w(T) = \sum_{(u,v) \in T} w(u,v)$ is minimized.

Example of MST

- Minimum spanning trees are not unique
  - If we replace (b, c) with (a, h), get a different spanning tree with the same cost
- MST have no cycles
  - We can take out an edge of
  - a cycle, and still have the
  - vertices connected while reducing the cost



Generic Solution to Compute MST

Minimum-spanning-tree problem: Find a MST for a connected, undirected graph, with a weight function associated with its edges

A generic solution:

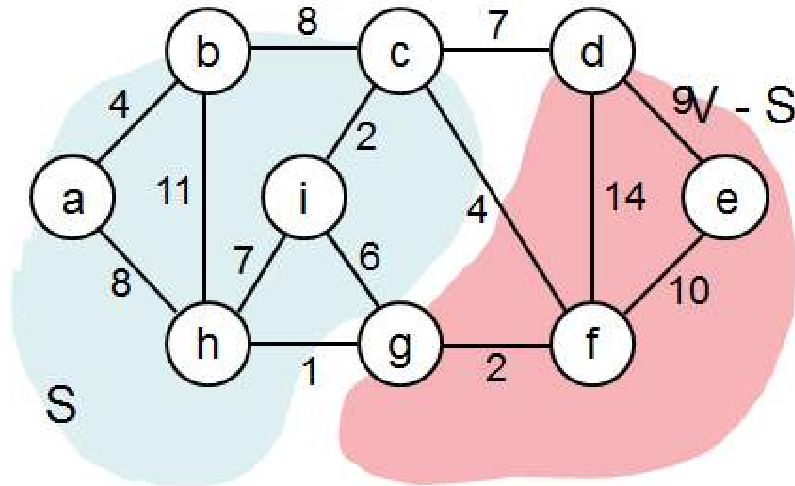
Build a set A of edges (initially empty)

Incrementally add edges to A such that they would belong to a MST

An edge (u, v) is **safe** for A $\Leftrightarrow$ A $\cup$ {(u, v)} is also a subset of some MST

https://studyc.info/

How to Find Safe Edge?



Let us look at edge (h, g)

- Is it safe for A initially?
- Let S ⊂ V be any set of vertices that includes h but not g (so that g is in V - S)
- In any MST, there has to be one edge (at least) that connects S with V - S
- Why not choose edge with minimum weight (h, g)

Generic Algorithm: Minimum Spanning Tree

GENERIC-MST(G, w)
1 $A \leftarrow \emptyset$
2 **while** A does not form a spanning tree
3 **do** find an edge (u, v) that is safe for A
4 $A \leftarrow A \cup \{(u, v)\}$
5 **return** A

Strategy: Growing Minimum Spanning Tree

- The algorithm uses greedy strategy which grows MST one edge at a time.
- Given a connected, undirected graph $G = (V, E)$ with a weight function $w : E \rightarrow R$
- Algorithm manages a set of edges A, maintaining loop invariant

Prior to each iteration, A is a subset of some MST

- An edge (u, v) is a **safe edge** for A such that $A \cup \{(u, v)\}$ is also a subset of some MST.

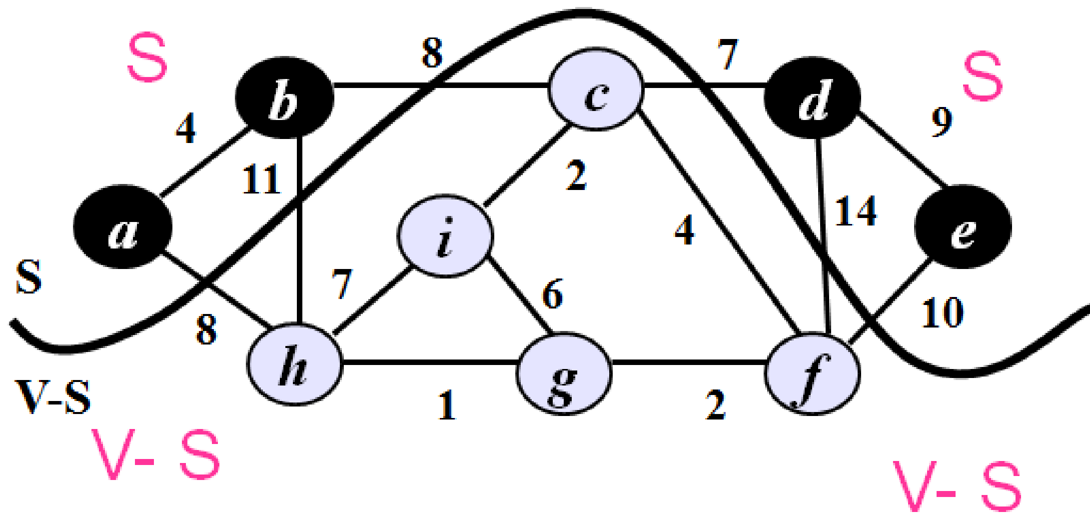
Algorithms, discussed here, to find safe edge are Kruskal's Algorithm and Prim's Algorithm.

Definitions (Kruskal's Algorithm)

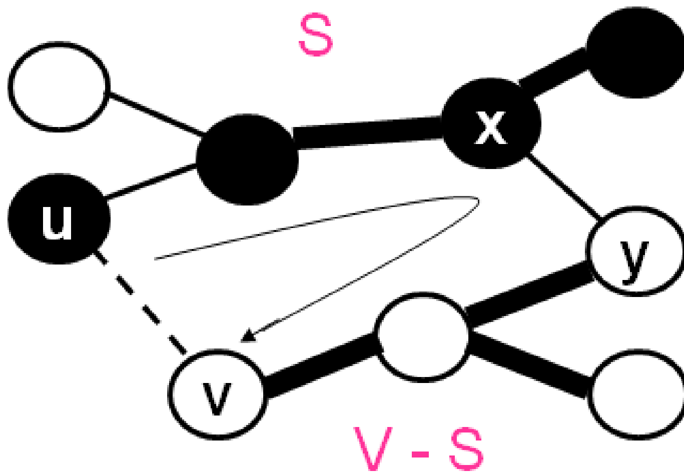
- A **cut** (S, V-S) of an undirected graph is a partition of V
- An edge **crosses** the cut (S, V-S) if one of its endpoints is in S and the other is in V-S.

https://studyc.info/

- A cut **respects** set A of edges if no edge in A crosses cut.
- An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.



Theorem (Kruskal's Algorithm)



Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.

Proof

- Let T be a minimum spanning tree that includes A *(edges of A are shaded)*
- Assume that T does not contain the light edge (u, v), since if it does, we are done.

Construction of another MST

https://studyc.info/

- For (u, v) ∉ T
- We construct another MST T′ that includes A ∪ {(u, v)} by cut-and-paste, and showing that (u, v) is a safe A.
- Since (u, v) crosses cut set (S, V-S) and
- (u, v) ∉ T,
- Hence there must be an edge (x, y) ∈ T which crosses the cut set
- By removing (x, y) breaks T into two components.
- Adding (u, v) reconnects them to form a new spanning tree T′ = T - {(x, y)} □ {(u, v)}.

Show that T′ is a minimum spanning tree.

- Since (u, v) is a light edge crossing (S, V - S) and (x, y) also crosses this cut, w(u, v) ≤ w(x, y).
- Hence, w(T′) = w(T) - w(x, y) + w(u, v) ≤ w(T).
- But T is a MST, so that w(T) ≤ w(T′); thus, T′ must be a minimum spanning tree also.

Show that (u, v) is safe for A: (u, v) can be part of MST

- Now (x, y) is not in A, because the cut respects A.
- Since A ⊆ T and (x, y) ∉ A ⇒ A ⊆T- {(x, y)}
- A ∪ {(u, v)} ⊆ T- {(x, y)} ∪ {(u, v)} = T'
- Since T' is an MST ⇒ (u, v) is safe for A

Kruskal's Algorithm

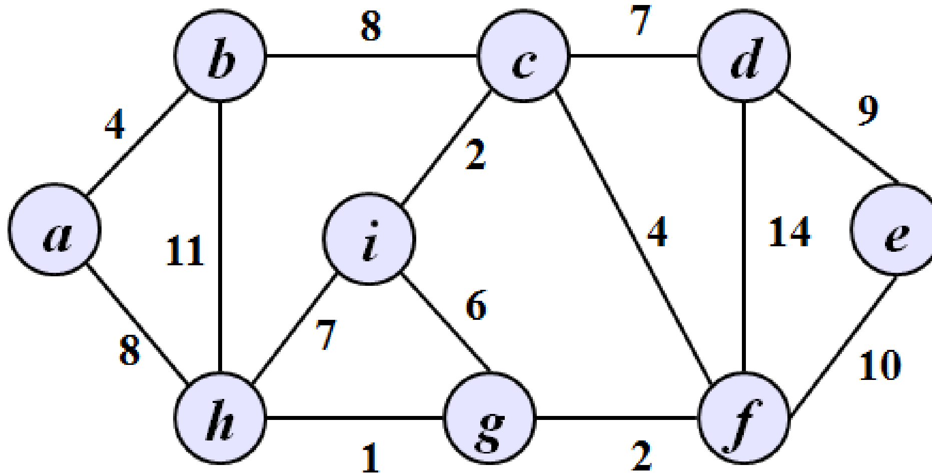
MST-KRUSKAL (*G*, *w*)

1 *A* ← Ø
2 **for** each vertex *v* ∈ *V*[*G*]
3   **do** MAKE-SET(*v*)
4 sort edges in non-decreasing order by weight *w*
5 **for** each (*u*, *v*) in non-decreasing order by weight
6   **do if** FIND-SET(*u*) ≠ FIND-SET(*v*)
7   **then** *A* ← *A* ∪ {(*u*, *v*)}
8   UNION (*u*, *v*)
9 **return** ***A***

Total Running time = O (E lg V),

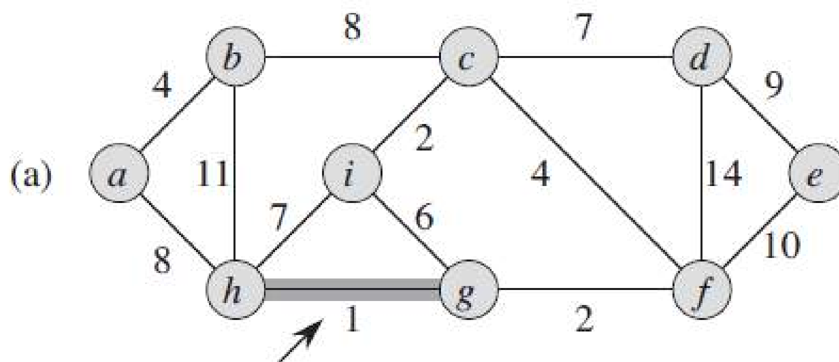
Kruskal's Algorithm

https://studyc.info/



Edges	Weight
(g, h)	1
(c, i)	2
(f, g)	2
(a, b)	4
(c, f)	4
(g, i)	6
(c, d)	7
(h, i)	7
(a, h)	8
(b, c)	8
(d, e)	9
(e, f)	10
(b h)	11
(d, f)	14

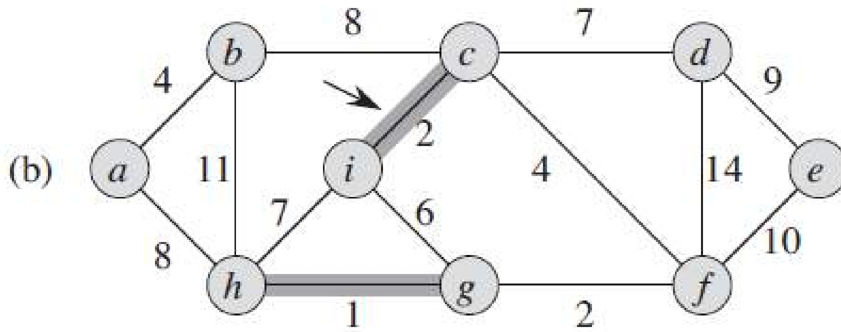
Initial sets = *{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}*



Initial sets = *{a}, {b}, {c}, {d}, {e}, {f}, {**g**}, {**h**}, {i}*

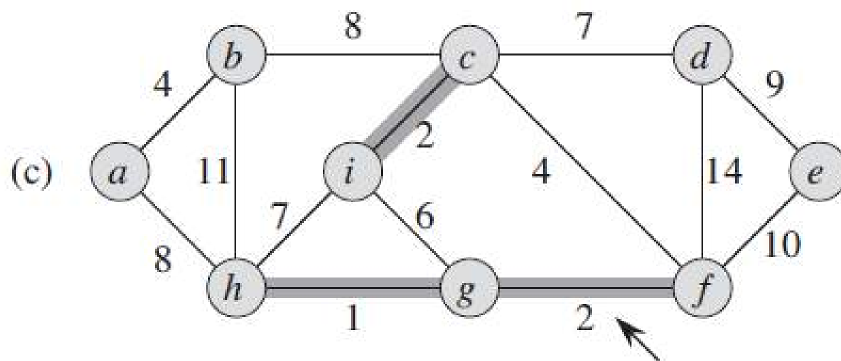
https://studyc.info/

Final sets = *{a}, {b}, {c}, {d}, {e}, {f},* {***g, h***}, *{i}*



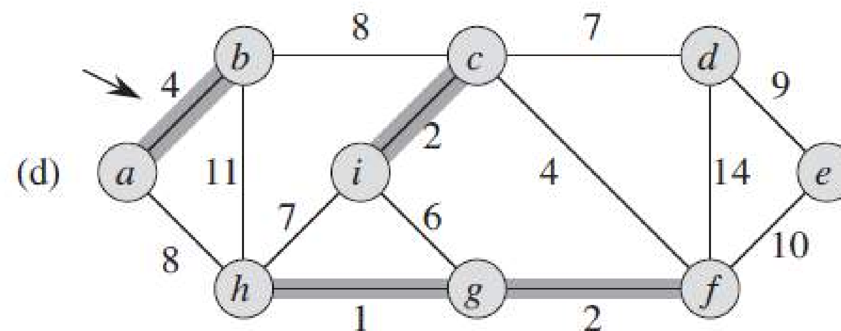
Initial sets = *{a}, {b},* {***c***}, *{d}, {e}, {f}, {g, h},* {***i***}

Final sets = *{a}, {b},* {***c, i***}, *{d}, {e}, {f}, {g, h}*



Initial sets = *{a}, {b}, {c, i}, {d}, {e},* {***f***}, {***g***, *h}*

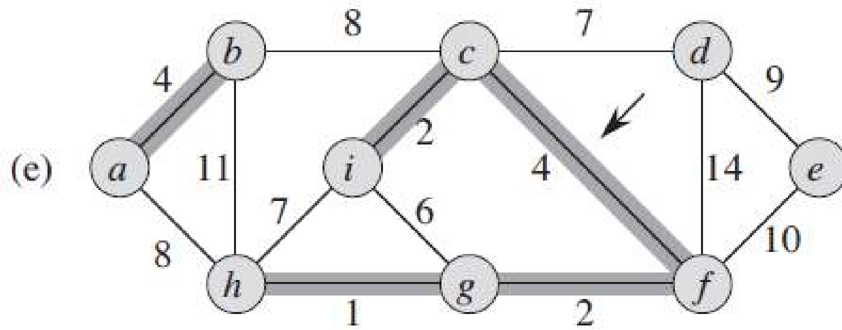
Final sets = *{a}, {b}, {c, i}, {d}, {e},* {***f, g, h***}



Initial sets = {***a***}, {***b***}, *{c, i}, {d}, {e}, {f, g, h}*

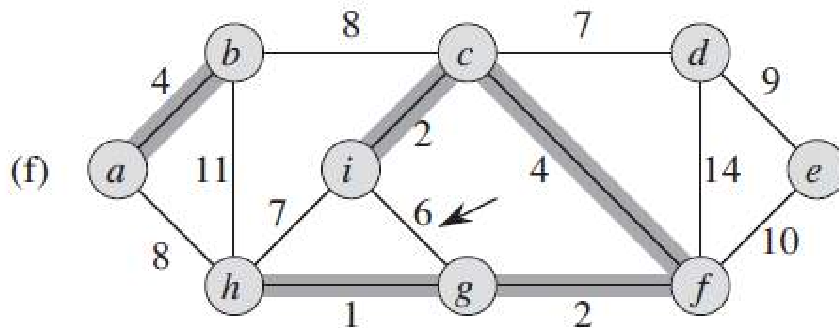
Final sets = {***a, b***}, *{c, i}, {d}, {e}, {f, g, h}*

https://studyc.info/



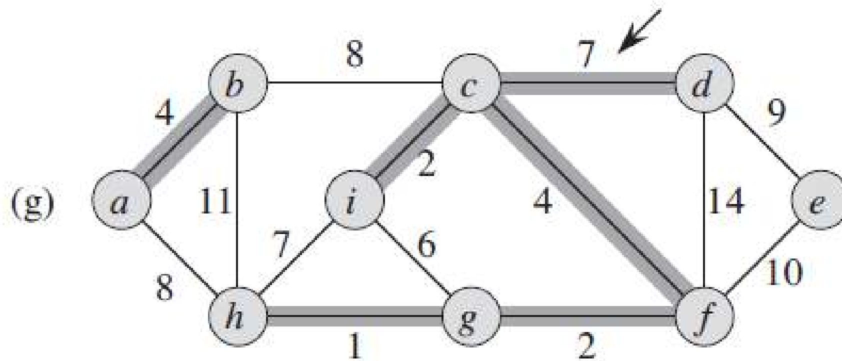
Initial sets = *{a, b}, {**c**, i}, {d}, {e}, {**f**, g, h}*

Final sets = *{a, b}, {**c, f, g, h , i**}, {d}, {e}*



Initial sets = *{a, b}, {c, f, **g,** h, **i**}, {d}, {e}*

Final sets = *{a, b}, {c, f, g, h, i}, {d}, {e}*

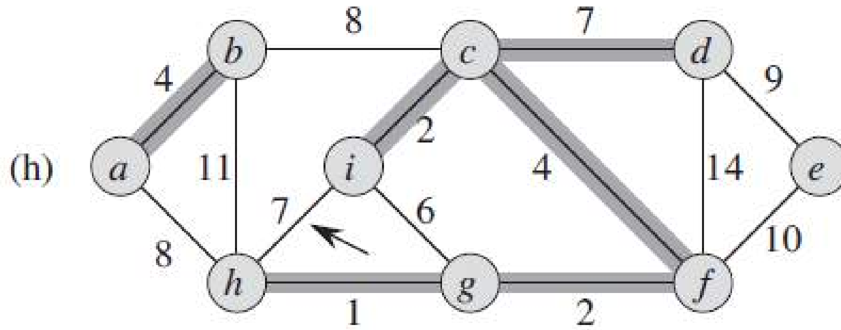


Initial sets = *{a, b}, {**c**, f, g, h, i}, {**d**}, {e}*

Final sets = *{a, b}, {**c, d, f, g, h, i**}, {e}*

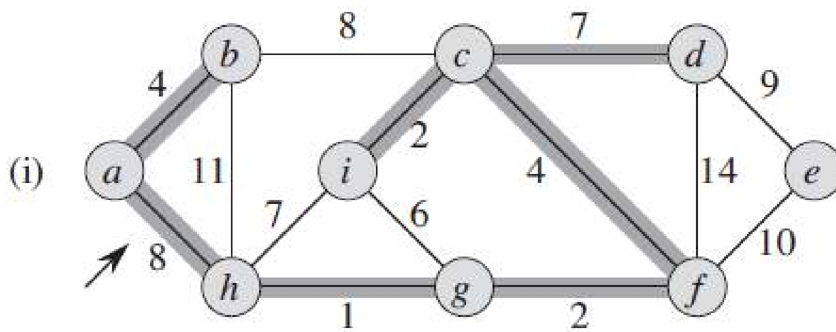
---

https://studyc.info/



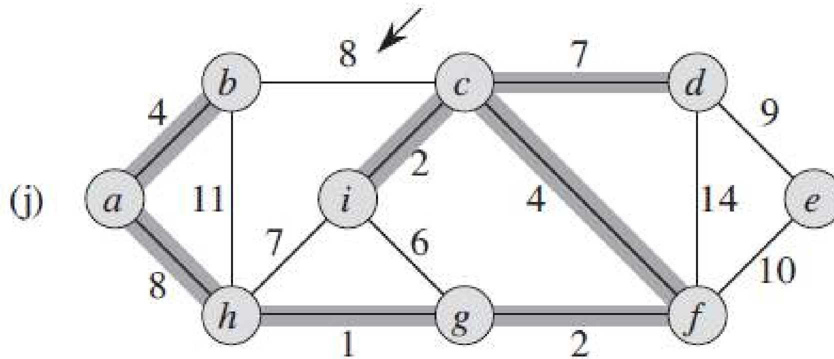
Initial sets = *{**a**, b}, {c, d, f, g, **h**, i}, {e}*

Final sets = *{**a, b, c, d, f, g, h , i**}, {e}*



Initial sets = *{**a**, b}, {c, d, f, g, **h**, i}, {e}*

Final sets = *{**a, b, c, d, f, g, h , i**}, {e}*

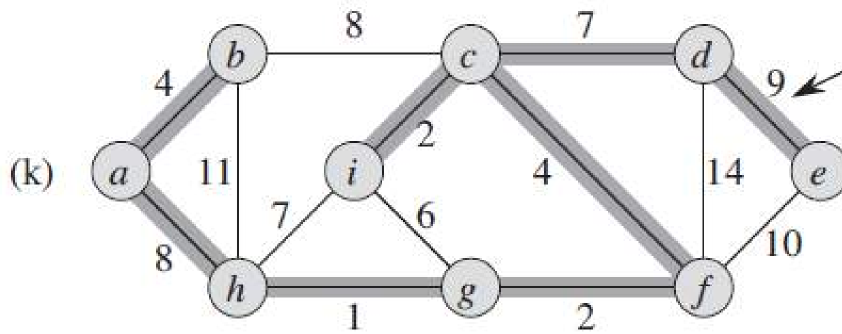


Initial sets = *{a, **b, c**, d, f, g, h , i}, {e}*

Final sets = *{a, b, c, d, f, g, h **,** i}, {e}*

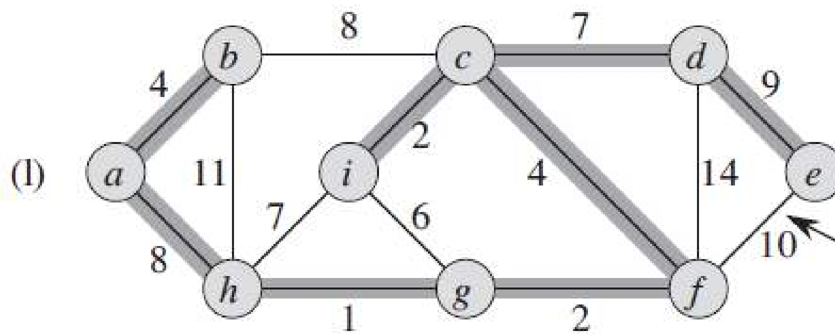
---

https://studyc.info/



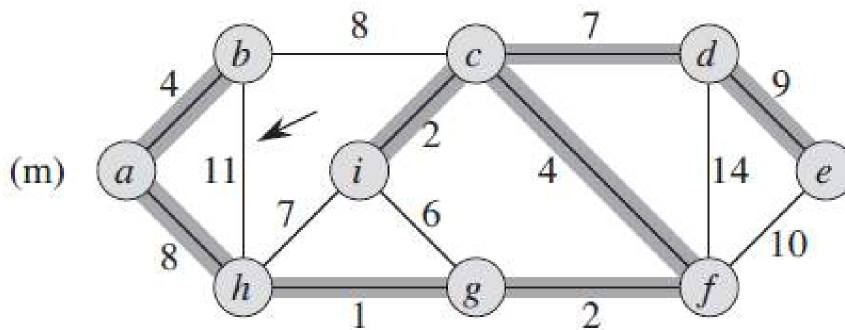
Initial sets = *{a, b, c,* ***d****, f, g, h , i}, {****e****}*

Final sets = *{****a, b, c, d, e, f, g, h , i****}*



Initial sets = *{a, b, c, d,* ***e, f****, g, h , i}*

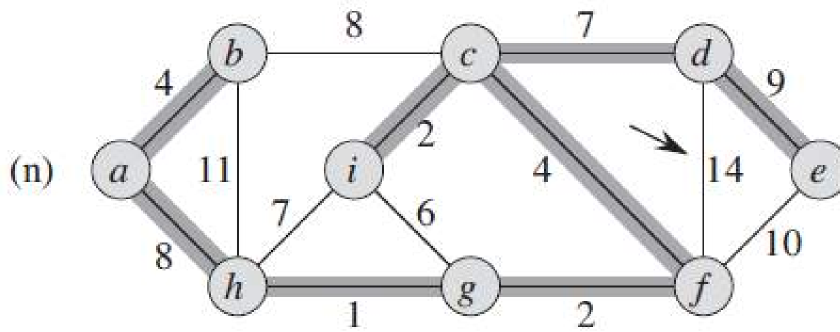
Final sets = *{a, b, c, d, e, f, g, h* **,** *i}*



Initial sets = *{a,* ***b****, c, d, e, f, g,* ***h*****,** *i}*

Final sets = *{a, b, c, d, e, f, g, h***,** *i}*

https://studyc.info/



Initial sets = *{a, b, c,* ***d****, e,* ***f****, g, h*, *i}*

Final sets = *{a, b, c, d, e, f, g, h* , *i}*

Correctness of Kruskal's Algorithm

- Used to determine the safe edge of GENERIC-MST
- The algorithm manages set of edges *A* which always form a single tree.
- The tree starts from an arbitrary vertex *r* and grows until tree spans all the vertices in *V*.
- At each step, a light edge is added to the tree *A* that connects *A* to an isolated vertex of G_A = (V, A)
- It is a greedy algorithm
  - At each step tree is augmented with an edge that contributes least possible amount to tree's weight
- Since vertices, of each edge considered, are in different sets hence no cycle is created.

Prim's Algorithm

MST-PRIM (*G*, *w*, *r*)

1 **for** each $u \in V[G]$
2 **do** $key[u] \leftarrow \infty$
3 $\pi[u] \leftarrow$ NIL
4 $key[r] \leftarrow 0$
5 $Q \leftarrow V[G]$
6 **while** $Q \neq \emptyset$
7 **do** $u \leftarrow$ EXTRACT-MIN(Q)
8 **for** each $v \in Adj[u]$
9 **do if** $v \in Q$ and $w(u, v) < key[v]$
10 **then** $\pi[v] \leftarrow u$
11 $key[v] \leftarrow w(u, v)$

Prim's Algorithm

- The performance depends on the implementation of min-priority queue *Q*.
- Using binary min-heap

Total Running time = O (E lg V)

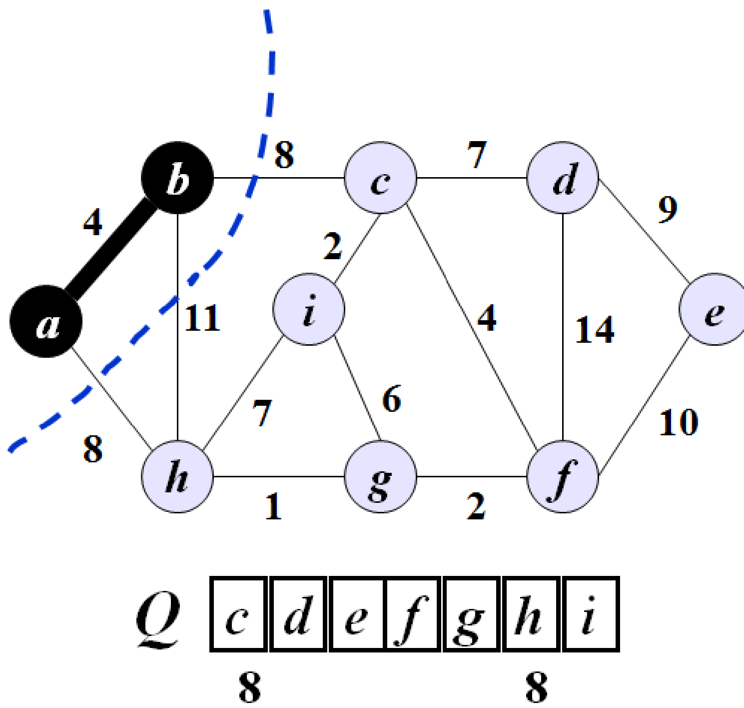
https://studyc.info/

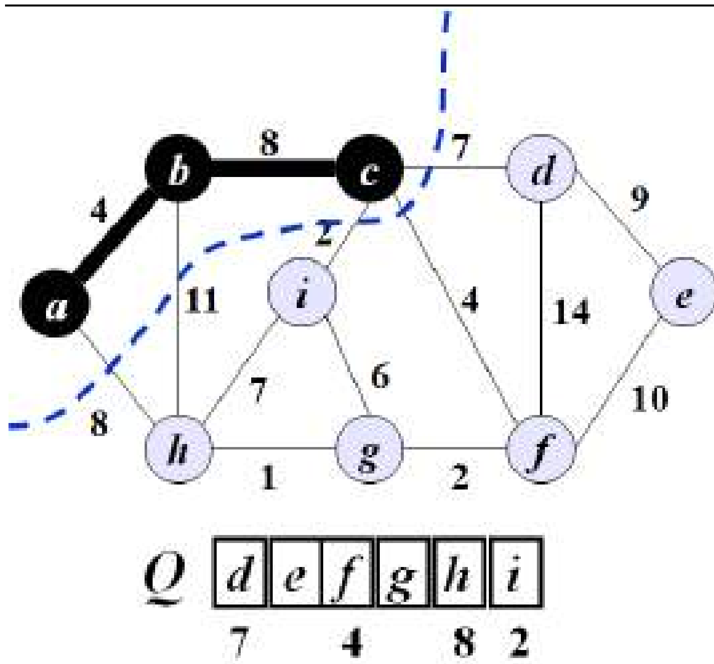
- Using fibonacci heaps

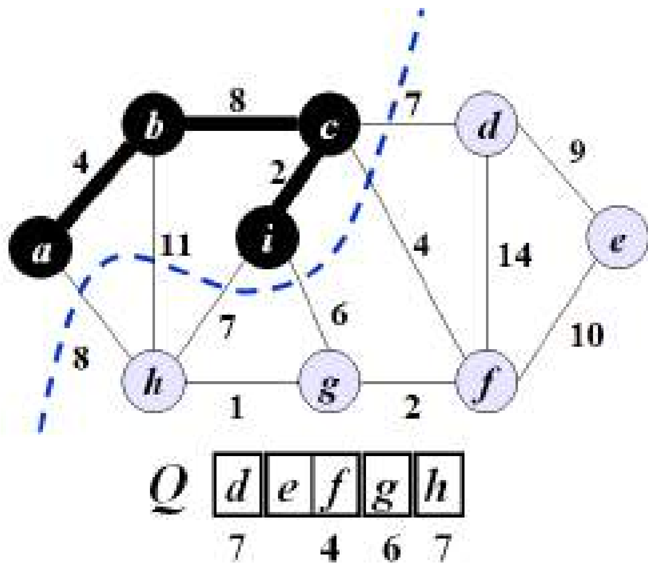
Total Running time = O (E + V lg V)

Prim's Algorithm

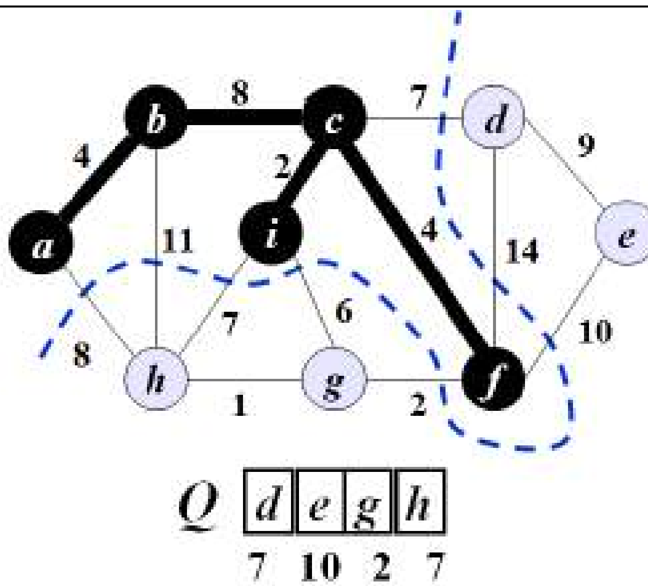
https://studyc.info/


b
c
d
a
i
e
h
g
f
4
8
7
9
2
11
4
14
7
6
10
8
1
2
Q
c
d
e
f
g
h
i
8
8

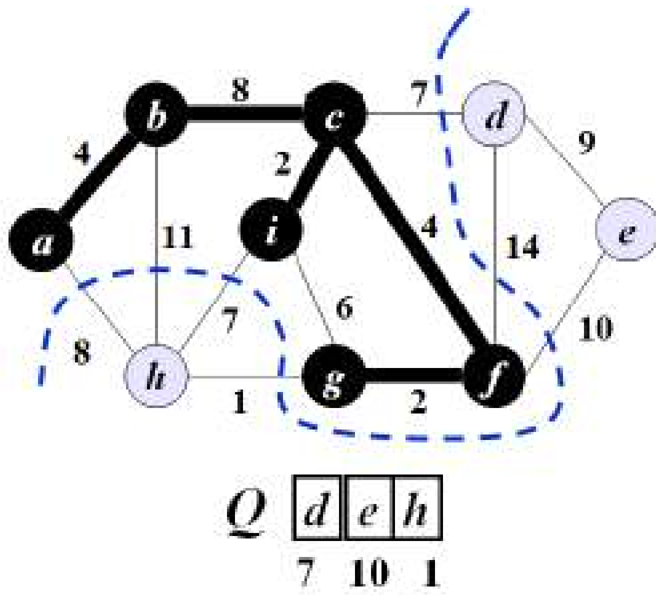

b
c
d
a
i
e
h
g
f
4
8
7
9
2
11
4
14
7
6
10
8
1
2
Q
d
e
f
g
h
i
7
4
8
2

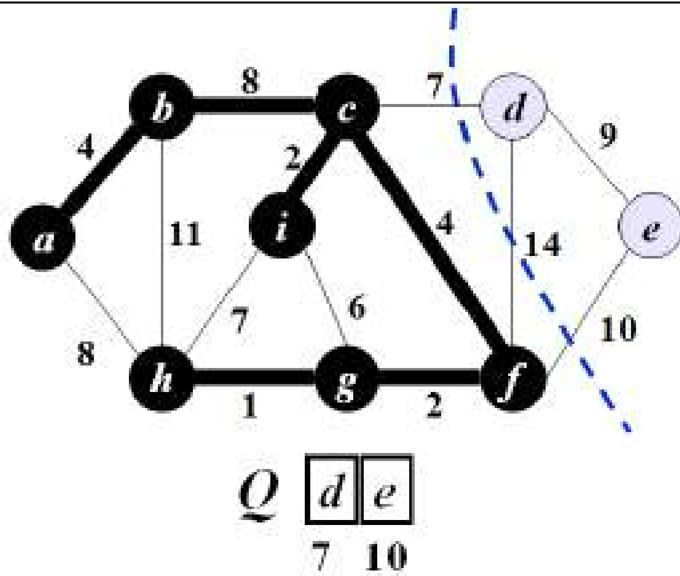


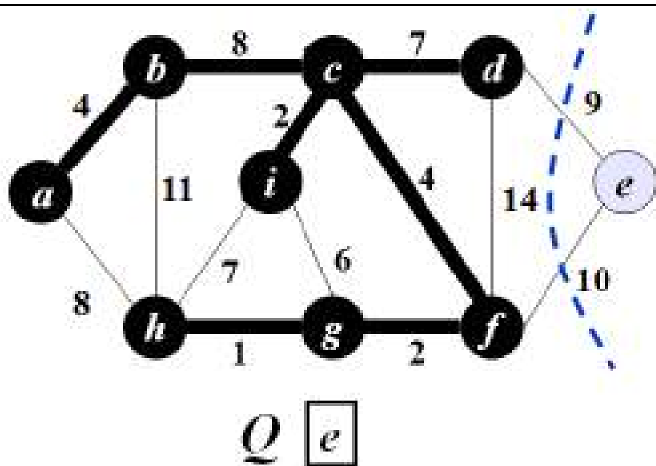
a
b
c
d
e
f
g
h
i
4
8
7
2
9
11
4
14
7
6
10
8
1
2
Q
d
e
f
g
h
7
4
6
7

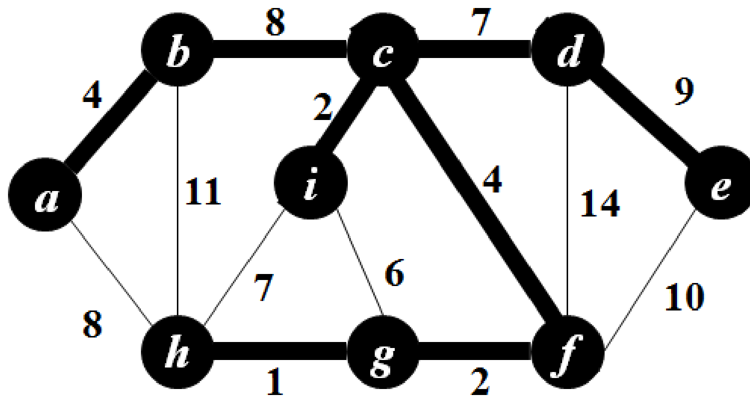


a
b
c
d
e
f
g
h
i
4
8
7
2
9
11
4
14
7
6
10
8
1
2
Q
d
e
g
h
7
10
2
7


a
b
c
d
e
f
g
h
i
8
7
4
2
9
11
4
14
7
6
10
8
1
2
Q d e h
7 10 1


a
b
c
d
e
f
g
h
i
8
7
4
2
9
11
4
14
7
6
10
8
1
2
Q d e
7 10


a
b
c
d
e
f
g
h
i
8
7
4
2
9
11
4
14
7
6
10
8
1
2
Q e



Importance of Minimal Spanning Trees

There are various applications of Minimal Spanning Trees (MST). Let us consider a couple of real-world examples

1. One practical application would be in designing a network.
- For example, a group of individuals, separated by varying distances, are to be connected in a telephone network.
- Although MST cannot do anything about distance from one connection to another, but it can reduce connecting cost.
2. Another useful application of it is finding airline routes.
- The vertices of the graph would represent cities, and the edges would represent routes between the cities.
- Obviously, more traveling require more cost
- Hence MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

https://studyc.info/

Lecture 33 Single-Source Shortest Path

Road Map Problem

We are given a road map on which the distance between each pair of adjacent cities is marked, and our goal is to determine the shortest route from one city to another. The number of possible routes can be huge. How do we choose which one routes is shortest? This problem can be modelled as a graph and then we can find the shortest path from one city to another using graph algorithms. How to solve this problem efficiently?

Linking Road Map with Graph Theory

This problem can be modeled as a graph problem

- Road map is a weighted graph where
  - set of vertices = set of cities
  - set of edges = road segments between cities
  - edge weight = length between two cities

Our goal is to find a shortest path between two vertices i.e. between two cities.

Weight of a Path

In a **shortest path problem**, a weighted, directed graph *G = (V, E)* is given with weight function *w: E → R* mapping edges to real-valued weights.

The **weight** of path $p = \langle v_0, v_1, ..., v_k \rangle$ is the sum of the weights of its constituent edges.

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$
$$= w(v_0, v_1) + w(v_1, v_2) + ... + w(v_{k-1}, v_k)$$

Shortest Path

A **shortest path** from vertex *u* to *v* is denoted by $\delta(u, v)$ *and is defined as*

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Weight of edges can represent any metric such as Distance, time, cost, penalty, loss etc.

Variants of Shortest Path

- **Single-source shortest path**
  - G = (V, E) ⇒ find a shortest path from a given source vertex s to each vertex v ∈ V
- **Single-destination shortest path**
  - Find a shortest path to a given destination vertex **t** from each vertex v

https://studyc.info/

- Reverse the direction of each edge ⇒ single-source

- **Single-pair shortest path**
  - Find a shortest path from u to v for given vertices u and v
  - Solve the single-source problem

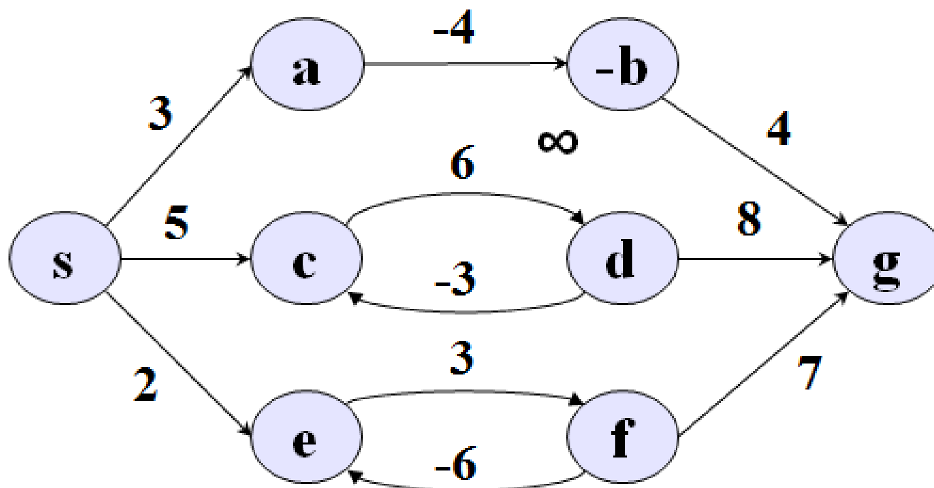
- **All-pairs shortest-paths**
  - Find shortest path for every pair of vertices u and v of G

Lemma: subpath of a shortest path, a shortest path

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbf{R}$, let $p = < v_1, v_2,..., v_k >$ be a shortest path from vertex v_1 to vertex v_k, for any i, j such that $1 \le i \le j \le k$, let $p_{ij} = < v_i, v_{i+1},..., v_j >$ be subpath of p from v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof: We prove this lemma by contradiction. If we decompose path p into $v_0 \overset{p0i}{\leadsto} v_i \overset{pij}{\leadsto} v_j \overset{pjk}{\leadsto} v_k$, then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. That is there is a subpath p'_{ij} from v_i to vertex v_j *which is shortest than* p_{ij}. Then, $v_0 \overset{p0i}{\leadsto} v_i \overset{p'ij}{\leadsto} v_j \overset{pjk}{\leadsto} v_k$, is a path from vertices v_1 to v_k whose weight $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$. It contradicts the assumption that p is a shortest path from v_1 to v_k. Hence subpath of a given shortest path is also a shortest path.

Why Positive Cycle Not?

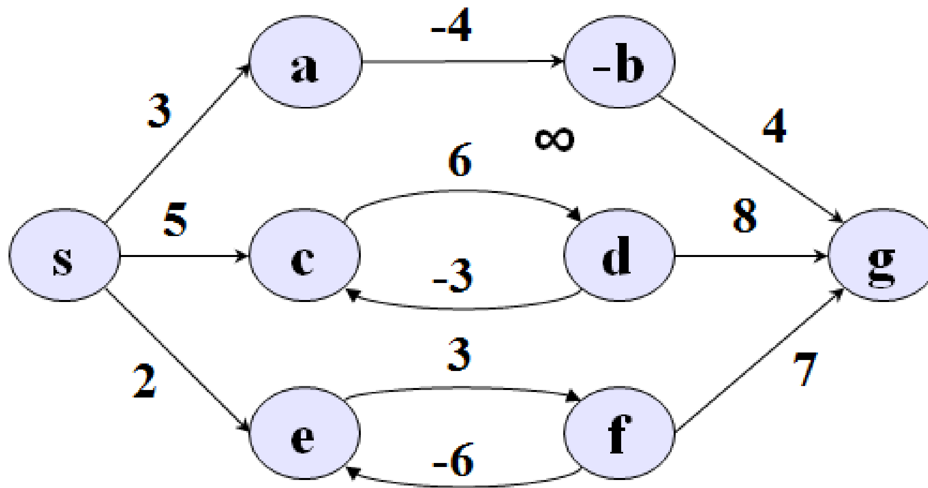


- s → a: only one path
  δ(s, a) = w(s, a) = 3
- s → b: only one path
  δ(s, b) = w(s, a) + w(a, b) = -1

https://studyc.info/

- s → c: infinitely many paths
  ⟨s, c⟩, ⟨s, c, d, c⟩, ⟨s, c, d, c, d, c⟩
- cycle has positive weight (6 - 3 = 3)
  ⟨s, c⟩ shortest path with weight δ(s, c) = w(s, c) = 5,
- Positive cycle increases length of paths

Why Negative Cycle Not?



s → e: infinitely many paths:

- ⟨s, e⟩, ⟨s, e, f, e⟩, ⟨s, e, f, e, f, e⟩ etc.
- cycle ⟨e, f, e⟩ has negative weight: 3 + (- 6) = -3
- paths from s to e with arbitrarily large negative weights
- δ(s, e) =- ∞ ⇒ no shortest path exists between s and e

Similarly:

δ(s, f) = - ∞, δ(s, g) = - ∞

Removing cycles from shortest paths

If $p = \langle v_0, v_1, ..., v_k \rangle$ is a path and $c = \langle v_i, v_{i+1}, ..., v_j \rangle$ is a positive weight cycle on this path then the path $p' = \langle v_0, v_1, ..., v_i, v_{j+1}, v_{j+2}, ..., v_k \rangle$ has weight *w(p') = w(p) – w(c) < w(p)*, and so *p* cannot be a shortest path from v_0 to v_k

As long as a shortest path has 0-weight cycles, we can repeatedly remove these cycles from path until a cycle-free shortest path is obtained.

When is no shortest path?

- There may be edges with negative weight.
- A cycle $p = v_0, v_1, ..., v_k, v_0$ is a **negative cycle** such that *w(p)* < 0
- If a graph *G = (V, E)* contains no negative weight cycle reachable from the source *s*, then for all *v* ∈ *V*, shortest path *δ(s, v)* remains well defined.
- If there is a negative weight cycles reachable from s, then shortest path weight are not well defined.

https://studyc.info/

- If there is a path from u to v that contains a negative cycle, then shortest path is defined as

  $\delta(u, v) = -\infty$

Summary of cycles in SPP

- Can shortest paths contain cycles?
- Negative-weight cycles: NO
- Positive-weight cycles: NO
  - By removing the cycle we can get a shorter path
- Zero-weight cycles
  - No reason to use them
  - Can remove them to obtain a path with similar weight

Note: We will assume that when we are finding shortest paths, the paths will have no cycles

Representing Shortest Paths

For a graph $G=(V, E)$, a **predecessor** $\pi[v]$ is maintained for each vertex $v \in V$

- Either vertex or NIL
- We are interested in **predecessor subgraph** $G_\pi=(V_\pi, E_\pi)$ induced by π values, such that

$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$

$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$

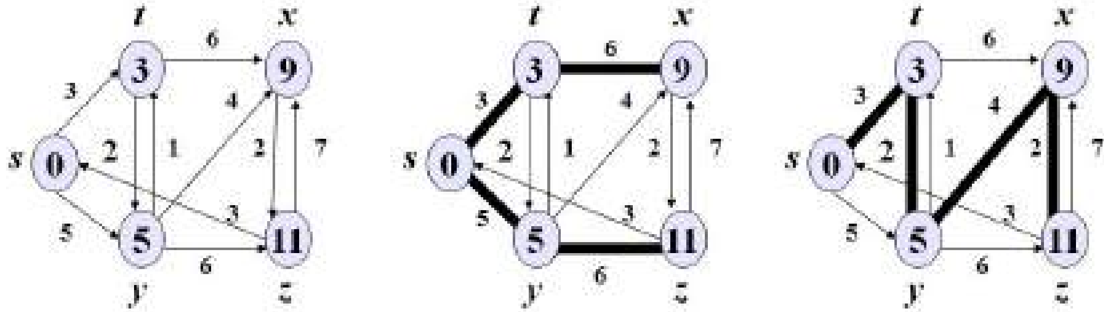
Shortest Path Rooted Tree

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow R$ and assume that G contains no negative weight cycles reachable from the source vertex $s \in V$, so that shortest paths are well defined.
A shortest path tree rooted at s is a directed subgraph $G'=(V', E')$, where $V' \subseteq V$ and $E' \subseteq E$
Shortest path are not necessarily unique and neither are shortest path trees.

Shortest path not unique

- Shortest path are neither necessarily
  - unique and
  - nor shortest path trees

https://studyc.info/



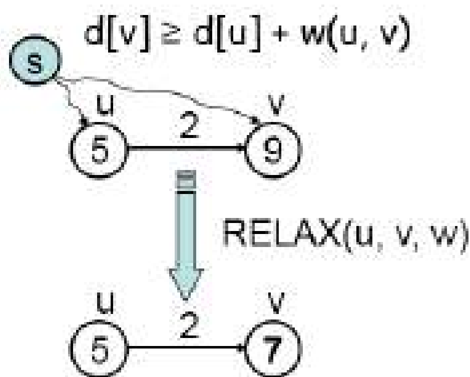
Initialization and Relaxation

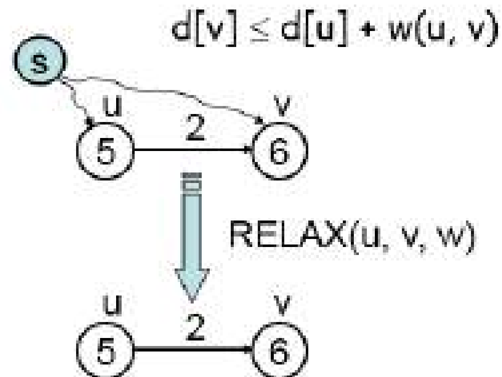
Initialization

- All the shortest-paths algorithms start with initialization of vertices.

Relaxation

- For each vertex $v \in V$, an attribute *d[v]* is defined and called a **shortest path estimate,** maintained
  - which is in fact, an upper bound on the weight of a shortest path from source *s* to *v*
- Process of **relaxing** an edge *(u, v)* consists of testing whether we can improve shortest path to *v* found so far, through *u,* if so update *d[v]* and *π[v]*.
- **Relaxing** edge (u, v), testing whether we can improve shortest path to v found so far through u
  - If d[v] > d[u] + w(u, v)
  - we can improve the shortest path to v
  - ⇒ update d[v] and π[v]





INITIALIZE-SINGLE-SOURCE (*G*, *s*)
1 **for** each vertex $v \in V[G]$
2 **do** $d[v] \leftarrow \infty$
3 $\pi[v] \leftarrow$ NIL
4 $d[s] \rightarrow 0$
RELAX (*u*, *v*, *w*)

https://studyc.info/

1 **if** $d[v] > d[u] + w(u, v)$
2 **then** $d[v] \leftarrow d[u] + w(u, v)$
3 $\pi[v] \leftarrow u$

Running Time: $\Theta(V)$

Note: All the single-source shortest-paths algorithms, start by calling INIT-SINGLE-SOURCE then relax edges. The algorithms differ in the order and how many times they relax each edge

The Bellman-Ford Algorithm

Input:

- Weighted, directed graph G, edges may be negative with weight function w : E → R,

Output

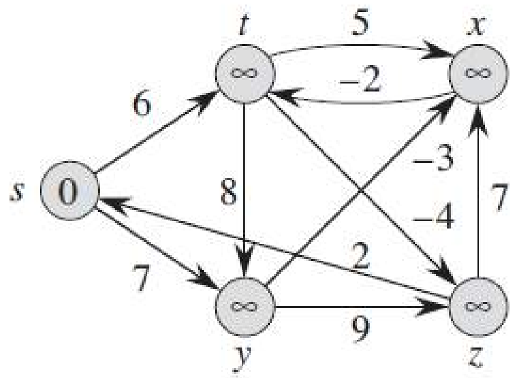
- It returns Boolean value indicating whether or not there is a negative-weight cycle reachable from source.
- If there is such a cycle, it indicates no solution exists
- Else it produces shortest paths and their weights.

Note: It uses relaxation progressively decreasing estimate d[v] on weight of a shortest path from source s to each vertex $v \in V$ until it achieves actual SP weight $\delta(s, v)$.

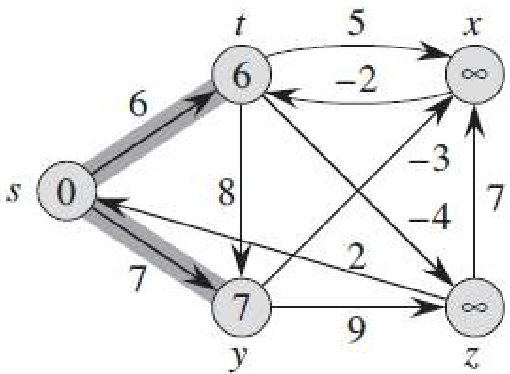
BELLMAN-FORD (G, w, s)
1 INITIALIZE-SINGLE-SOURCE (G, s)
2 **for** $i \leftarrow 1$ to $|V[G]| - 1$
3 **do for** each edge $(u, v) \in E[G]$
4 **do** RELAX (u, v, w)
5 **for** each edge $(u, v) \in E[G]$
6 **do if** $d[v] > d[u] + w(u, v)$
7 **then return** FALSE
8 **return** TRUE

Total Running Time = O(E)

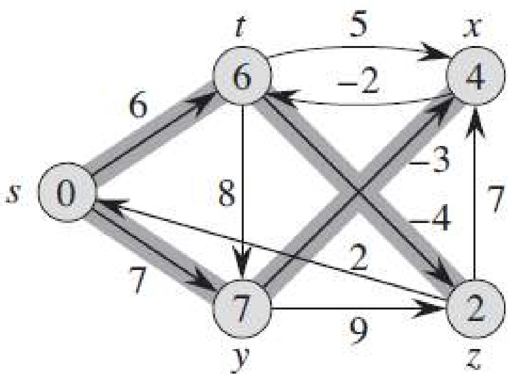
https://studyc.info/



(a)

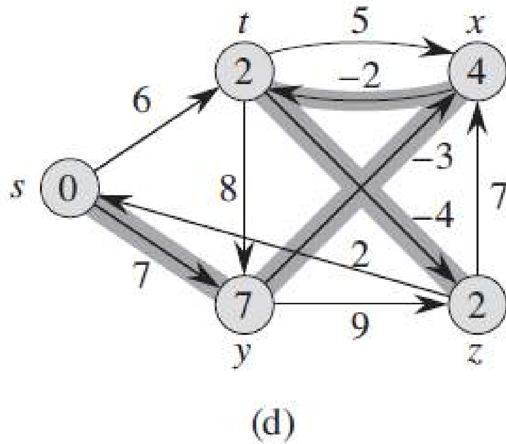


(b)

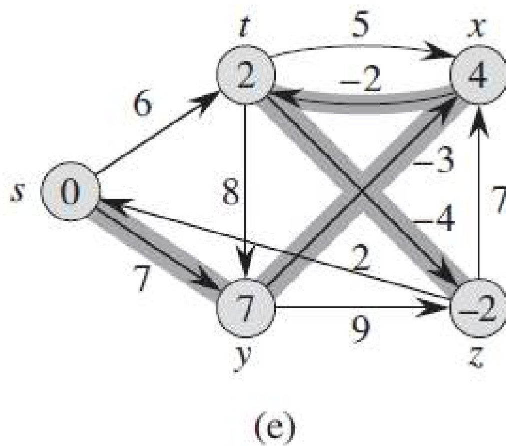


(c)

https://studyc.info/



(d)



(e)

Lemma 1

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w : E \rightarrow \mathbf{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the ***for*** loop of lines 2-4 of BELLMAN-FORD, we have $d[v] = \delta(s, v)$ for all vertices v that are reachable from s.

Proof: We prove the lemma by appealing to the path-relaxation property.

- Consider any vertex v that is reachable from s, and let $p = < v_0, v_1, ..., v_k >$, where $v_0 = s$ and $v_k = v$, be any acyclic shortest path from s to v.
- Path p has at most $|V|$ - 1 edges, and so $k \leq |V| - 1$.
- Each of the $|V|$ - 1 iterations of the **for** loop of lines 2-4 relaxes all E edges.
- Among the edges relaxed in the ith iteration, for $i = 1, 2, ..., k$, is (v_{i-1}, v_i).
- By the path-relaxation property, therefore, $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$.

Theorem: Correctness of Bellman-Ford algorithm

Let BELLMAN-FORD be run on weighted, directed graph G = (V, E), with source vertex s, and weight function w : E → R.

https://studyc.info/

- If G contains no negative-weight cycles that are reachable from s, then
  - d[v] = δ(s, v) for all vertices v ∈ V, and
  - the algorithm returns TRUE
  - the predecessor subgraph Gπ is shortest-paths tree rooted at s.
- If G does contain a negative weight cycle reachable from s, then the algorithm returns FALSE.

Proof

Case 1

Suppose graph G contains no negative-weight cycles that are reachable from the source s.

- We first prove the claim that at termination, $d[v] = \delta(s, v)$ for all vertices $v \in V$.
  - If v is reachable from s, **Lemma above** proves it.
  - If v is not reachable from s, then the claim follows from the no-path property. Thus, the claim is proven.
- The predecessor subgraph property, along with the claim, implies that G_π is a shortest-paths tree.
- Now we use the claim to show that BELLMAN-FORD returns TRUE.
  - At termination, for all edges (u, v)
  - $d[v] = \delta(s, v) \le \delta(s, u) + w(u, v) = d[u] + w(u, v)$,
  - It therefore returns TRUE

Case 2

- Suppose that graph G contains a negative-weight cycle that is reachable from the source s
- Let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$,

  Then, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$ ....(A)
- Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE.
- Thus, $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.
- Summing the inequalities around cycle c gives us

$$\sum_{i=1}^{k} d[v_i] \le \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$
$$= \sum_{i=1}^{k} (d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations and,

  and so $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$
- Of course $d[v_i]$ is finite for $i = 1, 2, \ldots, k$. Thus, $0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- Which contradicts inequality (A). And hence it proves the theorem

Different applications of shortest path

https://studyc.info/

- Transportation problems
  - finding the cheapest way to travel between two locations
- Motion planning
  - what is the most natural way for a cartoon character to move about a simulated environment
- Communications problems
  - How look will it take for a message to get between two places which two locations are furthest apart i.e.
  - what is the diameter of network

https://studyc.info/

Lecture 34 Proof: Bellman-Ford Algorithm & Shortest Paths in Directed Acyclic Graphs

Lemma 1

Let $G = (V, E)$ *be* directed, with source *s,* a weighted, with weight function $w : E \rightarrow \mathbf{R}$, and contains no negative-weight cycle reachable from *s.* Then, after the $|V| - 1$ iterations of the ***for*** loop of lines 2-4 of BELLMAN-FORD, we have $d[v] = \delta(s, v)$ for all vertices *v* that are reachable from *s.*

Path relaxation property

If $p = < v_0, v_1,..., v_k >$, be a shortest path from $s = v_0$ to v_k and edges of p are relaxed in the order (v_0, v_1), (v_1, v_2) . . . (v_{k-1}, v_k), then $d(v_k) = \delta(s, v_k)$

Proof

We prove it using path-relaxation property. Consider any vertex *v* that is reachable from *s* and let $p = < v_0, v_1,..., v_k >$, be any acyclic shortest path from *s* to *v,* where $v_0 = s$ and $v_k = v$. As there are k+1 vertices in the path p, hence there must be k edges in p. Because Graph has |V| vertices and path p contains no cycle, hence path *p* has at most $|V| - 1$ edges, and therefore, $k \leq |V| - 1$. Each of the |V| - 1 iterations of the for loop of lines 2-4 relaxes all E edges.

At i = 1, edge (v_0, v_1) is relaxed, and $d[v_1] = \delta(s, v_1)$. At i = 2, edge (v_1, v_2) is relaxed, and $d[v_2] = \delta(s, v_2)$.

By mathematical induction we can prove that at i = k, edge (v_{k-1}, v_k) is relaxed, $d[v_k] = \delta(s, v_k)$. Hence all the edges (v_{i-1}, v_i) will be relaxed after the iterations, i = 1, 2,..., *k*. By the path-relaxation property, after k^{th} iteration, $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$. Hence we have proved the required result using path relaxation property.

Theorem: Correctness of Bellman-Ford algorithm

Let BELLMAN-FORD be run on weighted, directed graph G = (V, E), with source vertex s, and weight function w: E → R.

- If G contains no negative-weight cycles that are reachable from s, then
  - $d[v] = \delta(s, v)$ for all vertices $v \in V$, and
  - the algorithm returns TRUE
  - the predecessor subgraph $G\pi$ is shortest-paths tree rooted at s.
- If G does contain a negative weight cycle reachable from s, then the algorithm returns FALSE.

Proof:

Case 1

Suppose graph G contains no negative-weight cycles that are reachable from the source s.

https://studyc.info/

- We first prove the claim that at termination, $d[v] = \delta(s, v)$ for all vertices $v \in V$.
  - If v is reachable from s, **Lemma above** proves it.
  - If v is not reachable from s, then claim follows from no-path property.
- The predecessor subgraph property, along with the claim, implies that G_π is a shortest-paths tree.
  (Once d[v] = δ(s, v) for all v ∈ V, the predecessor sub-graph is a shortest paths tree rooted at s)

- Now we use the claim to show that BELLMAN-FORD returns TRUE.
  - At termination, for all edges (u, v)
  - $d[v] = \delta(s, v) \le \delta(s, u) + w(u, v) = d[u] + w(u, v)$,
  - It therefore returns TRUE

Case 2

- Suppose that graph G contains a negative-weight cycle that is reachable from the source s
- Let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$,

  Then, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$ (A)
- Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE.
- Thus, $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.
- Summing the inequalities around cycle c gives us

$$\sum_{i=1}^{k} d[v_i] \le \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} (d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations and,

  and so $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$
- Of course $d[v_i]$ is finite for $i = 1, 2, \ldots, k$. Thus, $0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- Which contradicts inequality (A). And hence it proves the theorem

<u>Shortest Paths in Directed Acyclic Graphs</u>

- By relaxing edges of Directed Acyclic Graph (dag) $G = (V, E)$ according to topological sort of vertices single source shortest path can be computed in $\Theta(V + E)$ time
- Shortest paths are always well defined in a dag
  - Since even if there are negative-weight edges no negative weight cycle exists.
- It starts topologically sorting dag, to impose linear ordering of vertices.
  - If there is path from u to v then u precedes v.
- Each vertex and each edge that leaves the vertex is processed that is why this approach is well defined

https://studyc.info/

Algorithm: Topological Sort

TOPOLOGICAL-SORT (*G*)

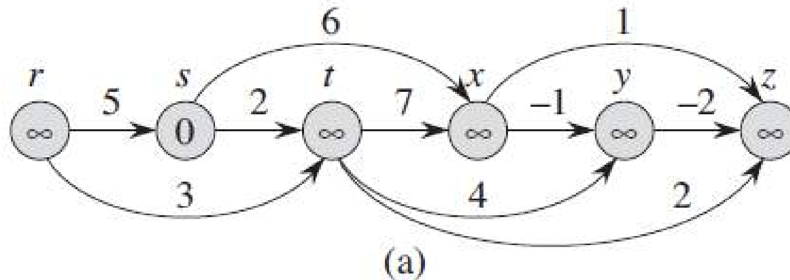
1. Call DFS(*G*) to compute *f* [*v*] of each vertex $v \in V$.
2. Set an empty linked list *L* = Ø.
3. When a vertex *v* is colored black, assign it *f* (*v*).
4. Insert *v* onto the front of the linked list, *L* = {*v*}.*L*.
5. **return** the linked list.
6. The rank of each node is its position in the linked list started from the head of the list.

Total Running Time = Θ (V + E)

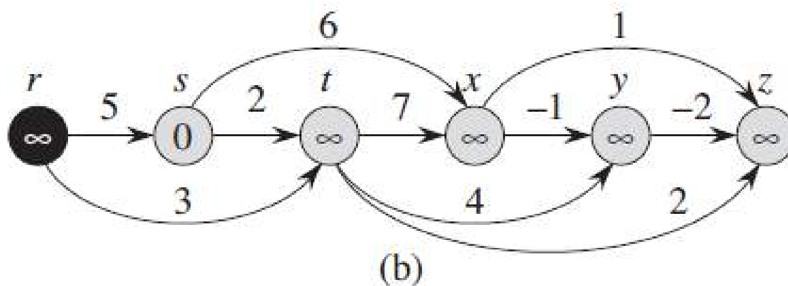
Algorithm: Shortest Path (dag)

DAG-SHORTEST-PATHS (*G*, *w*, *s*)
1 topologically sort the vertices of *G*
2 INITIALIZE-SINGLE-SOURCE (*G*, *s*)
3 **for** each vertex *u*, taken in topologically sorted order
4 **do for** each vertex $v \in Adj[u]$
5 **do** RELAX (*u*, *v*, *w*)

SSSP in Directed Acyclic Graphs

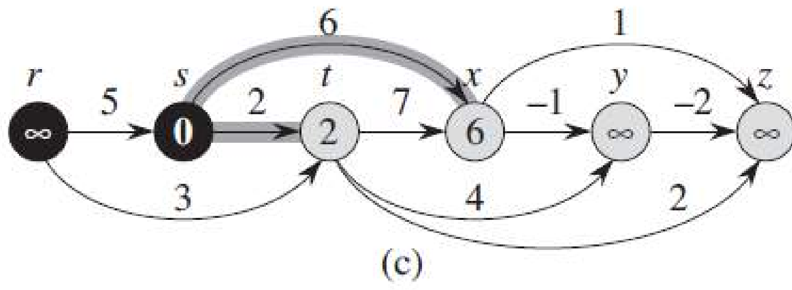


(a)

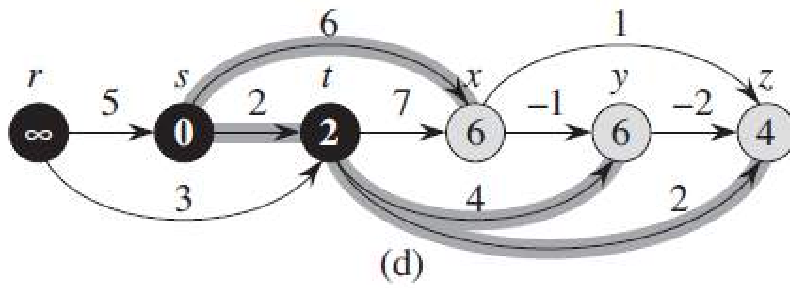


(b)

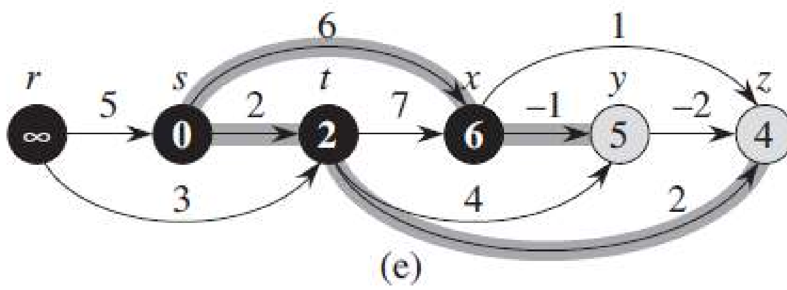
https://studyc.info/



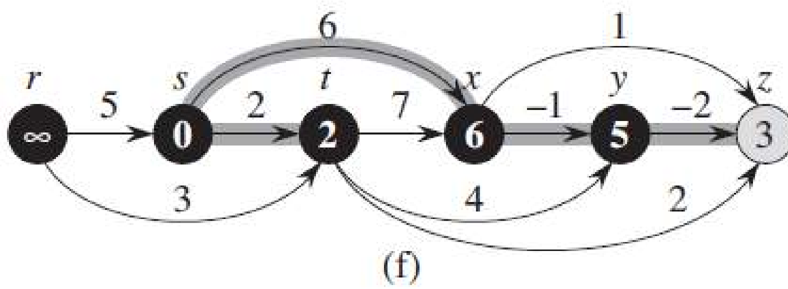
(c)



(d)

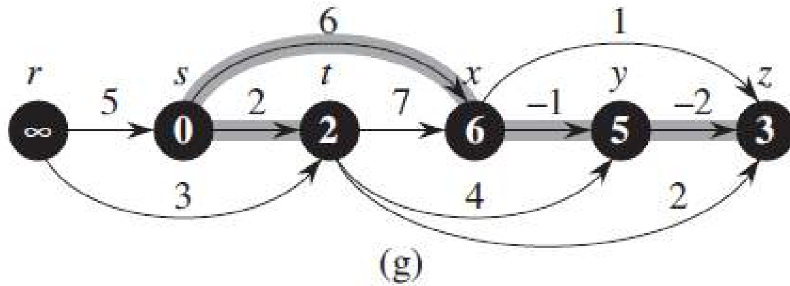


(e)



(f)

https://studyc.info/



Theorem: Proof of Correctness

If a weighted, directed graph $G = (V, E)$ has source vertex s and no cycles, then at the termination of the DAG-SHORTEST-PATHS procedure, $d[v] = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree.

Proof

We first show that $d[v] = \delta(s, v)$ for all vertices $v \in V$ at termination.

Case 1

If v is not reachable from s, then $d[v] = \delta(s, v) = \infty$ by the no-path property.

Case 2

- Now, suppose that v is reachable from s, so that there is a shortest path $p = <v_0, v_1,..., v_k>$, where $v_0 = s$ and $v_k = v$.
- Because we process the vertices in topologically sorted order, the edges on p are relaxed in the order $(v_0, v_1), (v_1, v_2),..., (v_{k-1}, v_k)$.
- The path-relaxation property implies that $d[v_i] = \delta(s, v_i)$ at termination for $i = 0, 1,..., k$.
- Hence it proves the theorem

https://studyc.info/

Lecture 35 Dijkstra's Algorithm

Problem Statement:

- Given a graph G = (V, E) with a source vertex s, weight function w, edges are non-negative, i.e., $w(u, v) \geq 0, \forall (u, v) \in E$
- The graph is directed, i.e., if $(u, v) \in E$ then (v, u) may or may not be in E.
- The objective is to find shortest path from s to every vertex $u \in V$.

Approach

- A "cloud S" of vertices, beginning with s, will be constructed, finally covering all vertices of graph
- For each vertex v, a label d(v) is stored, representing distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, d(u)
  - We update labels of the vertices adjacent to u

Mathematical Statement of Problem

Input Given graph G(V, E) with source s, weights w

Assumption

- Edges non-negative, $w(u, v) \geq 0, \forall (u, v) \in E$
- Directed, if $(u, v) \in E$ then (v, u) may be in E

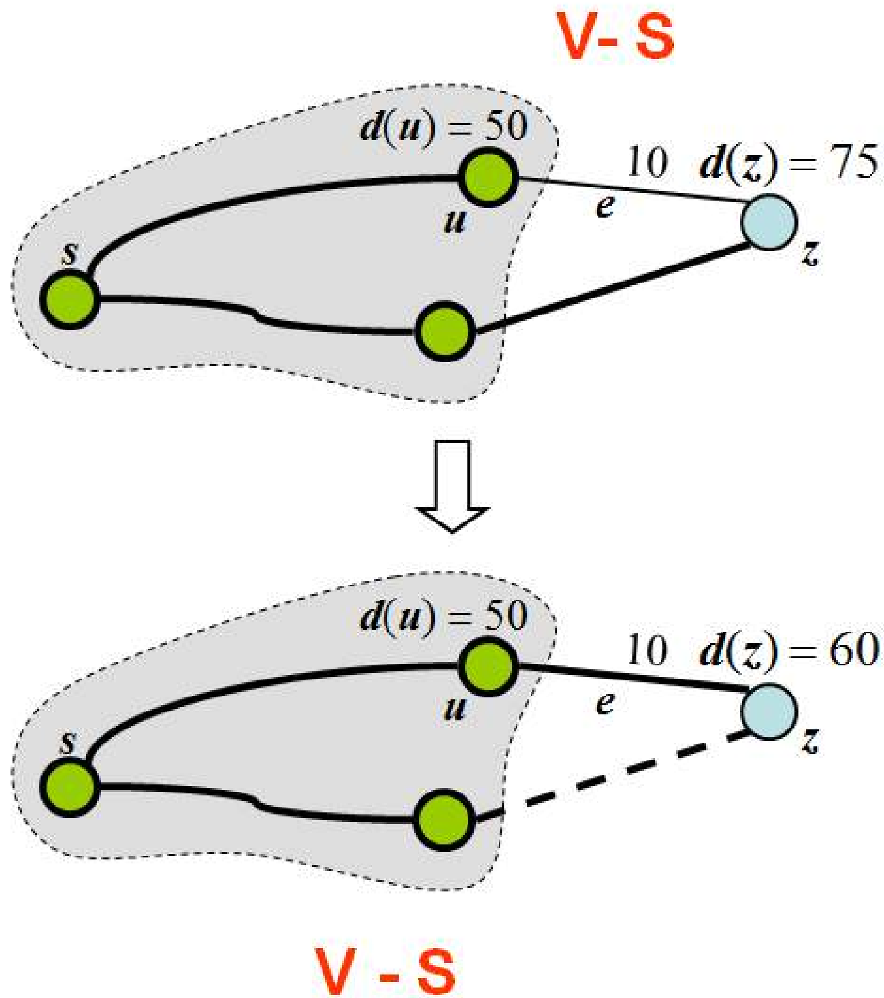
Objective: Find shortest paths from s to every $u \in V$

Approach

- Maintain a set S of vertices whose final shortest-path weights from s have been determined
- Repeatedly select, $u \in V - S$ with minimum shortest path estimate, add u to S, relax all edges leaving u.
- Greedy, always choose light vertex in V-S , add to S

Edge Relaxation

https://studyc.info/



- Consider edge e = (u, z) such that
  - u is vertex most recently added to the cloud S
  - z is not in the cloud
- Relaxation of edge e updates distance d(z) as
  d(z) = min {d(z), d(u) + weight(e)}

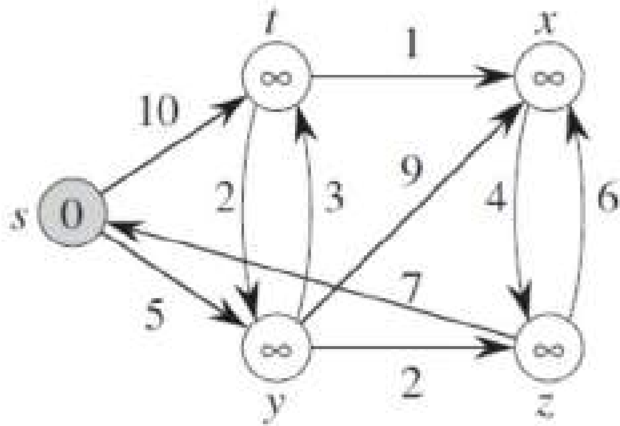
Dijkstra's Algorithm

DIJKSTRA(G, w, s)

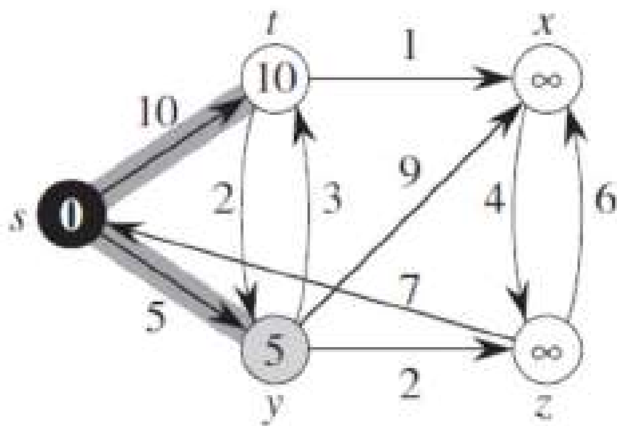
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 S ← Ø
3 Q ← V[G]
4 **while** Q ≠ Ø
5 **do** u ← EXTRACT-MIN(Q)
6 S ← S ∪ {u}
7 **for** each vertex v ∈ Adj[u]
8 **do** RELAX (u, v, w)

https://studyc.info/

Example: Dijkstra's Algorithm

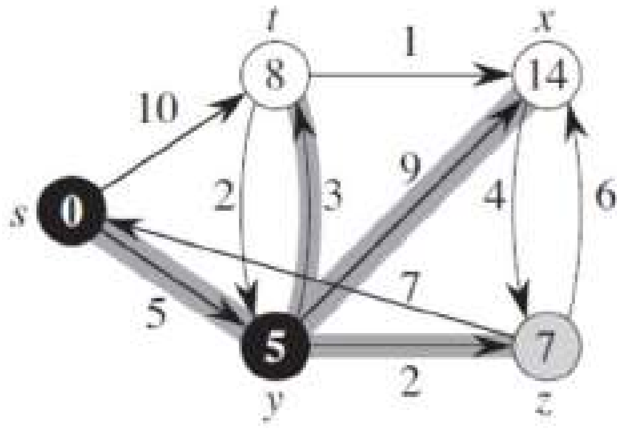


(a)

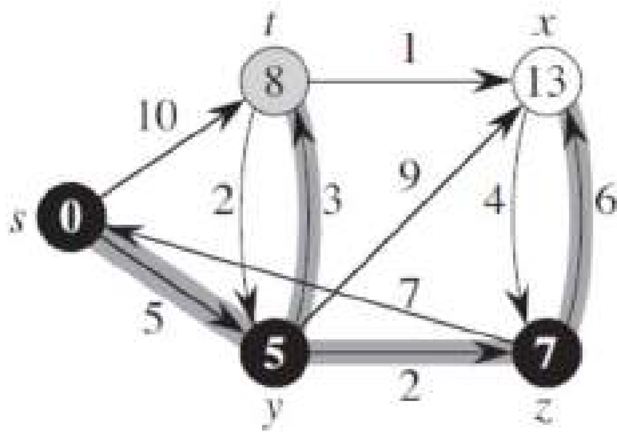


(b)

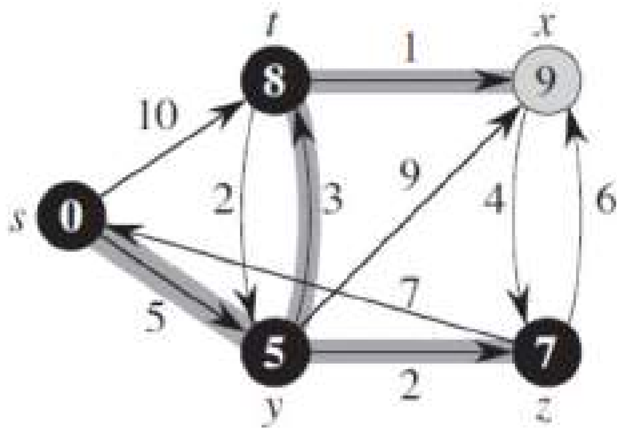
https://studyc.info/



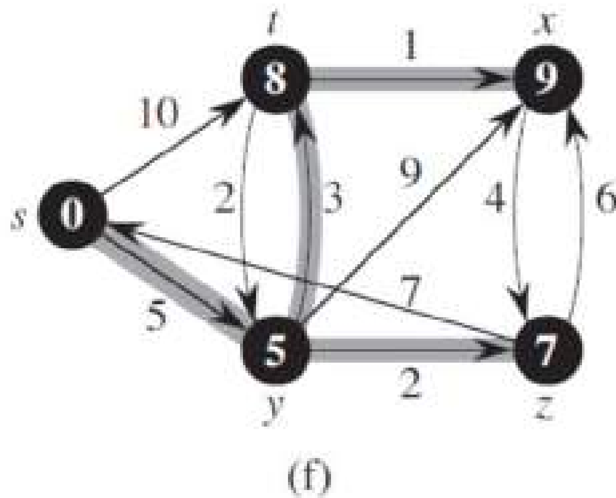
(c)



(d)



(e)



(f)

Analysis: Dijkstra's Algorithm

Cost depends on implementation of min-priority queue

Case 1:

Vertices being numbered 1 to $|V|$

- INSERT, DECREASE-KEY operations takes *O(1)*
- EXTRACT-MIN operation takes *O(V)* time
- ***Sub cost is $O(V^2)$***
- **Total number of edges in all adjacency list is |E|**
- **Total Running time = $O(V^2 + E) = O(V^2)$**

Case 2:

Graph is sufficiently spare, e.g., $E = O(V^2 / lgV)$

Implement min-priority queue with binary min heap

Vertices being numbered 1 to $|V|$

- Each EXTRACT-MIN operation takes *O(lgV)*
- There |V| operations, time to build min heap O(V)
- ***Sub cost is O(V lgV)***
- *Each DECREASE-KEY operation takes time O(lgV), and there are |E| such operation.*
- ***Sub cost is O(E lgV)***

Hence Total Running time = $O(V + E)$ lgV = E lgV

Case 3:

Implement min-priority queue with Fibonacci heap

Vertices being numbered 1 to $|V|$

- Each EXTRACT-MIN operation takes *O(lgV)*
- There |V| operations, time to build min heap O(V)
- ***Sub cost is O(V lgV)***

https://studyc.info/

- *Each DECREASE-KEY operation takes time O(1), and there are |E| such operation.*
- ***Sub cost is O(E)***
  Hence Total Running time = *O (V.lgV + E) = V* lgV

Case 1: Computation Time

```
1. INITIALIZE-SINGLE-SOURCE(V, s)          ← Θ(V)
2.  S ← ∅
3.  Q ← V[G]          ← O(V) build min-heap
4.  while Q ≠ ∅       ← O(V)
5.     do u ← EXTRACT-MIN(Q)          ← O(V)
6.         S ← S ∪ {u}
7.         for each vertex v ∈ Adj[u]     ← O(E)
8.             do RELAX(u, v, w)
```

Running time: $O(V^2 + E) = O(V^2)$

Note: Running time depends on Implementation Of min-priority (Q)

Case 2: Binary min Heap

```
1. INITIALIZE-SINGLE-SOURCE(V, s)          ← Θ(V)
2.  S ← ∅
3.  Q ← V[G]                  ← O(V) build min-heap
4.  while Q ≠ ∅               ← Executed O(V) times
5.     do u ← EXTRACT-MIN(Q)     ← O(lgV)
6.         S ← S ∪ {u}
7.         for each vertex v ∈ Adj[u]
8.             do RELAX(u, v, w)           ← O(E) times O(lgV)
```

Running time: O(VlgV + ElgV) = O(ElgV)

Case 3: Fibonacci Heap

```
1. INITIALIZE-SINGLE-SOURCE(V, s)          ← Θ(V)
2.  S ← ∅
3.  Q ← V[G]          ← O(V) build min-heap
4.  while Q ≠ ∅       ← Executed O(V) times
5.     do u ← EXTRACT-MIN(Q)          ← O(lgV)
6.         S ← S ∪ {u}
7.         for each vertex v ∈ Adj[u]
8.             do RELAX(u, v, w)           ← O(E) times O(1)
```

Running time: O(VlgV + E) = O(VlgV)

Theorem: Correctness of Dijkstra's Algorithm

Dijkstra's algorithm, runs on a weighted, directed graph $G = (V, E)$ with non-negative weight function w and source s, terminates with $d[u] = \delta(s, u)$ for all vertices $u \in V$.

Proof

- We use the following loop invariant:

https://studyc.info/

- At start of each iteration of the **while** loop of lines 4-8, $d[v] = \delta(s, v)$ for each vertex $v \in S$.

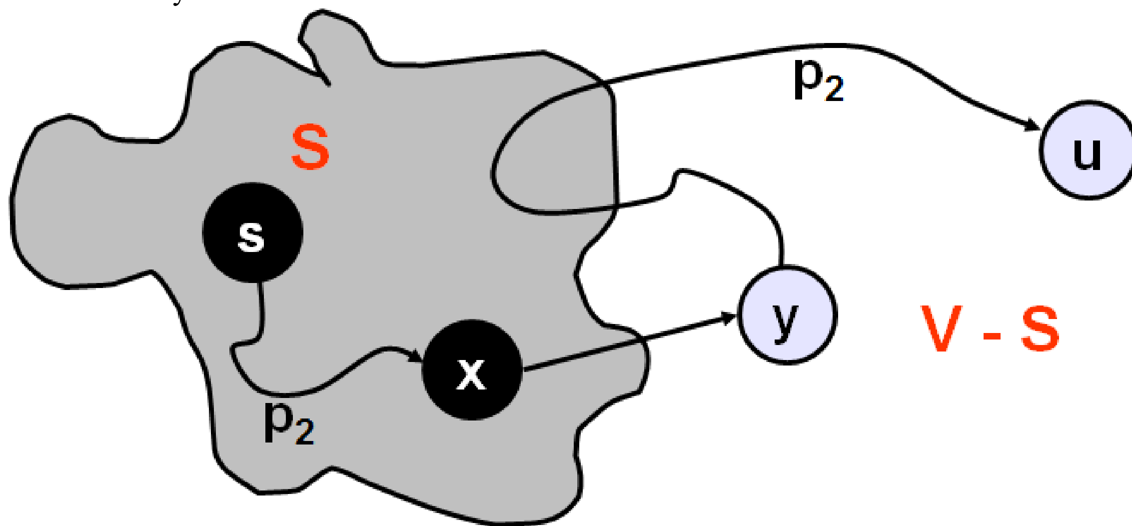
- It suffices to show for each vertex $u \in V$, we have $d[u] = \delta(s, u)$ at the time when u is added to set S.
- Once we show that $d[u] = \delta(s, u)$, we rely on the upper-bound property to show that the equality holds at all times thereafter.

Initialization**:**

- Initially, $S = \emptyset$, and so the invariant is trivially true

Maintenance**:**

- We wish to show that in each iteration, $d[u] = \delta(s, u)$, for the vertex added to set S.
- On contrary suppose that $d[u] \neq \delta(s, u)$ when u is added to set S. Also suppose that u *is* the first vertex for which the equality does not hold.
- We focus on situation at beginning of **while** loop in which u is added to S and derive a contradiction.
- First of all, $u \neq s$ because s is the first vertex added to set S and $d[s] = \delta(s, s) = 0$ at that time.
- Secondly $S \neq \emptyset$ just before u is added to S, this is because s is at least in S.
- There must be some path from s to u, otherwise $d[u] = \delta(s, u) = \infty$ by no-path property, which would violate our assumption that $d[u] \neq \delta(s, u)$.
- Because there is at least one path, there must be a shortest path p from s to u.
- Prior to adding u to S, path p connects a vertex in S, namely s, to a vertex in $V - S$, namely u.



- Let us consider the first vertex y along p such that $y \in V - S$, and let $x \in S$ be y's predecessor.
- Thus, path p can be decomposed: $s \overset{p1}{\leadsto} x \rightarrow y \overset{p2}{\leadsto} u$ (either of paths $p1$ or $p2$ may have no edges.)
- We claim that $d[y] = \delta(s, y)$ when u is added to S.

Proof of Claim: observe that $x \in S$.

https://studyc.info/

- Because u is chosen as the first vertex for which $d[u] \neq \delta(s, u)$ when it is added to S, we had $d[x] = \delta(s, x)$ when x was added to S.
- Edge (x, y) was relaxed at that time, and hence $d[y] = \delta(s, y)$ (convergence property).
- Because y occurs before u on a shortest path from s to u and all edge weights are nonnegative (on path p2), we have $\delta(s, y) \leq \delta(s, u)$,
- Now $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u] \Rightarrow d[y] \leq d[u]$ (1)
- But because both vertices u and y were in V - S when u was chosen, we have $d[u] \leq d[y]$. (2)
- From (1) and (2), $d[u] = d[y]$
- Now, $d[y] = \delta(s, y) \leq \delta(s, u) = d[u] = d[y] \quad \Rightarrow \delta(s, y) = \delta(s, u)$.
- Finally, $d[u] = \delta(s, u)$, it contradicts choice of u
- Hence, $d[u] = \delta(s, u)$ when u is added to S, and this equality is maintained at all times after that

<u>Termination:</u>

- At termination, $Q = \emptyset$ which, along with our earlier invariant that $Q = V - S$, implies that $S = V$.
- Thus, $d[u] = \delta(s, u)$ for all vertices $u \in V$.

<u>Lemma 1</u>

Statement

Let G = (V, E) be a weighted, directed graph with weight function $w: E \rightarrow R$, let $s \in V$ be a source vertex. Assume that G contains no negative-weight cycles reachable from s. Then, after the graph is initialized by INITIALIZE-SINGLE-SOURCE(G, s), the predecessor sub-graph G_π forms a rooted tree with root s, and any sequence of relaxation steps on edges of G maintains this property as an invariant.

Proof

- Initially, the only vertex in G_π is the source vertex, and the lemma is trivially true.
- Let $G\pi$ be a predecessor subgraph that arises after a sequence of relaxation steps.

(a) First we prove that $G\pi$ is a rooted tree.

1. G_π is acyclic

- On contrary suppose that some relaxation step creates a cycle in the graph $G\pi$.
- Let $c = \langle v_0, v_1, ..., v_k \rangle$ be cycle, where $v_k = v_0$.
- Then, $\pi[v_i] = v_{i-1}$ for i = 1, 2,..., k
- Now, without loss of generality, we can assume that it was the relaxation of edge (v_{k-1}, v_k) that created the cycle in G_π.

<u>Claim:</u> all vertices on cycle c reachable from s.

- Because each vertex has non-NIL predecessor, and it was assigned a finite shortest-path estimate when it was assigned non-NIL π value
- By upper-bound property, each vertex on c has a finite shortest-path weight, and reachable from s.

Shortest-path on c just prior RELAX(v_{k-1}, v_k, w)

https://studyc.info/

- Just before call, $\pi[v_i] = v_{i-1}$ for i = 1, 2,..., k - 1.
- Thus, for i = 1, 2,..., k - 1, last update to $d[v_i]$ was $d[v_i] \leftarrow d[v_{i-1}] + w(v_{i-1}, v_i)$.
- It is obvious that, $d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k)$.
- Summing it with k - 1 inequalities,

$$\sum_{i=1}^{k} d[v_i] > \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{1=1}^{k} w(v_{i-1}, v_i)$$

But, $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$

Hence, $0 > \sum_{1=1}^{k} w(v_{i-1}, v_i)$

- Thus, sum of weights around cycle c is negative, which provides the contradiction.
- We have proved that Gπ is a directed, acyclic.

2. To show that it forms a rooted tree with root s

- Sufficient to prove that ∀ v ∈ Vπ, there is a unique path from s to v in Gπ.
- On contrary, suppose there are two simple paths from s to some vertex v, and (x ≠ y)

  p_1: $s \overset{p1}{\leadsto} u \overset{p2}{\leadsto} x \rightarrow z \in v$

  p_2: $s \overset{p1}{\leadsto} u \overset{p1}{\leadsto} y \rightarrow z \in v$

- π[z] = x and π[z] = y, ⇒ x = y, a contradiction.
- Hence there exists unique path in Gπ from s to v.

  Thus Gπ forms a rooted tree with root s.

b. Now by predecessor subgraph property

- d[v] = δ(s, v) for all vertices v ∈ V. Proved

Lemma 2

If we run Dijkstra's algorithm on weighted, directed graph G = (V, E) with nonnegative weight function w and source s, then at termination, predecessor subgraph Gπ is a shortest paths tree rooted at s.

Proof:

- Immediate from the above lemma.

https://studyc.info/

Lecture 36 All Pairs Shortest Paths

All-Pairs Shortest Path (APSP): Approach

- In all-pair shortest path problems, graph G given as
  - Directed, weighted with weight function $w : E \rightarrow R$
  - where w is a function from edge set to real-valued weights
- Our objective is to find shortest paths, for all pair of vertices $u, v \in V$,

Approach

- All-pair shortest path problem can be solved by running single source shortest path in |V| times, by taking each vertex as a source vertex.

- Now there are two cases.

Edges are non-negative

Case 1

- Then use Dijkstra's algorithm
- Linear array of min-priority queue takes $O(V^3)$
- Binary min-heap of min-priority queue, $O(VE \lg V)$
- Fibonacci heap of min-priority queue takes $O(V^2 \lg V + VE)$

Negative weight edges are allowed

Case 2

- Bellman-Ford algorithm can be used when negative weight edges are present
- In this case, the running time is $O(V^2E)$
- However if the graph is dense then the running time is $O(V^4)$

Note

- Unlike single-source shortest path algorithms, most algorithms of all pair shortest problems use an adjacency-matrix representation
- Let us define adjacency matrix representation.

Adjacency Matrix Representation

Assumptions

- Vertices are numbered from 1, 2,. . ., |V|
- In this way input is an n × n matrix
- W represents edges weights of n-vertex directed weighted graph G *i.e.*, $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases}$$

Shortest Path and Solution Representation

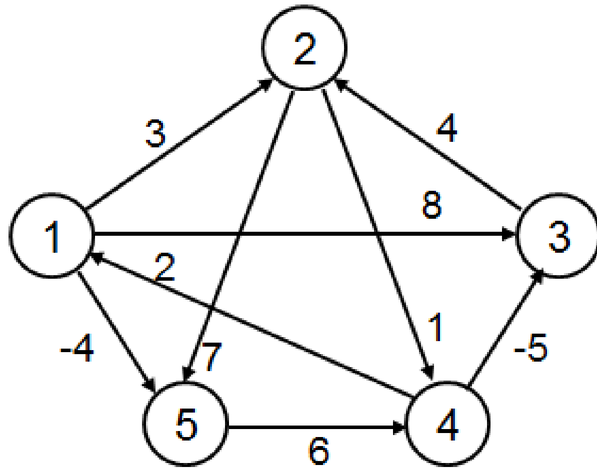
https://studyc.info/

- For a moment we assume that negative-weight edges are allowed, but input graph contains no negative-weight cycle

- The tabular output of all-pairs shortest-paths algorithms will be presented an $n \times n$ matrix $D = (d_{ij})$, where entry d_{ij} contains the weight of a shortest path from vertex i to vertex j. And the
- A **Predecessor Matrix** $\Pi = (\pi_{ij})$, where
  π_{ij} = NIL, if either $i = j$ or no path from i to j

π_{ij} = predecessor of j on some shortest path from i, otherwise

Example: All-Pairs Shortest Path



- **Given**
  - Directed graph G = (V, E)
  - Weight function w : E → **R**

- **Compute**
  - The shortest paths between all pairs of vertices in a graph
  - Representation of result:
    - 5 × 5 matrix of shortest-path distances δ(u, v)
    - 5 × 5 matrix of predecessor sub-graph

Structure of Output: Sub-graph for each row

- For each vertex $i \in V$ the **Predecessor Subgraph** of G for i is defined as

$G_{\pi, i} = (V_{\pi, i}, E_{\pi.i})$, where

$V_{\pi.i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$ and $E_{\pi.i} = \{(i, j) : j \in V_{\pi.i} - \{i\}\}$

- $G_{\pi, i}$ is shortest pat tree as was in single source shortest path problem

Printing Output

PRINT-ALL-PAIRS-SHORTEST-PATH (Π, i, j)

1 **if** $i = j$

https://studyc.info/

2 **then** print i
3 **else if** π_{ij} = NIL
4 **then** print "no path from" i "to" j "exists"
5 **else** PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_{ij})
6 print j

Shortest Paths and Matrix Multiplication

- Here we present a dynamic-programming algorithm for all-pairs shortest paths on a directed graph G = (V, E).
- Each major loop of dynamic program will invoke an operation very similar to multiplication of two matrices, and algorithm looks like repeated matrix multiplication
- At first we will develop $\Theta(V^4)$-time algorithm and then improve its running time to $\Theta(V^3 \lg V)$.
- Before we go for dynamic solution, let us have a review of steps involved in dynamic-programming algorithms.

Steps in Dynamic Programming

Steps on dynamic-programming algorithm are

1. Characterize the structure of an optimal solution.
2. Recursively define value of an optimal solution
3. Computing value of an optimal solution in bottom-up
4. Constructing optimal solution from computed information

Note: Steps 1-3 are for optimal value while step 4 is for computing optimal solution

1. Structure of an Optimal Solution

- Consider shortest path p from vertex i to j, and suppose that p contains at most m edges
  - If $i = j$, then p has weight 0 and no edges
  - If i and j are distinct, then decompose path p into $i \overset{p'}{\leadsto} k \rightarrow j$, where path p' contains at most m - 1 edges and it is a shortest path from vertex i to vertex k, and

Hence $\delta(i, j) = \delta(i, k) + w_{kj}$.

2. A Recursive Solution

Let $l_{ij}^{(m)}$ = minimum weight of path i to j at most m edges

- $m = 0$, there is shortest path i to j with no edges $\Leftrightarrow$ $i = j$, *thus*

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

- $m \geq 1$, compute using $l_{ij}^{(m-1)}$ and adjacency matrix w

https://studyc.info/

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right)$$
$$= \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}$$

The actual shortest-path weights are therefore given by
$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots$

3. Compute Shortest Path Weights Bottom-up

- Input matrix $W = (w_{ij})$,
- Suppose that, $L^{(m)} = (l_{ij}^{(m)})$, where, $m = 1, 2, \ldots, n-1$
- Compute series of matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$,
- Objective function, $L^{(n-1)}$, at most n-1 edges in each path

Note

- Observe that $l_{ij}^{(1)}$, $= w_{ij}$, for all $i, j \in V$, and so $L^{(1)} = W$
- Heart of the algorithm is : given matrices $L^{(m-1)}$ and W, and compute the matrix $L^{(m)}$
- That is, it extends shortest paths computed so far by one more edge.

Algorithm: Extension from $L^{(m-1)}$ to $L^{(m)}$

EXTEND-SHORTEST-PATHS (*L*, *W*)
1 $n \leftarrow rows[L]$
2 let $L' = (l'_{ij})$ be an $n \times n$ matrix
3 **for** $i \leftarrow 1$ **to** n
4 **do for** $j \leftarrow 1$ **to** n
5 **do** $l'_{ij} \leftarrow \infty$
6 **for** $k \leftarrow 1$ **to** n
7 **do**
8 **return** L' $l'_{ij} \leftarrow \min\left(l'_{ij}, l_{ik} + w_{kj}\right)$

Total Running Time = Θ (n^3)

Algorithm is Similar to Matrix Multiplication

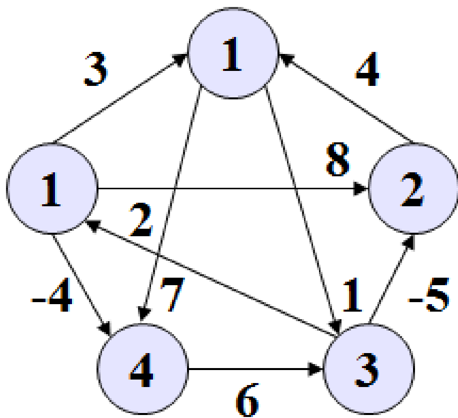
MATRIX-MULTIPLY (A, B)
1 $n \leftarrow rows[A]$
2 let C be an $n \times n$ matrix
3 **for** $i \leftarrow 1$ **to** n
4 **do for** $j \leftarrow 1$ **to** n
5 **do** $c_{ij} \leftarrow 0$
6 **for** $k \leftarrow 1$ **to** n
7 **do** $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
8 **return** C
Running Time = Θ (n^3)

https://studyc.info/

Complete but Slow Algorithm

SLOW-ALL-PAIRS-SHORTEST-PATHS (W)
1 $n \leftarrow rows\,[W]$
2 $L^{(1)} \leftarrow W$
3 **for** $m \leftarrow 2$ **to** n - 1
4 **do** $L^{(m)} \leftarrow$ EXTEND-SHORTEST-PATHS ($L^{(m-1)}$, W)
5 **return** $L^{(n-1)}$
Total Running Time = $\Theta(n^4)$

Example:



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

The reader may verify that $L^{(4)} = L^{(5)} = L^{(6)} = \cdots$

https://studyc.info/

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Improving Running Time

- Running time of previous algorithm is very high and needs improvement.
- The goal is not to compute all the $L^{(m)}$ matrices but only computation of matrix $L^{(n-1)}$ is of interest
- As Matrix multiplication associative, $L^{(n-1)}$ can be calculated with only $\lceil \lg(n - 1) \rceil$ matrix products as.

$$L^{(1)} = W, \qquad L^{(2)} = W^2 = W \cdot W,$$

$$L^{(4)} = W^4 = W^2 \cdot W^2, \qquad L^{(8)} = W^8 = W^4 \cdot W^4,$$

$$\vdots$$

$$L^{n-1} = L^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil}}$$

Improved Algorithm

FASTER-ALL-PAIRS-SHORTEST-PATHS (*W*)

1 $n \leftarrow rows[W]$
2 $L^{(1)} \leftarrow W$
3 $m \leftarrow 1$
4 **while** $m < n - 1$
5 **do** $L^{(2m)} \leftarrow$ EXTEND-SHORTEST-PATHS ($L^{(m)}$, $L^{(m)}$)
6 $m \leftarrow 2m$
7 **return** $L^{(m)}$

Total Running Time = $\Theta(n^3 \lg n)$

https://studyc.info/

Lecture 37 The Floyd-Warshall Algorithm and Johnson's Algorithm

Intermediate Vertices

- Vertices in G are given by V = {1, 2, …, n}
- Consider a path $p = \langle v_1, v_2, \ldots, v_l \rangle$
  - An **intermediate** vertex of p is any vertex in set $\{v_2, v_3, \ldots, v_{l-1}\}$

Example 1
If p = ⟨1, 2, 4, 5⟩ then
Intermediate vertex = {2, 4}

Example 2
If p = ⟨2, 4, 5⟩ then
Intermediate vertex = {4}

The Floyd Warshall Algorithm

Structure of a Shortest Path

- Let *V* = {1, 2,..., *n*} be a set of vertices of *G*
- Consider subset {1, 2,..., *k*} of vertices for some *k*
- *Let p be a* shortest paths from *i* to *j* with all intermediate vertices in the set {1, 2,..., *k*}.
- It exploits a relationship between path *p* and shortest paths from *i* to *j* with all intermediate vertices in the set {1, 2,..., *k* - 1}.
- The relationship depends on whether or not *k* is an intermediate vertex of path *p*.
- For both cases optimal structure is constructed

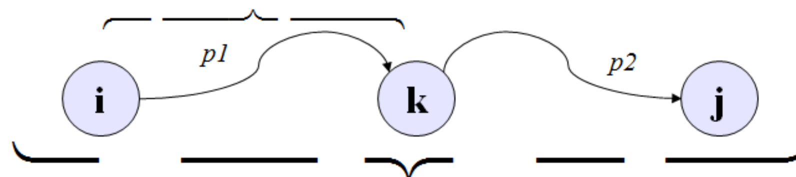
k not an Intermediate vertex of path p

- Shortest path i to j with I.V. from {1, 2, …, k} is shortest path i to j with I.V. from {1, 2, …, k - 1}

k an intermediate vertex of path p

- p1 is a shortest path from i to k
- p2 is a shortest path from k to j
- k is neither intermediate vertex of p1 nor of p2
- p1, p2 shortest paths i to k with I.V. from: {1, 2, …, k - 1}

all intermediate vertices in {1,…, k-1} **all intermediate vertices in {1,…, k-1}**



p: all intermediate vertices in {1,…,k}

https://studyc.info/

A Recursive Solution

- Let $d_{ij}^{(k)}$ = be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$.
- Now $D^{(n)} = (d_{i,j}^{(n)})$,
- Base case $d_{i,j}^{(0)}) = w_{i,j}$
- $D^{(0)} = (w_{i,j}) = W$
- The recursive definition is given below

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

The Floyd Warshall Algorithm

FLOYD-WARSHALL (W)
1 $n \leftarrow rows[W]$
2 $D^{(0)} \leftarrow W$
3 **for** $k \leftarrow 1$ **to** n
4 **do for** $i \leftarrow 1$ **to** n
5 **do for** $j \leftarrow 1$ **to** n
6 **do** $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
7 **return** $D^{(n)}$
Total Running Time = $\Theta(n^3)$

Constructing a Shortest Path

- One way is to compute matrix D of SP weights and then construct predecessor matrix Π from D matrix.
  - *It takes $O(n^3)$*
- A recursive formulation of $\pi_{ij}^{(k)}$
  - $k = 0$, shortest path from i to j has no intermediate vertex

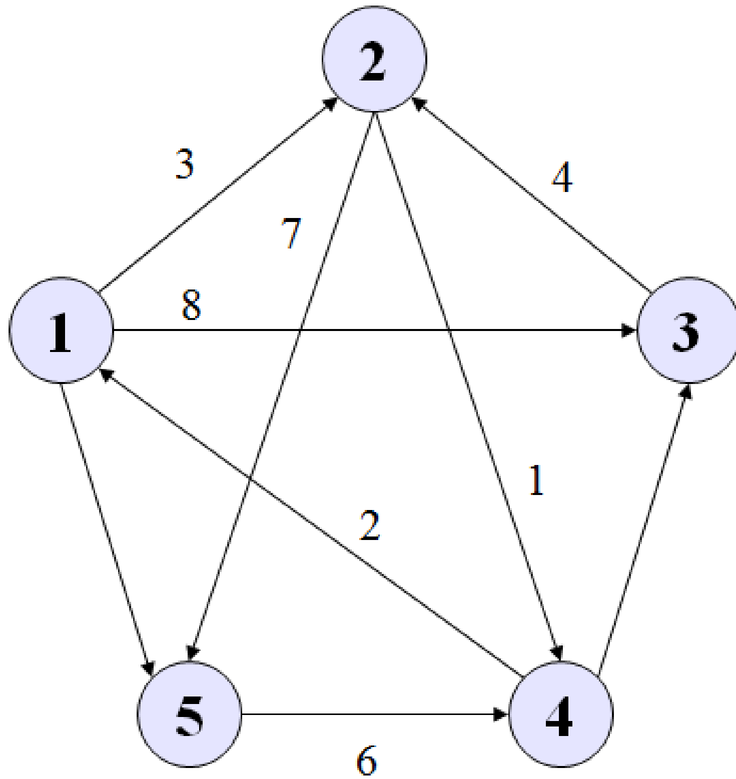
$$\pi_{ij}^{(0)} = \begin{cases} NIL & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

  - For $k \geq 1$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Example: Floyd Warshall Algorithm

https://studyc.info/



Adjacency matrix of above graph is given below.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

For $k = 0$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

For $k = 1$

https://studyc.info/

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

For $k = 2$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

For $k = 3$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

For $k = 4$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

For $k = 5$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

Existence of Shortest Paths between any Pair

Transitive Closure

https://studyc.info/

- Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2,...,n\}$, we may wish to find out whether there is a path in G from i to j for all vertex pairs $i, j \in V$.
- The **transitive closure** of G is defined as the graph $G^* = (V, E^*)$, where $E^* = \{(i, j)$: there is a path from vertex i to vertex j in $G\}$.
- One way is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm.
  - If there is a path from vertex i to j, then $d_{ij} < n$
  - Otherwise, we get $d_{ij} = \infty$.
  - The running time is $\Theta(n^3)$ time

Substitution

- Substitute logical operators, $\vee$ (for min) and $\wedge$ (for +) in the Floyd-Warshall algorithm
  - Running time: $\Theta(n^3)$ which saves time and space
  - A recursive definition is given by
  - k = 0 $\quad t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E \\ 1 & \text{if } i = j \text{ or } (i, j) \in E \end{cases}$
  - For k ≥ 1 $\quad t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge d_{kj}^{(k-1)}\right)$

Transitive Closure

TRANSITIVE-CLOSURE(G)

1 $n \leftarrow |V[G]|$
2 **for** $i \leftarrow 1$ **to** n
3 **do for** $j \leftarrow 1$ **to** n
4 **do if** $i = j$ or $(i, j) \in E[G]$
5 **then** $t_{ij}^{(0)} \leftarrow 1$
6 **else** $t_{ij}^{(0)} \leftarrow 0$
7 **for** $k \leftarrow 1$ **to** n
8 **do for** $i \leftarrow 1$ **to** n
9 **do for** $j \leftarrow 1$ **to** n
10 **do** $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge d_{kj}^{(k-1)}\right)$
11 **return** $T^{(n)}$

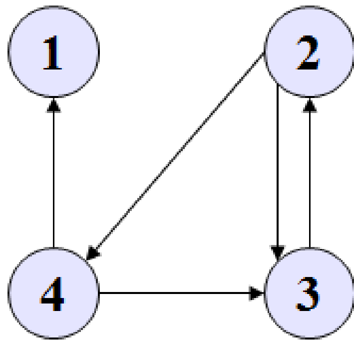
Total Running Time = Θ (n^3)

Transitive Closure

https://studyc.info/

$$T^{(0)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}\quad T^{(1)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&0\\1&0&1&1\end{pmatrix}\quad T^{(2)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&0&1&1\end{pmatrix}$$

$$T^{(3)}=\begin{pmatrix}1&0&0&0\\0&1&1&1\\0&1&1&1\\1&1&1&1\end{pmatrix}\quad T^{(4)}=\begin{pmatrix}1&0&0&0\\1&1&1&1\\1&1&1&1\\1&1&1&1\end{pmatrix}$$



Johnson's Algorithm

- For sparse graphs, Johnson's Algorithm is asymptotically better than
  - Repeated squaring of matrices and
  - The Floyd-Warshall algorithm.
- It uses as subroutines both
  - Dijkstra's algorithm and
  - The Bellman-Ford algorithm.
- It returns a matrix of shortest-path weights for all pairs of vertices OR
- Reports that the input graph contains a negative-weight cycle.
- This algorithm uses a technique of **reweighting**.

Re-weighting

- The technique of **reweighting** works as follows.
  - If all edge weights are nonnegative, find shortest paths by running Dijkstra's algorithm, with Fibonacci heap priority queue, once for each vertex.
  - If *G* has negative-weight edges, we simply compute a new set of nonnegative edges weights that allows us to use the same method.
- New set of edge weights must satisfy the following
  - For all pairs of vertices $u, v \in V$, a shortest path from u to v using weight function w is also a shortest path from u to v using weight function w'.
  - For all (u, v), new weight $w'(u, v)$ is nonnegative

δ, δ' Preserving Shortest Paths by Re-weighting

https://studyc.info/

- From the lemma given below, it is easy to come up with a re-weighting of the edges that satisfies the first property above.
- We use δ to denote shortest-path weights derived from weight function w
- And δ' to denote shortest-path weights derived from weight function w'.
- And then we will show that, for all (u, v), new weight $w'(u, v)$ is nonnegative.

Re-weighting does not change shortest paths

Lemma Statement

- Given a weighted, directed graph G = (V, E) with weight function w : E → R, let h : V → R be any function mapping vertices to real numbers.
- For each edge (u, v) E, define
  $$w'(u, v) = w(u, v) + h(u) - h(v)$$
- Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be any path from vertex v_0 to vertex v_k. Then p is a shortest path from v_0 to v_k with weight function w if and only if it is a shortest path with weight function w'.
- That is, $w(p) = \delta(v_0, v_k)$ if and only if $w'(p) = \delta'(v_0, v_k)$.
- Also, G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function w'.

Proof:

We start by showing that

$$w'(p) = w(p) + h(v_0) - h(v_k)$$

We have

$$w'(p) = \sum_{i=1}^{k} w'(v_{i-1}, v_i)$$

$$= \sum_{i=1}^{k} w(p) + h(v_{i-1}) - h(v_i)$$

$$= \sum_{i=1}^{k} w(p) + \sum_{i=1}^{k} (h(v_{i-1}) - h(v_i))$$

$$= \sum_{i=1}^{k} w(p) + h(v_0) - h(v_k)$$

Therefore, any path p from v_0 to v_k has $w'(p) = w(p) + h(v_0) - h(v_k)$.

If one path from v_0 to v_k is shorter than another using weight function w, then it is also shorter using w'.

Thus,

$$w(p) = \delta(v_0, v_k,) \Leftrightarrow w'(p) = \delta'(v_0, v_k,).$$

Finally, we show that G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function w'.

https://studyc.info/

Consider any cycle $c = <v_0, v_1,..., v_k>$, where $v_0 = v_k$.

Now $w'(c) = w(c) + h(v_0) - h(v_k) = w(c)$, and thus c has negative weight using w if and only if it has negative weight using w'. It completes the proof of the theorem

Producing nonnegative weights by re-weighting

Next we ensure that second property holds i.e. w'(u, v) to be nonnegative for all edges (u, v) $\in$ E.

- Given a weighted, directed graph G = (V, E) with weight function $w : E \rightarrow R$, we make a new graph G′ = (V′, E′), where $V' = V \cup \{s\}$ for some new vertex $s \notin V$ and
- $E' = E \cup \{(s, v) : v \in V\}$.
- Extend weight function w so that w(s, v) = 0 for all $v \in V$.
- Note that because s has no edges that enter it, no shortest paths in G′, other than those with source s, contain s.
- Moreover, G′ has no negative-weight cycles if and only if G has no negative-weight cycles.

Producing nonnegative weights by re-weighting

Now suppose that G and G′ have no negative-weight cycles.

- Let us define $h(v) = \delta(s, v)$ for all $v \in V'$.

By triangle inequality, we have

$$h(v) \leq h(u) + w(u, v), \quad \forall (u, v) \in E'. \qquad (1)$$

Thus, if we define the new weights w', we have $w'(u, v) = w(u, v) + h(u) - h(v) \geq 0$. by (1)

And the second property is satisfied.

Johnson's Algorithm

```
JOHNSON (G)
1  compute G′, where V [G′] = V [G] ∪ {s},
          E [G′] = E [G] ∪ {(s, v) : v ∈ V [G]}, and
          w(s, v) = 0 for all v ∈ V [G]
2  if BELLMAN-FORD(G′, w, s) = FALSE
3     then print "the input graph contains a negative-weight cycle"
4     else for each vertex v ∈ V [G′]
5              do set h(v) to the value of δ(s, v)
                     computed by the Bellman-Ford algorithm
6          for each edge (u, v) ∈ E [G′]
7              do w(u, v) = w(u, v) + h(u) - h(v)
8          for each vertex u ∈ V [G]
9              do run DIJKSTRA(G, w, u) to compute δ(u, v) for all v ∈ V [G]
```

https://studyc.info/

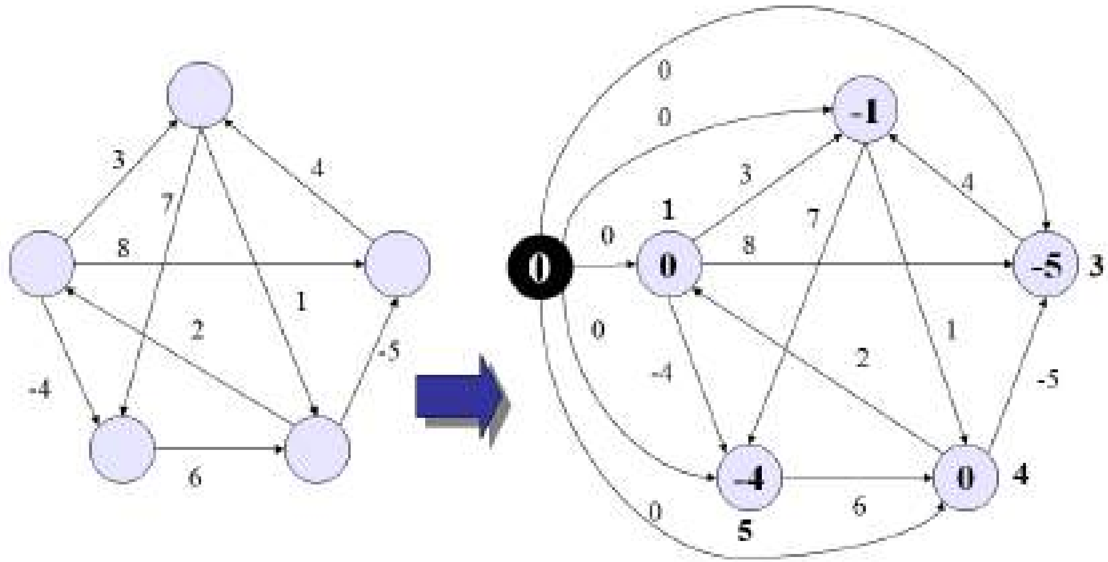
10 **for** each vertex $v \in V[G]$

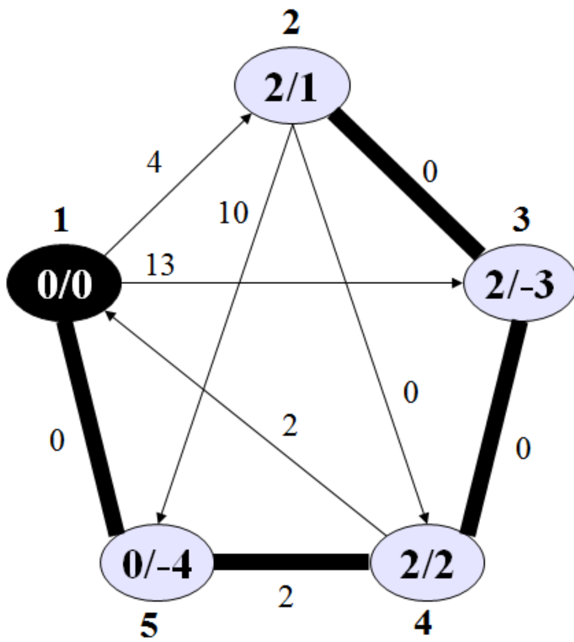
11 **do** $d_{uv} = \delta(u, v) + h(v) - h(u)$

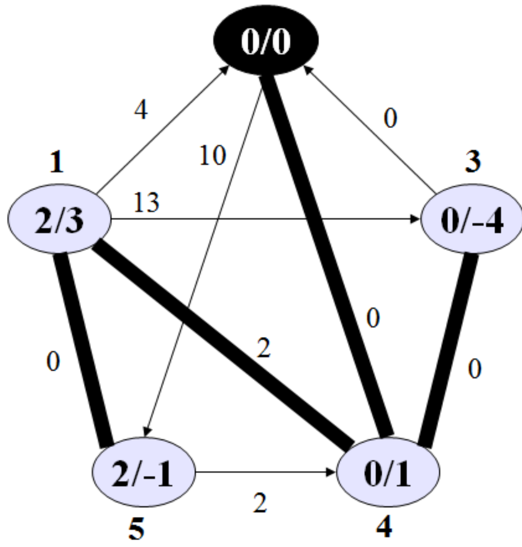
12 **return** D

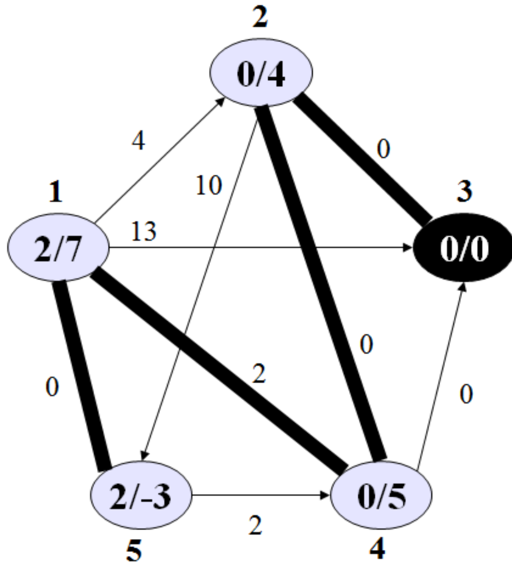
Total Running Time = $O(V^2 \lg V + VE)$

Johnson's Algorithm

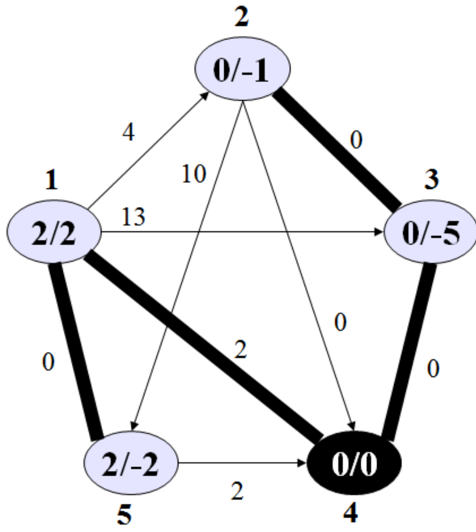


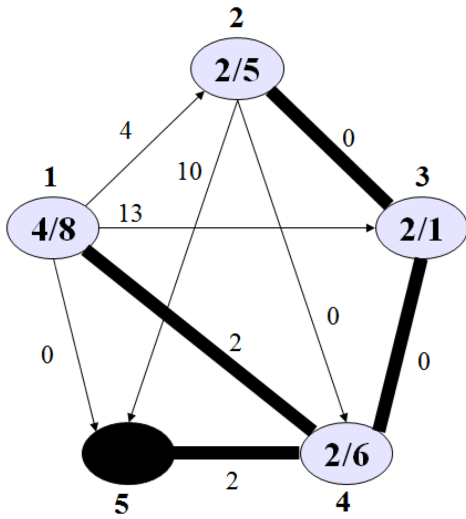






https://studyc.info/





https://studyc.info/

Lecture 38 Number Theoretic Algorithms

Applications of number theoretic algorithms

Electronic commerce

- Electronic commerce enables goods and services to be negotiated and exchanged electronically.
- The ability to keep information such as credit card numbers, passwords, and bank statements private is essential if electronic commerce is used widely.
- Public-key cryptography and digital signatures are among the core technologies used and are based on numerical algorithms and number theory.

Congruency equations modulo n

- For example, if we are given an equation $ax \equiv b \pmod{n}$, where a, b, and n are integers, and we wish to find all the integers x, modulo n, that satisfy the equation.
- There may be zero, one, or more solution.
- Using brute force approach, we can simply try $x = 0, 1, ..., n - 1$ in order, and can find the solution
- But our objective is not to find a solution only, in fact we want an efficient, of course this problem can be solved using number theory.

Numbers

- Z = set of all integers = . . .,-3, -2, -1, 0, +1, +2 +3, . . .
- Set of all even integers = $\{ 2k \mid k \in Z \}$
- Set of all odd integers = $\{ 2k + 1 \mid k \in Z \}$
- Q = Set of all rational numbers
  - p/q
  - $p, q \in Z$
  - $q \neq 0$
- I = set of all irrational numbers: which are not irrationals i.e.
  - $\sim p/q$ OR
  - $\sim(p, q \in Z)$ OR
  - $\sim(q \neq 0)$

Definitions

Divisibility

Let $a, b \in \mathbf{Z}$ with $a \neq 0$ then we say that

$$a|b \equiv a \text{ divides } b \mid \exists\, c \in \mathbf{Z} : b = ac$$

It means that $a|b$ if and only if there exists an integer c such that c times a equals b.

Example 1:

$3|(-12)$, because if we assume that a = 3, b = -12 then there exists c = -4 such that b = ac

https://studyc.info/

Example 2:

3 does not divide 7, because if we assume that a = 3, b = 7 then there does not exists any integer c such that b = ac

Some Facts

Statement: Prove that if $a|b$ then $a|(-b)$

Proof

Since $a|b$ hence there exist an integer x such that $b = ax$,
Now $-b = -ax = a(-x)$
Since if $x \in Z$ then $(-x) \in Z$, Hence, $a|(-b)$

Note: Because of the above lemma why not choose divisors as only positive. Hence if d is a divisor of b, then $1 \leq d \leq |b|$,

Example: Only divisors of 24 are: 1, 2, 3, 4, 6, 8, 12,and 24.

Statement: Prove that $a|0 \ \forall a \in \mathbf{Z}$

Proof

As we know that $a|b$ means there is an integer s such that $b = as$, and since $0 = a.0$, *where* 0 *is an integer, h*ence $a|0$

Statement: *If* $a|b$, $a|c$ then $a \mid (b + c) \ \forall$ a, b, c $\in \mathbf{Z}$

Proof

As we know that $a|b$ means there is an integer s such that $b = as$, and $a|c$ means that there is a t such that $c = at$. Now $b + c = as + at = a(s + t)$, and hence $a|(b + c)$.

Statement: *If* $a|b$, $a|c$ then $a \mid (b - c) \ \forall$ a, b, c $\in \mathbf{Z}$

Proof

If $a|b$ means then there is an integer s such that $b = as$, and i*f* $a|c$ then there is an integer t such that $c = at$. Now $b - c = as - at = a(s - t)$. Since if s, t $\in$ Z $\Rightarrow$ s - t $\in$ Z. Hence $a|(b - c)$.

Statement:

If $a|b$ and $b|a$ then prove that $a = \pm b$

Proof

Since $a|b$ hence there is an integer x such that $b = ax$, (1)
Since we are given that $b|a$ therefore there is an integer t such that $a = bt$, (2)
From (1) and (2), $a = axt \Rightarrow xt = 1$
Since x and t are both integers hence $x = t = \pm 1$

https://studyc.info/

Hence $a = \pm b$

Generalized Result

Statement: Prove that if $a|b$ then $a|bc$ $\forall$ a, b, c $\in$ **Z**

Proof

As we know that $a|b$ means there is an integer s such that $b = as$, and now $bc = asc = a(sc)$ and hence $a|bc$

Statement: Prove that if $a|b$, $b|c$ then $a|c$, $\forall$ a, b, c $\in$ **Z**

Proof:

Since $a|b$, it means $\exists$ s $\in$ **Z** such that $b = as$, *and* since $b|c$, it means $\exists$ t $\in$ **Z** such that $c = bt$. Now $c = bt = ast = a(st)$ and hence $a|c$

Statement:

$\forall$ a, b, c $\in$ Z, *if* $a|b$ *and* $a|c$ then $a \mid (bx + cy)$, $\forall$ x, y $\in$ Z

Proof

As we know that $a|b$ means there is an integer s such that $b = as \Rightarrow bx = asx$. If $a|c$ means that there is a t such that $c = at \Rightarrow cy = aty$. Now $bx + cy = asx + aty = a(sx + ty)$, and hence $a|(bx + cy)$, this is because $(sx + ty) \in Z$

Statement:

$\forall$ a, b, c $\in$ Z, *if* $a|b$ *and* $a|c$ then $a \mid (bx - cy)$, $\forall$ x, y $\in$ Z

Proof

As we know that $a|b$ therefore there is an integer s such that $b = as \Rightarrow bx = asx$, and since $a|c$ therefore there is an integer t such that $c = at \Rightarrow cy = aty$. Now $bx - cy = asx - aty = a(sx - ty)$, and hence $a|(bx - cy)$, this is because $(sx - ty) \in Z$

Prime Numbers

Definition:

A number p is prime if and only if it is divisible 1 and itself. OR A number p is prime if it can not be written as $p = x.y$ where $x, y \in Z$ and $x, y > 1$.

Note: 1 is prime, but is generally not of interest so we do not include it in set of primes

Examples: 2, 3, 5,7 etc. are all prime. 4, 6, 8, 9, 10 are not primes

- Prime numbers are central to number theory
- We will study some algorithms on prime numbers

https://studyc.info/

- List of prime number less than 200
  2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199
- There are infinitely many primes.
- Any positive integer that is not prime is composite.
- 1 is the "unit" for multiplication and we assume that it is neither prime nor composite.

The Division Algorithm

Statement:

- For any integer *dividend* a and *divisor* $d \neq 0$, there is a unique *quotient* q and *remainder* $r \in \mathbf{N}$ such that
  $a = dq + r,$ where $0 \leq r < |d|$
- In other way: $\forall\ a, d \in \mathbf{Z}, d > 0, \exists\ q, r \in \mathbf{Z}$ such that $0 \leq r < |d|$, and $a = d.q + r$
- We can find q by: $q = \lfloor a/d \rfloor$, and
- We can find r by: r = (a mod d) = a − dq

Example:

- a = 21; d = 4 then

  $q = \lfloor a/d \rfloor = \lfloor 21/4 \rfloor = 5$, and $r = a - dq = 21 - 4.5 = 1$

<u>Classification of Integers</u>

When an integer is divided by 2 remainder is 0 or 1

1. $C_1 = \{ 2k \mid k \in Z \}$ and
2. $C_2 = \{ 2k + 1 \mid k \in Z \}$

When an integer is divided by 3 remainder is 0, 1 or 2

1. $C_1 = \{ 2k \mid k \in Z \}$,
2. $C_2 = \{ 2k + 1 \mid k \in Z \}$ and
3. $C_3 = \{ 2k + 2 \mid k \in Z \}$

When an integer divided by 4 remainder, 0, 1, 2 or 3

1. $C_1 = \{ 2k \mid k \in Z \}$,
2. $C_2 = \{ 2k + 1 \mid k \in Z \}$
3. $C_3 = \{ 2k + 2 \mid k \in Z \}$
4. $C_4 = \{ 2k + 3 \mid k \in Z \}$ . . .

<u>Congruencies and Remainder</u>

<u>Remainder</u>

- When a is divided by n then we can write (a mod n) as remainder of this division.
- If, remainder when a is divisible by n = remainder when b is divisible by n then

https://studyc.info/

(a mod n) = (b mod n) e.g. (8 mod 3) = (11 mod 3)

Congruency

- If (a mod n) = (b mod n) we write it as $a \equiv b \pmod n$
  "a is equivalent to b modulo n."
- Thus, a and b have same remainder, w.r.t. n then
  $a = q_a n + r$, for some $q_a \in Z$
  $b = q_b n + r$, for some $q_b \in Z$

Lemma

If $a \equiv b \ (mod\ n)$ then prove that $n|(b - a)$

Proof:

Since $a \equiv b \ (mod\ n)$ *hence* (a mod n) = (b mod n)

Let (a mod n) = (b mod n) = r

By division theorem, $\exists\ q_1, q_2 \in Z$ such that

$a = q_1 n + r, \quad 0 \quad r < n$

$b = q_2 n + r, \quad 0 \quad r < n$

Now, $b - a = (q_2 - q_1)n + r - r = (q_2 - q_1)n$

Hence, $n|(b - a)$ because $(q_2 - q_1) \in Z$

Congruencies and Remainder

The equivalence of b class modulo n:

$[b]_n = \{b + kn : k \in Z\}$, *e. g.*

$[3]_7 = \{...,-11, -4, 3, 10, 17, ...\}$

$[-4]_7 = . . .$

$[17]_7 = . . .$

$Z_n = \{[a]_n : 0 \leq a \leq n - 1\}$ and we often write

Example

$Z_4 = \{[0]_4, [1]_4, [2]_4, [3]_4\}$

Usually we write Z4 = {0, 1, 2, 3}

$Z_n = \{0, 1, 2, 3, ..., n - 1\}$, we associate a with $[a]_n$.

Prime Factorization

- By **factorization** of a number n, we mean that $n = a \times b \times c$, where $a, b, c \in \mathbf{Z}$
- By a **prime factorization** of a number n, we mean that $n = a \times b \times c$, where $a, b, c \in \mathbf{Z}$ and all factors are prime number

Example:

https://studyc.info/

- 3600 = 24 × 32 × 52 factorization
- 3600 = $2^{10} \times 3^1 \times 13^1$ prime factorization
- 91 = 7×13 prime factorization

$$n = p_1^{m_1}.p_2^{m_2}...p_k^{m_k} = \prod_{i=1}^{k} p_i^{m_i}$$

Relatively Prime Numbers

Definition

Two numbers a, b are **relatively prime** if they have **no common divisors** except 1

Example

15, 23 are relatively prime, this is because

- Factors of 15 are 1, 3, 5, 15 and
- Factors of 23 are 1, 23 and
- Common factor is only 1
- Hence 15 and 23 are relatively prime

Some More Results

Definition: The greatest common divisor of a and b, not both zero, is the largest common divisor of a and b

Some elementary gcd properties

gcd(a, b) = gcd(b, a), gcd(a, b) = gcd(-a, b)

gcd(a, b) = gcd(|a|, |b|), gcd(a, 0) = |a|

Examples

gcd(24, 30) = 6, gcd(5, 7) = 1

gcd(0, 9) = 9, gcd(0, 0) = 0 by definition

$\gcd(a, ka) = |a|$ *for any* $k \in Z$.

Note: $1 \leq \gcd(a, b) \leq \min(|a|, |b|)$

Example: Greatest Common Divisor

GCD of two integers can be obtained by comparing their prime factorizations and using least powers

Example

- 600 = 2 x 2 × 2 x 3 × 5 x 5 = $2^3 \times 3^1 \times 5^2$
- 36 = 2 x 2 × 3 x 3 = $2^2 \times 3^2$
- Rearrange the factorization of 600 and 36
- 600 = 2 x 2 × 2 x 3 × 5 x 5 = $2^3 \times 3^1 \times 5^2$
- 36 = 2 x 2 × 3 x 3 = $2^2 \times 3^2 \times 5^0$
- GCD(36, 600) = $2^{\min(3,2)} \times 3^{\min(1,2)} \times 5^{\min(2,0)}$

  = $2^2 \times 3^1 \times 5^0$

  = 4 x 3

https://studyc.info/

= 12

Brute Force Approach Finding GCD

Statement:

- Given two integers a and b. Find their greatest common divisor

Method:

- Compute prime factorization of a

$$a = p_1^{m_1}.p_2^{m_2}...p_k^{m_k} = \prod_{i=1}^{k} p_i^{m_i}$$

- Compute prime factorization of b

$$b = p_1^{n_1}.p_2^{n_2}...p_l^{n_l} = \prod_{i=1}^{l} p_i^{n_i}$$

- Let $p_1, p_2, \ldots, p_t$ be the set of all prime numbers both in the factorization of a and b

$$a = p_1^{m_1}.p_2^{m_2}...p_t^{m_t} = \prod_{i=1}^{t} p_i^{m_i}, where\ p_1 < p_2 < ... < p_t$$

- Now the prime factorization of a is rearranged as

$$a = p_1^{m_1}.p_2^{m_2}...p_t^{m_t} = \prod_{i=1}^{t} p_i^{m_i}, where\ p_1 < p_2 < ... < p_t$$

- Similarly the prime factorization of b is rearranged as

$$b = p_1^{n_1}.p_2^{n_2}...p_t^{n_t} = \prod_{i=1}^{t} p_i^{n_i}, \text{ where } \mathrm{p}_1 < p_2 < ... < p_t$$

- Finally GCD of a and b can be computed as

$$\gcd(a,b) = p_1^{\min(m_1,n_1)}.p_2^{\min(m_2,n_2)}....p_t^{\min(m_t,n_t)}$$

$$\gcd(a,b) = \prod_{i=1}^{t} p_i^{\min(m_i,n_i)}.$$

Methods of Proof

Direct Method:

- Express the statement to be proved in the form:
  $\forall$ x $\in$ D, P(x) $\Rightarrow$ Q(x)
- Suppose that P(x) is true for an arbitrary element x of D
- Prove that Q(x) is true for the supposed above value x of D.

Parity:

Two integers have same parity if both are either odd or even, otherwise opposite parity.

Direct Proof

Lemma:

https://studyc.info/

Prove that m + n and m – n have same parity, for all $m, n \in Z$

Proof: There are three cases

Case 1:

Both m, n are even i.e.,

$m = 2k_1$ and $n = 2k_2$ for some $k_1, k_2 \in Z$

Now, $m + n = 2k_1 + 2k_2 = 2(k_1 + k_2)$ an even

And, $m - n = 2k_1 - 2k_2 = 2(k_1 - k_2)$ an even

Case 2:

Both m, n are odd i.e.,

$m = 2k_1 + 1$ and $n = 2k_2 + 1$ for some $k_1, k_2 \in Z$

Now, $m + n = 2k_1 + 1 + 2k_2 + 1 = 2(k_1 + k_2 + 1) = 2k'$

And, $m - n = 2k_1 + 1 - 2k_2 - 1 = 2(k_1 - k_2) = 2k''$

Hence m + n and m - n both are even

Case 3:

m is even and n is odd i.e.,

$m = 2k_1$ and $n = 2k_2 + 1$ for some $k_1, k_2 \in Z$

Now, $m + n = 2k_1 + 2k_2 + 1 = 2(k_1 + k_2) + 1 = 2k' + 1$, odd

And, $m - n = 2k_1 - 2k_2 - 1 = 2(k_1 - k_2 - 1) + 1 = 2k'' + 1$, odd

Hence m + n and m - n both have the same parity.

An Alternate Method of Direct Proof

We can formulate the same problem as

Notations

- Let S-EVEN (m, n) ≡ m + n is even
- Let S-ODD (m, n) ≡ m + n is odd
- Let D-EVEN (m, n) ≡ m - n is even
- Let D-ODD (m, n) ≡ m - n is odd

Mathematical Statement of the problem

- S-EVEN (m, n) ⇔ D-EVEN (m, n), $\forall m, n \in Z$
- S-ODD (m, n) ⇔ D-ODD (m, n), $\forall m, n \in Z$

Proof

Case 1

- Suppose that S-EVEN (m, n), $\forall m, n \in Z$
- Now, m – n = m + n – 2n = even - even = even integer
- Hence D-EVEN (m, n), $\forall m, n \in Z$

https://studyc.info/

Case 2

- Suppose that D-EVEN (m, n), $\forall$ m, n $\in$ Z
- Now, m + n = m - n + 2n = even + even = an even integer, $\forall$ m, n $\in$ Z
- Hence S-EVEN (m, n), $\forall$ m, n $\in$ Z

Case 3

- Suppose that S-ODD (m, n), $\forall$ m, n $\in$ Z
- Now, m – n = m + n – 2n = odd – even = odd
- D-ODD (m, n), $\forall$ m, n $\in$ Z

Case 4

- Suppose that D-ODD (m, n), $\forall$ m, n $\in$ Z
- Now, m + n = m - n + 2n = odd + even = odd
- S-ODD (m, n), $\forall$ m, n $\in$ Z

Hence

- S-EVEN (m, n) $\Leftrightarrow$ D-EVEN (m, n), $\forall$ m, n $\in$ Z
- S-ODD (m, n) $\Leftrightarrow$ D-ODD (m, n), $\forall$ m, n $\in$ Z

Disproof by Counter Example

To disprove a statement of the form:
$\forall$ x $\in$ D, P(x) $\Rightarrow$ Q(x)

- Find a value of x in D for which P(x) is true and Q(x) is false.
- Such an example is called counter example.

Example: Prove or disprove

$$\forall a, b \in Z, a^2 = b^2 \Rightarrow a = b$$

Disproof:

Let $P(a, b) \equiv a^2 = b^2$, $Q(a, b) \equiv a = b$,
Now $P(1, -1) \equiv (1)^2 = (-1)^2$ true but $Q(1, -1) \equiv 1 \neq -1$

Method of Proof by Contradiction

Steps in proving by contradiction

- Suppose the statement to be proved is false
- Show that this supposition leads logically to a contradiction
- Conclude that the statement to be proved is true

Example: Prove that sum of an irrational and a rational number is irrational

Proof: Suppose a is a rational, b is an irrational number and their sum is also rational

- Since a is rational, a = p/q, p, q $\in$ Z and q $\neq$ 0

https://studyc.info/

- Now according to our supposition a + b is rational and hence it can be written as

  $a + b = m/n$, where $m, n \in Z$ and $n \neq 0$
- Now consider $a + b = m/n$

$\Rightarrow$ $b = m/n - a = m/n - p/q = (mp - nq)/nq = r/s$, where $r, s \in Z$ and $s \neq 0$

$\Rightarrow$ b is a rational number, which is contradiction.

$\Rightarrow$ Hence sum of an irrational and a rational number is always irrational.

Lemma

For any integer n and any prime number p, if p divides n then p does not divide n + 1

Proof

- Express the statement in the form: $\forall x \in D, P(x) \Rightarrow Q(x)$
- Let, Z = set of all integers, and P = set of all primes
- $D(p, n) \equiv$ p divides n
- $DN(p, n) \equiv$ p does not divide n
- Now our problem becomes

  $\forall n \in Z, p \in P, D(p, n) \Rightarrow DN(p, n + 1)$
- Suppose that for some integer n and prime p, p divides $n \equiv D(p, n)$
- Now we have to prove that p does not divide n + 1
- On contrary we suppose that p divide n + 1
- It means that there exists an integer q_1 such that $n + 1 = pq_1$
- Since p divides n. Hence there exists an integer q_2 such that $n = pq_2$
- Now, $n + 1 - n = pq_1 - pq_2$

  $1 = pq_1 - pq_2 = p(q_1 - q_2) \Rightarrow p = 1$ or -1 contradiction
- Hence p does not divide $n + 1 \equiv DN(p, n)$

https://studyc.info/

Lecture 39 Number Theoretic Algorithms

Method of Proof by Contraposition

Steps in proving by contraposition

- Express the statement to be proved in the form: $\forall x \in D, P(x) \Rightarrow Q(x)$
- Rewrite the statement in the contrapositive form: $\forall x \in D, \neg Q(x) \Rightarrow \neg P(x)$
- Prove the contrapositive by direct proof
  - Suppose that x is an arbitrary but particular element of D such that Q(x) is false
  - Show that P(x) is false

Examples: Proof by Contraposition

Example 1: Prove that for all integers n, if n^2 is even then n is also even

Proof

Express the above statement in the form: $\forall x \in D, P(x) \Rightarrow Q(x)$
Suppose that

$D = Z$,

$Even(n, 2) \equiv n^2$ is even

$Even(n) \equiv n$ is even

- We have to prove that $\forall n \in Z, Even(n, 2) \Rightarrow Even(n)$
- Contraposition of the above statement

  $\forall n \in Z, \neg Even(n) \Rightarrow \neg Even(n, 2)$ is even
- Now we prove above contrapositive by direct proof
- Suppose that n is an arbitrary element of Z such that, $\neg Even(n)$ (n is not even) i.e., n is odd
- $n^2 = n.n = odd. odd = odd$
- n^2 is odd
- $\neg Even(n, 2)$ is even
- Hence, $\forall n \in Z, \neg Even(n) \Rightarrow \neg Even(n, 2)$ is even
- Therefore, $\forall n \in Z, Even(n, 2) \Rightarrow Even(n)$ is even
- Hence $\forall n \in Z$, if n^2 is even then n is even

Example 2: Prove that for all integers n, if n^2 is divisible by 7 then n is divisible by 7.

Proof

- Express the above statement in the form: $\forall x \in D, P(x) \Rightarrow Q(x)$
- Suppose that

  $D = Z$,

  $Div(n, 2, 7) \equiv n^2$ is divisible by 7

  $Div(n, 7) \equiv n$ is divisible by 7
- We have to prove that $\forall n \in Z, Div(n, 2, 7) \Rightarrow Div(n, 7)$
- Contraposition of the above statement

  $\forall n \in Z, \neg Div(n, 7) \Rightarrow \neg Div(n, 2, 7)$

https://studyc.info/

- Now we prove above contrapositive by direct proof
- Suppose that n is an arbitrary element of Z such that, $\neg$ Div(n, 7) (n is not divisible by 7)
- n does contain any factor of 7
- n^2 does contain any factor of 7
- Hence, $\forall$ n $\in$ Z, $\neg$ Div(n, 7) $\Rightarrow$ $\neg$ Div(n, 2, 7)
- Therefore, $\forall$ n $\in$ Z, Div(n, 2, 7) $\Rightarrow$ Div(n, 7)
- Hence, $\forall$ n $\in$ Z, if n^2 is divisible by 7 then n is divisible by 7.

Lemma 1: Statement : The square of an odd integer is of the form 8m + 1 for some integer m.

Proof:

- Suppose n is an arbitrary odd integer.
- By quotient remainder theorem any integer has the form
  4m, 4m + 1, 4m + 2 OR 4m+3
- Now since n is an odd integer, hence n can be represented as
  4m + 1 OR 4m+3
- Now we have to prove that squares of 4m + 1 and 4m + 3 are of the form 8m + 1.

Case 1

Square of 4m + 1

$(4m + 1)^2 = 16m^2 + 8m + 1 = 8(2m^2 + m) + 1$

$= 8m' + 1$, where $m' = (2m^2 + m)$

Case 2

Square of 4m + 3

$(4m + 3)^2 = 16m^2 + 24m + 9 = 8(2m^2 + 3m + 1) + 1$

$= 8m'' + 1$, where $m'' = (2m^2 + 3m + 1)$

- Hence any odd integer has the form 8m + 1 for some m

Theorem 1

Statement: If *a* and *b* are any integers, not both zero, then gcd(*a*, *b*) is the smallest positive element of the set $\{ax + by : x, y \in \mathbf{Z}\}$ of linear combinations of *a* and *b.*

Proof

Let *s* be the smallest positive such linear combination of *a* and *b*, i.e.

$s = ax + by,$ for some $x, y \in Z$

By quotient remainder theorem

a = qs + r = qs + a mod s, where $q = \lfloor a/s \rfloor$.

$a \bmod s = a - qs = a - q(ax + by) = a\,(1 - qx) + b(-qy)$

- Hence ***a* mod *s*** is a linear combination of *a* and *b*.
- But, since *a* mod *s* < *s*, therefore, *a* mod *s* = 0
- Now *a* mod *s* = 0 $\Rightarrow$ $s \mid a$

https://studyc.info/

- Similarly we can prove that, $s \mid b$.
- Thus, s is a common divisor of both a and b,
- Therefore, $s \leq \gcd(a, b)$ (1)
- *We know that if $d \mid a$ and $d \mid b$ then $d \mid ax + by$ for all x, y integers.*
- Since $\gcd(a, b) \mid a$ and $\gcd(a, b) \mid b$, hence $\gcd(a, b) \mid s$ and $s > 0$ imply that $\gcd(a, b) \leq s$. (2)
- By (1) and (2), $\gcd(a, b) = s$

Corollary

Statement: For all integers a and b and any nonnegative integer n, $\gcd(an, bn) = n \gcd(a, b)$.

Proof

- If $n = 0$, the corollary is trivial.
- If $n > 0$, then $\gcd(an, bn)$ is the smallest positive element of the set $\{anx + bny\}$, i.e.
  $\gcd(an, bn) = \min \{anx + bny\} = \min\{n.\{ax + by\}\}$
  $= n. \min\{ax + by\}$
  n times smallest positive element of set $\{ax + by\}$.
- Hence $\gcd(an, bn) = n.\gcd(x, y)$

Relatively Prime Integers

- Two integers a, b are said to be ***relatively prime*** if their only common divisor is 1, i. e, if $\gcd(a, b) = 1$.

Generalized Form of Relatively Prime Integers

- We say that integers $n_1, n_2, ..., n_k$ are ***pairwise relatively prime*** if, whenever $i \neq j$, we have $\gcd(n_i, n_j) = 1$.

Lemma 2

Statement: For any integers a, b, and p, if both $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, then $\gcd(ab, p) = 1$.

Proof

- As, $\gcd(a, p) = 1$, there exist integers x, y *such that*
  $ax + py = 1$ (1)
- $\gcd(b, p) = 1$, there exist integers x', y' *such that* $bx' + py' = 1$ (2)
- Multiplying equations (1) and (2) and rearranging, $ab(x x') + p(ybx' + y'ax + pyy') = 1$, $abx'' + py'' = 1$
- Since 1 is a positive linear combination of ab and p,
- Hence $\gcd(ab , p) = 1$,**which completes the proof**

Lemma 3

Statement: For all primes p and all integers a, b, if $p \mid ab$, then $p \mid a$ or $p \mid b$ (or p divides both and b).

https://studyc.info/

Proof

- Let P = set of all primes; Z = set of all integers
- P(p, ab) ≡ p | ab; Q(p, a, b) ≡ p | a or p | b
- Express above statement to be proved in the form:
  $\forall$ a, b, p, P(p, ab) $\Rightarrow$ Q(p, a, b)
- $\forall$ p $\in$ P, a, b $\in$ Z, p | ab $\Rightarrow$ (p | a or p | b)
- Assume for the purpose of contradiction that p | ab but that p ∤ a and p ∤ b.
- Now p ∤ a $\Rightarrow$ gcd(a, p) = 1
- And, p ∤ b $\Rightarrow$ gcd(b, p) = 1
- Since only divisors of p are 1 and p, and by assumption p divides neither a nor b.
- **Above Lemma 2, states that for any integers a, b, and p, if both gcd(a, p) = 1 and gcd(b, p) = 1, then gcd(ab, p) = 1.**
- Now, gcd(ab, p) = 1, contradicts our assumption that p | ab, since p | ab implies gcd(ab, p) = p.
- This contradiction completes the proof.

<u>Theorem 2: GCD Recursion Theorem</u>

Statement

- For any nonnegative integer a and any positive integer b, gcd(a, b) = gcd(b, a mod b).

Proof

- If we will be able to prove that gcd(a, b) and gcd(b, a mod b) divide each other, It will complete the proof of the theorem. This is because both are nonnegative.

Case 1

- We first show that gcd(a, b) | gcd(b, a mod b).
- If we let d = gcd(a, b).
- By quotient remainder theorem:
  (a mod b) = a - qb, where $q = \lfloor a/b \rfloor$.
- Now d = gcd(a, b) $\Rightarrow$
  - d | a and
  - d | b,
- Hence, d | (a – qb),
  (this is because, a – qb is a linear combination of a and b, where x = 1, y = -q)
- And consequently **d | (*a* mod *b*),**
  this is because (a mod b = a – qb)
- Now, d | b and d | (a mod b), implies that:
  d | gcd(b, a mod b)
- Hence gcd(a, b) | gcd(a, a mod b).   (A)

Case 2

- We now show that: gcd(a, a mod b) | gcd(a, b).
- If we let, d = gcd(b, a mod b), then

https://studyc.info/

$d \mid b$ and

$d \mid (a \bmod b)$.

- By quotient remainder theorem

  $a = qb + (a \bmod b)$, where $q = \lfloor a/b \rfloor$,
- a is a linear combination of b and a mod b, $\Rightarrow$ d | a
- Now, d | a and d | b $\Rightarrow$ d | gcd(a, b)
- Hence, gcd(a, a mod b) | gcd(a, b) (B)
- By (A) and (B):

  gcd(a, b) = gcd(b, a mod b).

Example: Compute gcd (1970, 1066)

a = 1970, b = 1066

1970 = 1 x 1066 + 904 = gcd(1066, 904), R = 904

1066 = 1 x 904 + 162 = gcd(904, 162), R = 162

904 = 5 x 162 + 94 = gcd(162, 94), R = 94

162 = 1 x 94 + 68 = gcd(94, 68), R = 68

94 = 1 x 68 + 26 = gcd(68, 26), R = 26

68 = 2 x 26 + 16 = gcd(26, 16), R = 16

26 = 1 x 16 + 10 = gcd(16, 10), R = 10

16 = 1 x 10 + 6 = gcd(10, 6), R = 6

10 = 1 x 6 + 4 = gcd(6, 4), R = 4

6 = 1 x 4 + 2 = gcd(4, 2), R = 2

4 = 2 x 2 + 0 = gcd(2, 0), R = 0

Hence gcd(1970, 1066) = 2

Euclid's Algorithm

EUCLID(a, b)

1 **if** $b = 0$

2 **then return** a

3 **else return** EUCLID(b, a mod b)

Example: Compute the gcd of 30 and 21

Solution

EUCLID(30, 21) = EUCLID(21, 9) = EUCLID(9, 3) = EUCLID(3, 0) = 3

- Here, there are three recursive invocations of EUCLID.
- The correctness of EUCLID follows from Theorem 2
- And the fact that if the algorithm returns a in line 2, then $b = 0$, and so gcd(a, b) = gcd(a, 0) = a

Note:

- The algorithm cannot recurse indefinitely
- This is because the second argument strictly decreases in each recursive call
- And this second argument is also always nonnegative.

https://studyc.info/

- Hence it must be 0 after some number of calls
- Therefore, EUCLID always terminates with the correct answer.

Running Time of Euclid's Algorithm

- We analyze the worst-case running time of EUCLID as a function of the size of *a* and *b*.
- We assume without loss of generality that $a > b \geq 0$.
- This assumption justified because if $b > a \geq 0$, then EUCLID(a, b) makes recursive call EUCLID(b, a).
- That is, if first argument is less than second one, EUCLID spends one recursive call swapping a, b
- Similarly, if $b = a > 0$, the procedure terminates after one recursive call, since a mod b = 0.
- The overall running time of EUCLID is proportional to the number of recursive calls it makes.
- Our analysis makes use of the Fibonacci numbers F_k, defined earlier in the first part of our course

Running Time of Euclid's Algorithm

Statement

If $a > b \geq 1$ and the invocation EUCLID(a, b) takes $k \geq 1$ recursive calls, then $a \geq F_{k+2}$ and $b \geq F_{k+1}$.

Proof

The proof is by induction on *k*.

Case 1

- For base case, let $k = 1$. Then, $b \geq 1 = F2$, and since $a > b$, we must have $a \geq 2 = F3$.
- Hence the statement is true for k = 1
- Please note that, $b > (a \bmod b)$, in each recursive call, i.e., first argument is strictly larger than the second and hence the assumption that $a > b$ therefore holds for each recursive call.

Case 2

- Now suppose that the lemma is true for $k – 1$ i.e., if $a > b \geq 1$ and invocation EUCLID(a, b) takes $k\text{-}1 \geq 1$ recursive calls, then $a \geq F_{k+1}$ and $b \geq F_k$.

Case 3

- Now we have to prove that statement is true for k i.e. if $a > b \geq 1$ and invocation EUCLID(a, b) takes $k \geq 1$ recursive calls, then $a \geq F_{k+2}$ and $b \geq F_{k+1}$.
- Since $k > 0$, and $b > 0$, and EUCLID(a, b) calls EUCLID(b, a mod b) recursively, which in turn makes k - 1 recursive calls.
- Since we know that statement is true for k-1, hence $b \geq F_{k+1}$, and (a mod b) $\geq Fk$.
- Now we have

https://studyc.info/

$b + (a \bmod b) = b + (a - \lfloor a/b \rfloor b)$ (1)

- Since, $a > b > 0$, therefore, $\lfloor a/b \rfloor \geq 1$

  $\Rightarrow \lfloor a/b \rfloor b \geq b$

  $\Rightarrow b - \lfloor a/b \rfloor b \leq 0$

  $\Rightarrow a + b - \lfloor a/b \rfloor b \leq 0 + a$

  $\Rightarrow b + (a - \lfloor a/b \rfloor b) \leq a$
- By (1), $b + (a \bmod b) = b + (a - \lfloor a/b \rfloor b) \leq a$
- $b + (a \bmod b) \leq a$
- Thus, $a \geq b + (a \bmod b) \geq F_{k+1} + F_k = F_{k+2}$.
- Hence, $a \geq F_{k+2}$, It completes proof of the theorem

Extended Euclid's Algorithm

EXTENDED-EUCLID(a, b)

```
1   if b = 0
2       then return (a, 1, 0)
3   (d', x', y') ← EXTENDED-EUCLID(b, a mod b)
4   (d, x, y) ← (d', y', x' - ⌊ a/b⌋ y')
5   return (d, x, y)
```

Proof of Correctness

$d' = bx' + (a \bmod b)y'$

$d = bx' + (a - \lfloor a/b \rfloor b)y'$ $\gcd(a, b) = \gcd(b, a \bmod b)$

$d = ay' + b(x' - \lfloor a/b \rfloor y')$

Reduced set of residues mod n

- **Complete set of residues** is: 0 . . . n-1
- **Reduced set of residues** consists of all those numbers (residues) which are relatively prime to n
- And it is denoted by

  $Z_n^* = \{k : \gcd(k, n) = 1, 0 \leq k < n\}$
- The number of elements in reduced set of residues is called the **Euler Totient Function φ(n)**

Example

- For n = 10, find reduced list of residues of n
- All residues: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Reduced residues (primes) = {1, 3, 7, 9}, $\varphi(n) = 4$

Group

Definition of a Group

Group is a set, G, together with a binary operation : $G * G \rightarrow G$, usually denoted by $a*b$, such that the following properties are satisfied :

https://studyc.info/

- Associativity :
  $(a*b)*c = a*(b*c)$ for all a, b, c $\in$ G
- Identity :
  $\exists$ e $\in$ G, such that $e*g = g = g*e$ for all g $\in$ G.
- Inverse :
  For each g $\in$ G, there exists the g', inverse of g, such that $g'*g = g*g' = e$

The Multiplicative Group Z^*_n

$Zn^* = \{k : \gcd(k, n) = 1, 1 \leq k < n\}$

For any positive integer *n*, Z*n** forms a group under multiplication modulo *n*.

Proof:

- Binary Operation
  Let a, b $\in$Zn*, gcd(a, n) = 1; gcd(b, n) = 1
  gcd(ab, n) = gcd(a, n)*gcd(b,n) = 1*1 = 1
- Associativity holds,
- 1 is the identity element.
- inverse of each element exits
  Hence (Z*n** ,*) forms a group.

Rings

Definition

A **ring** is a set R with two binary operations $+ : R \times R \rightarrow R$ and $\cdot : R \times R \rightarrow R$ (where $\times$ denotes the Cartesian product), called *addition* and *multiplication*, such that:

- (R, +) is an abelian group with identity element 0
  1. $(a + b) + c = a + (b + c)$
  2. $0 + a = a + 0 = a$
  3. For every a in R, there exists an element denoted $-a$, such that $a + -a = -a + a = 0$
  4. $a + b = b + a$
- (R, $\cdot$) is a monoid with identity element 1:
  1. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
  2. $1 \cdot a = a \cdot 1 = a$
- Multiplication distributes over addition:
  1. $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
  2. $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$

Note

- Ring addition is commutative so that $a + b = b + a$
- *But* ring with multiplication is not required to be commutative i.e. $a \cdot b$ need not equal $b \cdot a$.

https://studyc.info/

- Rings that satisfy commutative property for multiplication are called **commutative rings.**

Not all rings are commutative.

- Rings need not have multiplicative inverses either.
- An element a in a ring is called a **unit** if it is invertible with respect to multiplication
- An element a is called invertible under multiplication if there is an element b in the ring such that $a \cdot b = b \cdot a = 1$,
- This b is uniquely determined by a and we write $a^{-1} = b$.

Lemma: The set of all units in R forms a group under ring multiplication

Example: Rings

Example: Prove that Z (+, *) (the set of integers) is a ring.

Solution: + and * are binary operation on Z because sum and product of two integers are also an integer

- Now, $\forall$ a, b, c $\in$ Z
  1. $(a + b) + c = a + (b + c)$,
  2. $0 + a = a + 0 = a$
  3. $a + (-a) = (-a) + a = 0$
  4. $a + b = b + a$

  Hence $(Z, +)$ is an abelian group with identity element 0

- Since, $\forall$ a, b, c $\in$ Z
  1. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
  2. $1 \cdot a = a \cdot 1 = a$

  Hence $(Z, \cdot)$ is a monoid with identity element 1

- Finally $\forall$ a, b, c $\in$ Z
  1. $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
  2. $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$

  i.e., multiplication is distributive over addition

Hence we can conclude that Z (+, *) is a ring

https://studyc.info/

Lecture 40 Chinese Remainder Theorem and RSA Cryptosystem

Addition: Modulo 8

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

Multiplication: Modulo 8

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

Reduced set of residues mod n

- **Complete set of residues** is $Z_n = \{0, 1, \ldots, n-1\}$
- **Reduced set of residues** consists of all those numbers (residues) which are relatively prime to n
- And it is denoted by
  $Z_{n^*} = \{k : \gcd(k, n) = 1, 0 \le k < n\}$
- The number of elements in reduced set of residues is called the **Euler Totient Function φ(n)**

Example 1

- For n = 10, find reduced list of residues of n
- All residues: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Reduced residues (primes) = {1, 3, 7, 9}, φ**(n)** = 4

Group

https://studyc.info/

Definition of a Group

Group is a set, G, together with a binary operation: $G * G \to G$, usually denoted by a*b, such that the following properties are satisfied :

- Associativity :
  $(a*b)*c = a*(b*c)$ for all $a, b, c \in G$
- Identity :
  $\exists\, e \in G$, such that $e*g = g = g*e$ for all $g \in G$.
- Inverse :
  For each $g \in G$, there exists the g', inverse of g, such that $g'*g = g*g' = e$

Result: The Multiplicative Group Z_n*

Statement: $Zn^* = \{k : \gcd(k, n) = 1, 1 \leq k < n\}$. For any positive integer n, Zn^* forms a group under multiplication modulo n.

Proof:

- Binary Operation
  Let $a, b \in Zn^*$, $\gcd(a, n) = 1$; $\gcd(b, n) = 1$
  $\gcd(ab, n) = \gcd(a, n)*\gcd(b,n) = 1*1 = 1$
- Associativity holds,
- 1 is the identity element.
- inverse of each element exits
  Hence (Zn^* ,*) forms a group.

Rings

Definition

A **ring** is a set R with two binary operations $+ : R \times R \to R$ and $\cdot : R \times R \to R$ (where $\times$ denotes the Cartesian product), called *addition* and *multiplication*, such that:

- $(R, +)$ is an abelian group with identity element 0
  1. $(a + b) + c = a + (b + c)$
  2. $0 + a = a + 0 = a$
  3. For every a in R, there exists an element denoted $-a$, such that $a + -a = -a + a = 0$
  4. $a + b = b + a$
- $(R, \cdot)$ is a monoid with identity element 1:
  1. $(a\cdot b)\cdot c = a\cdot(b\cdot c)$
  2. $1\cdot a = a\cdot 1 = a$
- Multiplication distributes over addition:
  1. $a\cdot(b + c) = (a\cdot b) + (a\cdot c)$
  2. $(a + b)\cdot c = (a\cdot c) + (b\cdot c)$

Definition:

https://studyc.info/

An element a in a ring R is called **unit** if there exists b in R such that $a \cdot b = b \cdot a = 1$

Lemma: Set of all units in R forms a group under ring multiplication

Example 2: Prove that Z (+, *) (the set of integers) is a ring.

Solution: + and * are binary operation on Z because sum and product of two integers are also an integer

- Now, $\forall$ a, b, c $\in$ Z
  1. $(a + b) + c = a + (b + c)$,
  2. $0 + a = a + 0 = a$
  3. $a + (-a) = (-a) + a = 0$
  4. $a + b = b + a$

  Hence (Z, +) is an abelian group with identity element 0

- Since, $\forall$ a, b, c $\in$ Z
  1. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
  2. $1 \cdot a = a \cdot 1 = a$

  Hence (Z, ·) is a monoid with identity element 1

- Finally $\forall$ a, b, c $\in$ Z
  1. $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
  2. $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$

  i.e., multiplication is distributive over addition

Hence we can conclude that Z (+, *) is a ring

Modular Arithmetic

Modular arithmetic for integer n: $Z_n = \{0, 1, . . ., n\text{-}1\}$ forms a commutative ring for addition with a multiplicative identity

Lemma 1

For a, b, c $\in$ Z
If (a + b) ≡ (a + c) mod n then b ≡ c mod n

Lemma 2

For a, b, c $\in$ Z
If (a*b) ≡ (a*c) mod n then b ≡ c mod n **only if** a is relatively prime to n i.e. gcd(a, n) = 1.

Solving Modular Linear Equations

Definition: A congruence of the form ax ≡ b (mod m) is called a linear congruence.

Solving:

https://studyc.info/

- To solve this congruence, objective is to find the x that satisfy the given equation.
- An inverse of a, modulo m is any integer a' such that, $a'a \equiv 1 \pmod{m}$.
- If we can find such an a', then we can solve $ax \equiv b$ by multiplying throughout by it, giving $a'ax \equiv a'b$,
- Thus, $1 \cdot x \equiv a'b, \Rightarrow x \equiv a'b \pmod{m}$.

Theorem: If $\gcd(a, m) = 1$ and $m > 1$, then a has a unique inverse a' (modulo m).

Proof:

- Since $\gcd(a, m) = 1$,
  hence $\exists$ s, t such that, $sa + tm = 1$
- So, $sa + tm \equiv 1 \pmod{m}$.
- Since $tm \equiv 0 \pmod{m}$, $sa \equiv 1 \pmod{m}$.
- Thus s is an inverse of a (mod m).
- Hence this Theorem guarantees that if $ra \equiv sa \equiv 1$ then $r \equiv s$, thus this inverse is unique mod m.

Chinese Remainder Theorem

Theorem:

Let $m_1,\ldots,m_k > 0$ be relatively prime. Then the system of equations:

$x \equiv a_1 \pmod{m_1}$
$x \equiv a_2 \pmod{m_2}$
.
.
.
$x \equiv a_k \pmod{m_k}$

has a unique solution modulo $m = m_1 \cdot \ldots \cdot m_k$.

Proof:

- We are given that, $m = m_1 \cdot \ldots \cdot m_k$.
- Let $M_i = m/m_i$. for $i = 1, \ldots, k$
- Since $\gcd(m_i, M_i) = 1$, hence by above Theorem,
  $\exists\ y_i = M_i'$ such that $y_i M_i \equiv 1 \pmod{m_i}$ for $i = 1, \ldots, k$
- Let $x = a_1y_1M_1 + a_2y_2M_2 + \ldots + a_ky_kM_k = \sum a_iy_iM_i$
- Now m_1 does don't divide M_1
- But $m_2|M_1, m_3|M_1, \ldots, m_k|M_1$
- Similarly m_2 does don't divide M_2
- But $m_1|M_2, m_3|M_2, m_4|M_2, \ldots, m_k|M_2$ and so on
- Hence m_i does don't divide M_i, $\forall\ i \in \{1, 2, \ldots, k\}$
- But $m_i|M_j$, $\forall\ i \neq j$, $i, j \in \{1, 2, \ldots, k\}$
- Therefore, $M_j \equiv 0 \pmod{m_i}$ $j \neq i$,
- Now we show that x is simultaneous solution
- $x \equiv a_1 \pmod{m_1}$
- Since $x = a_1y_1M_1 + a_2y_2M_2 + \ldots + a_ky_kM_k$

https://studyc.info/

- Hence $x \equiv a_1y_1M_1 \equiv 1.a_1 = a_1 \pmod{m_1}$.
- $x \equiv a_2y_2M_2 \equiv 1.a_2 = a_2 \pmod{m_2}$.
- . . .
- $x \equiv a_ky_kM_k \equiv 1.a_k = a_k \pmod{m_k}$.
- Thus, x is the solution

Application: Example 3

Solve the given system of linear modular equations using Chinese Remainder Theorem.

$x \equiv 2 \pmod 3$, $a_1 = 2$
$x \equiv 3 \pmod 5$, $a_2 = 3$
$x \equiv 2 \pmod 7$, $a_3 = 2$

Solution

- As $m_1 = 3$, $m_2 = 5$, $m_3 = 7$, hence $m = 3.5.7 = 105$
- Now $M_1 = m/m_1 = 105/3 = 35$, $M_2 = m/m_2 = 105/5 = 21$ and $M_3 = m/m_3 = 105/7 = 15$
- Inverse of M_1 (modulo 3) = $y_1 = 2$
- Inverse of M_2 (modulo 5) = $y_2 = 1$
- Inverse of M_3 (modulo 7) = $y_3 = 1$
- Now, x, solution to this systems is
  $x \equiv a_1y_1M_1 + a_2y_2M_2 + a_3y_3M_3$
  $= 2.35.2 + 3.21.1 + 2.15.1 = 233 \pmod{105}$
  $\equiv 23 \pmod{105}$
- Thus 23 is the smallest positive integer that is a simultaneous solution.

Verification

$23 \equiv 2 \pmod 3$
$23 \equiv 3 \pmod 5$
$23 \equiv 2 \pmod 7$

Unique Representation of a Number by CRT

Let $m_1,\ldots,m_k$ are pair-wise relatively prime integers, let $m = m_1\cdot\ldots\cdot m_k$. Then by CRT it can be proved that any integer a, $0 \le a \le m$ can be uniquely represented by n-tuple consisting of its remainders upon division by m_i $(i = 1, 2, \ldots, k)$. That is we can uniquely represent a by

$(a \bmod m_1, a \bmod m_2, \ldots, a \bmod m_k) = (a_1, a_2, \ldots, a_k)$

Example 4

Pairs to represent non-negative integers < 12, first component is result of division by 3, second by 4

0 = (0, 0); 1 = (1, 1); 2 = (2, 2); 3 = (0, 3);
4 = (1, 0); 5 = (2, 1); 6 = (0, 2); 7 = (1, 3);
8 = (2, 0); 9 = (0, 1); 10 = (1, 2); 11 = (2, 3)

Example 5

https://studyc.info/

Compute (1, 2) if $m_1 = 3$ and $m_2 = 4$

Solution

$x \equiv 1 \pmod 3$
$x \equiv 2 \pmod 4$

- $m_1 = 3$, $m_2 = 4$, hence $m = 12$
- Now $M_1 = m/m_1 = 4$, $M_2 = m/m_2 = 3$
- Inverse of M_1 (modulo 3) = $y_1 = 1$
- Inverse of M_2 (modulo 4) = $y_2 = 3$

Now $x \equiv a_1y_1M_1 + a_2y_2M_2 = 1.1.4 + 2.3.3 = 22 \bmod 12 = 10$

Example 6: Chinese Remainder Theorem

Let $m_1 = 99$, $m_2 = 98$, $m_3 = 97$ and $m_4 = 95$
Now any integer < 99.98.97.95 = 89,403,930 can be uniquely represented by its remainders when divided by 99, 98, 97 and 95 respectively.
If a = 123,684, b = 413,456 then compute a + b.

Solution

Now 123,684 **mod** 99 = 33; 123,684 **mod** 98 = 8
123,684 **mod** 97 = 9; 123,684 **mod** 95 = 89
Hence a = 123,684 = (33, 8, 9, 89)

Similarly

413,456 **mod** 99 = 32; 413,456 **mod** 98 = 92
413,456 **mod** 97 = 42; 413,456 **mod** 95 = 16
Hence b = 413,456 = (32, 92, 42, 16)
Now a + b = 123,684 + 413,456
= (33, 8, 9, 89) + (32, 92, 42, 16)
= (65 **mod** 99, 100 **mod** 98, 51 **mod** 97, 105 **mod** 99)
Now we want to find a number x satisfying following
$x \equiv 65 \pmod{99}$
$x \equiv 2 \pmod{98}$
$x \equiv 51 \pmod{97}$
$x \equiv 10 \pmod{95}$
This can be solved using CRT,

Answer = 537,140

The RSA Public Key Cryptosystem

- RSA involves a public key and a private key.
- The public key can be known to everyone and is used for encrypting messages.
- Messages encrypted with the public key can only be decrypted using the private key.
- The keys for the RSA algorithm are generated by the following way:

https://studyc.info/

1. Choose two distinct large random prime numbers p and q such that $p \neq q$
2. Compute n by the equation $n = pq$, n is used as the modulus for both the public and private keys
3. Compute the totient function $\varphi(n)$
4. Choose an integer e such that $1 < e < \varphi(n)$ and e and $\varphi(n)$ share no factors other than 1 (co-prime), e is released as the public key exponent
5. Compute d to satisfy the congruence relation; $de \equiv 1 \mod \varphi(n)$ i.e.
   $de = 1 + k\varphi(n)$ for some integer k
   d is kept as the private key exponent
6. Publish the pair P =(e, n) as his RSA public Key
7. Keep secret pair S =(d, n) as his RSA secret Key

Property: Totient Function

Prove that $\varphi(p.q) = (p-1).(q-1)$, where p and q are prime numbers

Proof

If n = p, a prime number, then $\phi(p) = (p-1)$; e.g., ($\varphi(7) = 6$ because 7 is prime)
If n = p * q where p and q are both prime then

$$\phi(n) = \phi(p*q)$$

As above $\phi(p) = p - 1$
Similarly $\phi(q) = q - 1$
For $\phi(n) = \phi(p*q)$, the residues will be

$S_1 = \{0, 1, 2, \ldots, (pq-1)\}$

Out of S_1, residues that are not relatively prime to n:

$S_2 = \{p, 2p, \ldots (q-1)p\}$, $S_3 = \{q, 2q, \ldots\ldots (p-1)q\}$, $S_4 = \{0\}$

The number of elements of $S_1 = pq$
The number of elements of $S_2 = q-1$
The number of elements of $S_3 = p-1$
The number of elements of $S_4 = 1$
Hence number of relatively prime elements in S1 is

$$\phi(n) = pq - [(q-1)+(p-1)+1]$$
$$= pq - q + 1 - p + 1 - 1$$
$$= pq - q - p + 1 = (p-1)(q-1) = \phi(p) * \phi(q)$$

https://studyc.info/

Lecture 41 RSA Cryptosystem and String Matching

Fermat Theorem

Statement: If p is prime, a is positive integer not divisible by p,

$a^{p-1} = 1 \text{ mod } p$ OR $a^p = a \text{ mod } p$

Proof

Consider the set, $Z_p = \{0,1,\ldots, p-1\}$
Multiplying each element of Z_p by "a mod p", the result is a set, A, of all the elements of Z_p with a different sequence, where A = Zp

$A = \{0, a \text{ mod } p, 2a \text{ mod } p\ldots\ldots(p-1)a \text{ mod } p\}$

$\{0, a \text{ mod } p, 2a \text{ mod } p\ldots\ldots(p-1)a \text{ mod } p\} = \{0,1,\ldots, p-1\}$ Since A = Zp
If all the elements are multiplied together, except 0, on both sides we should
$\{a \text{ mod } p * 2a \text{ mod } p\ldots *(p-1) a \text{ mod } p\} \text{ mod } p = 1.2. \ldots(p-1) \text{ mod } p$ OR

$a^{p-1} (p-1)! \text{ mod } p = (p-1)! \text{ mod } p$

Since (p-1)! is relatively prime to p. So It can be cancelled from both sides
$a^{p-1} \text{ mod } p \equiv 1$ OR
$a^{p-1} \equiv 1 \text{ mod } p$ OR
$a^p \equiv a \text{ mod } p$

Euler's Theorem: Generalization of Fermat's

Statement: If a and n are relatively prime then

$a^{\varphi(n)+1} = a \text{ mod } n$ OR $a^{\varphi(n)} = 1 \text{ mod } n$

Proof
If n = prime, then $\phi(n) = n - 1$
By Fermat's Theorem $a^{n-1} = a^{\phi(n)} = 1 \text{ mod } n$
If n is a positive integer, then $\phi(n)$ = number of positive integers less than n, relatively prime to n.
Consider such positive integers as follows:

$S_1 = \{x_1, x_2, \ldots, x_{\phi(n)}\}$

Now multiply each element with a mod n

$S_2 = \{a\, x_1 \text{ mod } n, a\, x_2 \text{ mod } n, \ldots, a\, x_{\phi(n)} \text{ mod } n\}$

The set S_2 is a permutation of S_1 because:

1. a is relatively prime to n.
2. x_i is relatively prime to n.
3. Therefore ax_i is also relatively prime to n.

Hence each ax_i mod n has value less than n
Hence every element of S_2 is relatively prime to n and less than n.
The number of elements of S_2 equal to that of S_1
Moreover S2 contains no duplicates. It is because if $ax_i \text{ mod } n = ax_j \text{ mod } n$, then $x_i = x_j$
But S1 has no duplicates

https://studyc.info/

Euler's Theorem

On multiplying the terms of S1 and S2

$$\prod_{i=1}^{\phi(n)}(ax_i \bmod n) = \prod_{i=1}^{\phi(n)} x_i \qquad \text{OR}$$
$$\prod_{i=1}^{\phi(n)}(ax_i) = (\prod_{i=1}^{\phi(n)} x_i) \bmod n \qquad \text{OR}$$
$$a^{\phi(n)} = 1 \bmod n \ \text{ OR } \quad a^{\phi(n)+1} = a \bmod n, \qquad \text{PROVED}$$

Corollary:

Given primes p and q. Let m and n are integers such that n = p*q and 0 < m < n then

$$m^{\phi(n)+1} = m \bmod n \text{ OR } m^{\phi(n)} = 1 \bmod n$$

RSA Cryptosystem

Encryption:

Any number m, (m < n), can be encrypted.

ciphertext $c = m^e \bmod n$

Decryption:

$c^d \bmod n$ gives us back m.

Proof

To prove that $c^d \bmod n$ is equal to m:

$c^d \bmod n = (m^e)^d \bmod n$

$= m^{de} \bmod n$

Since $de = 1 \bmod \Phi(n) \Rightarrow de = k\Phi(n) + 1$

$c^d = m^{de} = m^{k\Phi(n)+1}$

By the above corollary to Euler's theorem,

$c^d = m^{de} = m^{k\Phi(n)+1} = m \bmod n = m$, *since $m < n$*

Example 7: RSA Cryptosystem

Encrypt message STOP using RSA cryptosystem with p = 43, q = 59 and e = 13, n = pq = 2537,

Solution

gcd(e, (p-1)(q-1)) = 1, encryption can be done

Translate STOP in numerical values, blocks of 4

1819 1415

https://studyc.info/

Encrypt

$C = M^e$ **mod** 2537 = M^{13} **mod** 2537

After computing using fast modular multiplication

1819^{13} **mod** 2537 = 2081;1415^{13} **mod** 2537 = 2181

The encrypted message is: 2081 2182

Example 8: RSA Cryptosystem

Decrypt 0981 0461 if encrypted using RSA

Public key = (e, n) = (13, 43.59 = 2537)

Solution

p = 43, p-1 = 42, q = 59, q-1 = 58, e = 13

d = e^{-1} mod (p-1).(q-1) = 13^{-1} mod 42.58 = 937

Decrypt

$M = C^{937}$ **mod** 2537 = C^{937} **mod** 2537

After computing using fast modular multiplication

0981^{937} **mod** 2537 = 0704;0461^{937} **mod** 2537 = 1115

The decrypted message is: 0704 1115

Translating back to English: HELP

String Matching

String Matching Problem

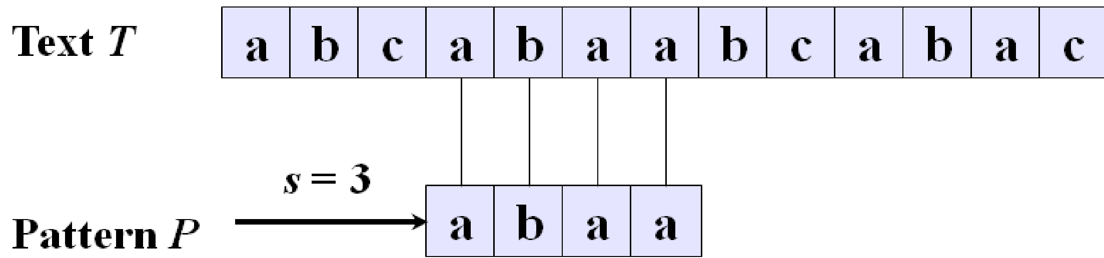
- We assume that the text is an array $T[1 .. n]$ of length n and that the pattern is an array $P[1 .. m]$ of length $m \leq n$.
- We further assume that the elements of P and T are characters drawn from a finite alphabet Σ.
  - For example, we may have Σ = {0, 1} or Σ = {a, b, . . . , z}.
- The character arrays P and T are often called **strings** of characters.
- We say that pattern P **occurs with shift s** in text T (or, equivalently, that pattern P **occurs beginning at position s + 1** in text T) if $0 \leq s \leq n - m$ and $T[s + 1 .. s + m] = P[1 .. m]$ i.e. $T[s + j] = P[j]$, for $1 \leq j \leq m$).
  - If P occurs with shift s in T, we call s a **valid shift**;
  - otherwise, we call s an **invalid shift**.

String Matching Problem

The string-matching problem is "finding all valid shifts with which a given pattern P occurs in a given text T".

Example: String Matching Problem

https://studyc.info/



Definitions and Notations

Notation	Terminology
Σ*	The set of all finite-length strings formed using characters from the alphabet Σ.
ε	The zero-length empty string, also belongs to Σ*.
|x|	The length of a string *x*.
xy	The concatenation of two strings x and y has length |x| + |y| and consists of the characters from x followed by the characters from y.
w ⊏ x	A string w is a prefix of a string x, if x = wy for some string y ∈ Σ*. If w ⊏ x, then |w| ≤ |x|.
w ⊐ x	A string w is a suffix of a string x, if x = yw for some y ∈ Σ*. If w ⊐ x that |w| ≤ |x|.

1. Naïve Approach

The idea is based on Brute Force Approach.
The naive algorithm finds all valid shifts using a loop that checks the condition $P[1 .. m] = T[s + 1 .. s + m]$ for each of the $n - m + 1$ possible values of s.
It can be interpreted graphically as sliding a “template“ containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text.

Naive String Matching Algorithm

NAIVE-STRING-MATCHER(*T*, *P*)
1 $n \leftarrow length[T]$
2 $m \leftarrow length[P]$
3 **for** $s \leftarrow 0$ **to** $n - m$
4 **do if** $P[1 .. m] = T[s + 1 .. s + m]$
5 **then** print "Pattern occurs with shift" s

Worst case Running Time

- Outer loop: $n – m + 1$

https://studyc.info/

- Inner loop: m
- Total $\Theta((n - m + 1)m)$

Best-case: n-m

Note:

- Not an optimal procedure for String Matching problem.
- It has high running time for worst case.
- The naive string-matcher is inefficient because information gained about the text for one value of s is entirely ignored in considering other values of s.

2. The Rabin-Karp Algorithm

Let us assume that $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit.
A string of k consecutive characters is viewed as representing a length-k decimal number.
Given a pattern $P[1 .. m]$, let p denote its corresponding decimal value and a text $T[1 .. n]$, we let t_s denotes the decimal value of the length-m substring $T[s + 1 .. s + m]$, for $s = 0, 1, ..., n - m$.
Now, $t_s = p$ if and only if $T[s + 1 .. s + m] = P[1 .. m]$; thus, s is a valid shift if and only if $t_s = p$.
We can compute p in time $\Theta(m)$ using Horner's rule

$$p = P[m] + 10 (P[m - 1] + 10(P[m - 2] + \cdots + 10(P[2] + 10P[1]))).$$

Example: Horner's rule

"345" = 5 + 10(4 + 10(3)) = 5 + 10(4 + 30) = 5 + 340 = 345
The value t_0 can be similarly computed from $T[1 .. m]$ in time $\Theta(m)$.
To compute the remaining values $t_1, t_2, \ldots, t_{n-m}$ in time $\Theta(n - m)$, it suffices to observe that t_{s+1} can be computed from t_s in constant time.
Subtracting $10^{m-1} T[s + 1]$ removes the high-order digit from t_s, multiplying the result by 10 shifts the number left one position, and adding $T[s + m + 1]$ brings in the appropriate low-order digit.

$$t_{s+1} = (10(t_s - T[s + 1]\, 10^{m-1}) + T[s + m + 1])$$

The only difficulty with this procedure is that p and t_s may be too large to work with conveniently.
Fortunately, there is a simple cure for this problem compute p and the t_s's modulo a suitable modulus q.

https://studyc.info/

Lecture 42 String Matching

String Matching Problem

Given a text $T[1 .. n]$ of length n, a pattern $P[1 .. m]$ of length $m \leq n$, both as arrays.
Further assume that elements of P and T are characters drawn from a finite set of alphabets Σ.
Now for $0 \leq s \leq n - m$ if

$$T[s + j] = P[j], \forall j \in \{1, 2, \ldots, m\}$$

then p occurs in T with shift s, and we call s as a **valid shift**; otherwise, s an **invalid shift**.
Our objective is "finding all valid shifts with which a given pattern P occurs in a text T".

Naive String Matching Algorithm

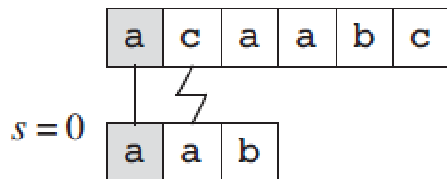
NAIVE-STRING-MATCHER(T, P)
1 $n \leftarrow length[T]$
2 $m \leftarrow length[P]$
3 **for** $s \leftarrow 0$ **to** $n - m$
4 **do if** $P[1 .. m] = T[s + 1 .. s + m]$
5 **then** print "Pattern occurs with shift" s

Worst case Running Time

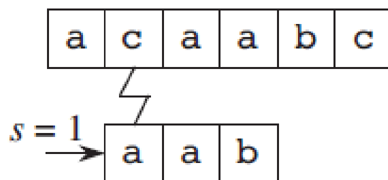
- Outer loop: $n - m + 1$
- Inner loop: m
- Total $\Theta((n - m + 1)m)$

Best-case: n-m

Example: Naive String Matching Algorithm

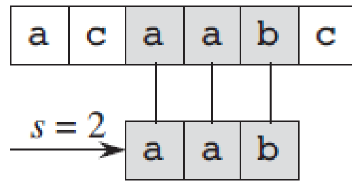


(a)

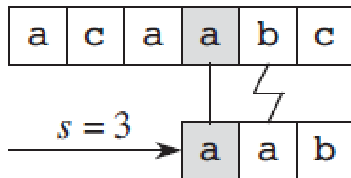


(b)

https://studyc.info/



(c)



(d)

The operation of the naive string matcher for the pattern P = aab and the text T = acaabc. We can imagine the pattern P as a template that we slide next to the text. (a) – (d). The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. The algorithm finds one occurrence of the pattern, at shift s = 2, shown in part (c).

The Rabin-Karp Algorithm

Special Case
Given a text $T[1 .. n]$ of length n, *a* pattern $P[1 .. m]$ of length $m \leq n$, both as arrays.
Assume that elements of P and T are characters drawn from a finite set of alphabets Σ.
Where $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit.
Now our objective is "finding all valid shifts with which a given pattern *P* occurs in a text *T*".

Notations: The Rabin-Karp Algorithm

Let us suppose that

- p denotes decimal value of given a pattern $P[1 .. m]$
- t_s = decimal value of length-m substring $T[s + 1 .. s + m]$, of given text $T[1 .. n]$, for $s = 0, 1, ..., n - m$.
- It is very obvious that, $t_s = p$ if and only if
  $T[s + 1 .. s + m] = P[1 .. m]$;
  thus, s is a valid shift if and only if $t_s = p$.
- Now the question is how to compute p and t_s efficiently
- Answer is Horner's rule

Horner's Rule

Example: Horner's rule

https://studyc.info/

[3, 4, 5] = 5 + 10(4 + 10(3)) = 5 + 10(4 + 30) = 5 340 = 345

$p = P[3] + 10\ (P[3 - 1] + 10(P[1]))$.

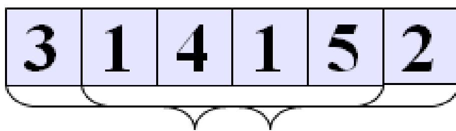
Formula

- We can compute p in time $\Theta(m)$ using this rule as

$p = P[m] + 10\ (P[m-1] + 10(P[m-2] + \ldots + 10(P[2] + 10P[1])\ ))$

- Similarly t_0 can be computed from $T\ [1\ ..\ m]$ in time $\Theta(m)$.
- To compute $t_1, t_2, \ldots, t_{n-m}$ in time $\Theta(n - m)$, it suffices to observe that t_{s+1} can be computed from t_s in constant time.

Computing t_{s+1} from t_s in constant time



Text = [3, 1, 4, 1, 5, 2]; t_0 = 31415
m = 5; Shift = 0
Old higher-order digit = 3
New low-order digit = 2
$t_1 = 10.(31415 - 10^{4}.T(1)) + T(5+1)$
$= 10.(31415 - 10^{4}.3) + 2$
$= 10(1415) + 2 = 14152$
$t_{s+1} = 10(t_s - T[s + 1]\ 10^{m-1}\ ) + T[s + m + 1])$
$t_1 = 10(t_0 - T[1]\ 10^{4}) + T[0 + 5 + 1])$
Now $t_1, t_2, \ldots, t_{n-m}$ *can be computed* in $\Theta(n - m)$

Procedure: Computing t_{s+1} from t_s

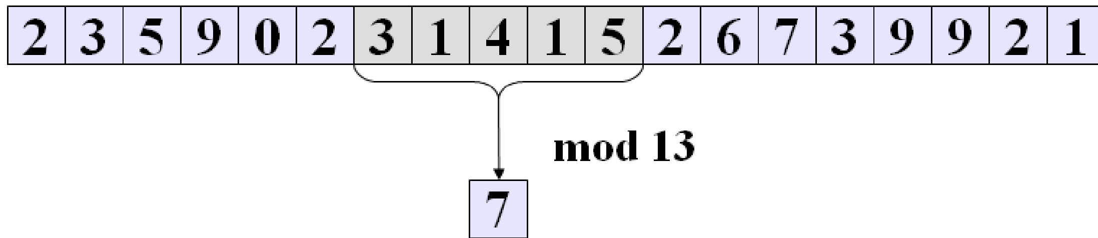
1. Subtract $T[s + 1]10^{m-1}$ from t_s, removes high-order digit
2. Multiply result by 10, shifts the number left one position
3. Add $T\ [s + m + 1]$, it brings appropriate low-order digit.
   $t_{s+1} = (10(t_s - T[s + 1]\ 10^{m-1}\ ) + T[s + m + 1])$

Another issue and its treatment

- The only difficulty with the above procedure is that p and t_s may be too large to work with conveniently.
- Fortunately, there is a simple cure for this problem, compute p and the t_s modulo a suitable modulus q.

Computing t_{s+1} from t_s Modulo q = 13

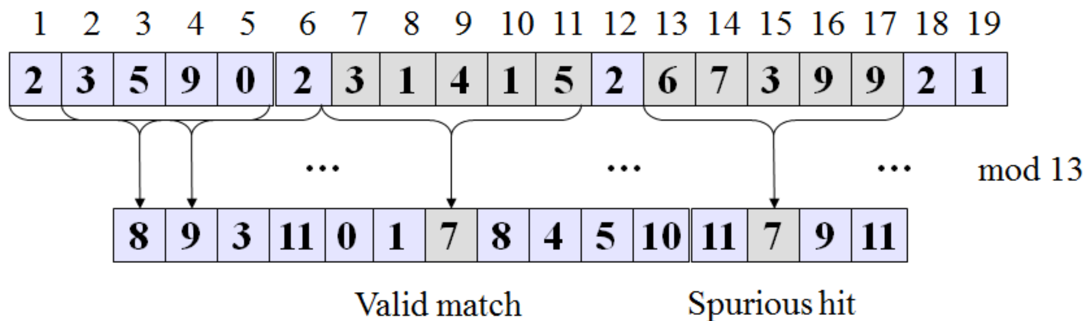
https://studyc.info/



A window of length 5 is shaded.
The numerical value of window = 31415
31415 mod 13 = 7

Spurious Hits and their Elimination

m = 5.
p = 31415,
Now, 31415 ≡ 7 (mod 13)
Now, 67399 ≡ 7 (mod 13)
Window beginning at position 7 = valid match; s = 6
Window beginning at position 13 = spurious hit; s = 12
After comparing decimal values, text comparison is needed.



The Rabin-Karp Algorithm

Generalization

- Given a text T [1 .. n] of length n, a pattern P[1 .. m] of length $m \leq n$, both as arrays.
- Assume that elements of P and T are characters drawn from a finite set of alphabets Σ = {0, 1, 2, . . . , d-1}.
- Now our objective is "finding all valid shifts with which a given pattern P occurs in a text T".

Note:

$t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$
where $h = d^{m-1} \pmod q$ is the value of the digit "1" in the high-order position of an m-digit text window.

Sequence of Steps Designing Algorithm

1. Compute the lengths of pattern P and text T
2. Compute p and t_s under modulo q using Horner's Rule
3. For any shift s for which $t_s \equiv p \pmod{q}$, must be tested further to see if s is really valid shift or a **spurious hit**.
4. This testing can be done by checking the condition: $P[1 .. m] = T[s + 1 .. s + m]$. If these strings are equal s is a valid shift otherwise spurious hit.
5. If for shift s, $t_s \equiv p \pmod{q}$ is false, compute t_{s+1} and replace it with t_s and repeat the step 3.

Note: As $t_s \equiv p \pmod{q}$ does not imply that $t_s = p$, hence text comparison is required to find valid shift

The Rabin-Karp Algorithm

```
RABIN-KARP-MATCHER(T, P, d, q)
1   n ← length[T]
2   m ← length[P]
3   h ← d^(m-1) mod q
4   p ← 0
5   t0 ← 0
6   for i ← 1 to m                    (Preprocessing)
7       do p ← (dp + P[i]) mod q
8          t0 ← (dt0 + T[i]) mod q
9   for s ← 0 to n - m                (Matching)
10      do if p = ts
11          then if P[1 .. m] = T [s + 1 .. s + m]
12                  then print "Pattern occurs with shift" s
13         if s < n - m
14          then ts+1 ← (d(ts - T[s + 1]h) + T[s + m + 1]) mod q
```

Analysis: The Rabin-Karp Algorithm

- Worst case Running Time
  - Preprocessing time: $\Theta(m)$
  - Matching time is $\Theta((n - m + 1)m)$
- If $P = a^m$, $T = a^n$, verifications take time $\Theta((n - m + 1)m)$, since each of the $n - m + 1$ possible shifts is valid.
- In applications with few valid shifts, matching time of the algorithm is only $O((n - m + 1) + cm) = O(n + m)$, plus the time required to process spurious hits.

String Matching with Finite Automata

- A **finite automaton** M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where
  - Q is a finite set of **states**,
  - $q_0 \in Q$ is the **start state**,
  - $A \subseteq Q$ is a distinguished set of **accepting states**,

https://studyc.info/

   - Σ is a finite **input alphabet**,
   - δ is a function from $Q \times \Sigma$ into Q, called the **transition function** of M.
- String-matching automata are very efficient because it examines each character *exactly once*, taking constant time.
- The matching time used-after preprocessing the pattern to build the automaton-is therefore $\Theta(n)$.

Some Results

1. Empty string is both a suffix and a prefix of every string.
2. For any strings x and y and any character a, we have $x \sqsupset y$ if and only if $xa \sqsupset ya$.
3. Also it can be proved that $\sqsubset$ and $\sqsupset$ are transitive relations.

Proof: Property 3

- Suppose that $x \sqsubset y$ and $y \sqsubset z$, we have to prove that x □ z.
- $x \sqsubset y \Rightarrow \exists\, w_1 \in \Sigma^*$ such that $y = xw_1$ (A)
- $y \sqsubset z \Rightarrow \exists\, w_2 \in \Sigma^*$ such that $z = yw_2$ (B)
- From (A) and (B)
  $z = yw_2 = xw_1w_2$
- And hence $x \sqsubset z$, this is because $w_1w_2 \in \Sigma^*$

Example: Transition Table and Finite Automata

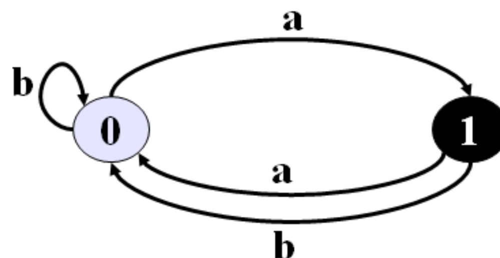
$Q = \{0, 1\}$, $\sum = \{a, b\}$ and transition function is shown below

A simple two-state finite automaton which accepts those strings that end in an odd number of a's.

input

state	a	b
0	1	0
1	0	0

A tabular representation of transition function δ



State set $Q = \{0, 1\}$
Start state $q_0 = 0$
Input alphabet $\Sigma = \{a, b\}$

Final State Function φ

A finite automaton M induces a function φ, called the **final-state function**, from Σ^* to Q such that $\varphi(w)$ is the state of M *that* ends up in after scanning the string w.
Thus, M accepts a string w if and only if $\varphi(w) \in A$.

https://studyc.info/

The function φ is defined by the recursive relation

$\varphi(\varepsilon) = q_0$,

$\varphi(wa) = \delta(\varphi(w), a)$ for $w \in \Sigma^*$, $a \in \Sigma$.

There is a string-matching automaton for every pattern P; constructed in a preprocessing step.

Suffix Function σ

An auxiliary function σ, called the **suffix function** is defined corresponding to given pattern P.

Function σ is a mapping from Σ^* to $\{0, 1, \ldots, m\}$ such that $\sigma(x)$ is length of the longest prefix of P that is a suffix of x *i.e.*

$\sigma(x) = \max \{k : P_k \sqsupset x\}$.

The suffix function σ is well defined since the empty string $P_0 = \varepsilon$ is a suffix of every string.

For a pattern P of length m, we have $\sigma(x) = m$ if and only if $P \sqsupset x$.

It follows from the definition of the suffix function that if $x \sqsupset y$, then $\sigma(x) \leq \sigma(y)$.

String Matching Automata

The string-matching automaton that corresponds to a given pattern $P[1 .. m]$ is defined as

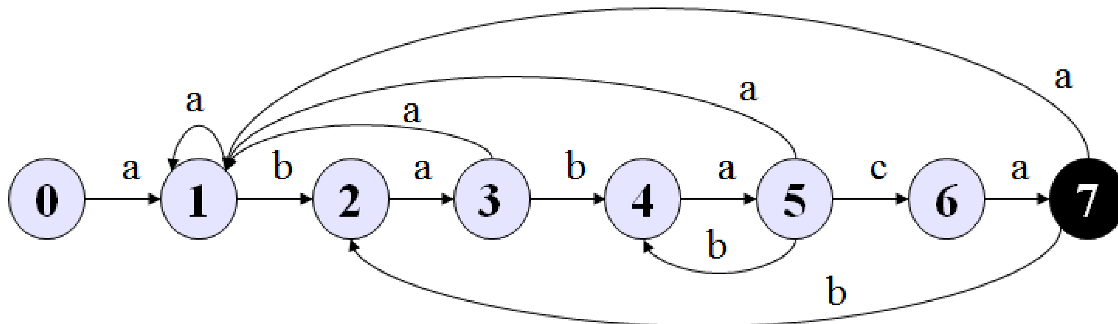
- The state set Q is $\{0, 1, \ldots, m\}$. The start state $q0$ is state 0, and state m is the only accepting state.
- The transition function δ is defined by the following equation, for any state q and character a:

$\delta(q, a) = \sigma(P_q a)$

The machine maintains as an invariant of its operation

$\varphi(T_i) = \sigma(T_i)$

String Matching Automata for given Pattern



state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

- Pattern string P = ababaca.
- Edge towards right shows matching
- Edge towards is for failure
- No edge for some for state and some alphabet means that edge hits initial state

Finite Automata for Pattern P = ababaca

Text T = abababacaba.

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	**7**	2	3

Algorithm

FINITE-AUTOMATON-MATCHER(T, δ, m)
1 $n \leftarrow length[T]$
2 $q \leftarrow 0$
3 **for** $i \leftarrow 1$ **to** n
4 **do** $q \leftarrow \delta(q, T[i])$
5 **if** $q = m$
6 **then** print "Pattern occurs with shift" $i - m$

Matching time on a text string of length n is $\Theta(n)$.

Memory Usage: $O(m|\Sigma|)$,

https://studyc.info/

Preprocessing Time: Best case: $O(m|\Sigma|)$.

Algorithm

COMPUTE-TRANSITION-FUNCTION(P, Σ)

```
1  m ← length[P]
2  for q ← 0 to m
3      do for each character a ∈ Σ
4          do k ← min(m + 1, q + 2)
5              repeat k ← k - 1
6                  until P_k ⊐ P_q a
7              δ(q, a) ← k
8  return δ
```

Running Time = $O(m^3 |\Sigma|)$

Summary

Algorithm	Preprocessing Time	Matching Time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite Automaton	$O(m|\Sigma|)$	$\Theta(n)$

https://studyc.info/

Lecture 43 Polynomials and Fast Fourier Transform

Field: A **Field** is a set F with two binary operations $+ : F \times F \rightarrow F$ and $* : F \times F \rightarrow F$ such that

1. $(F, +)$ is an abelian group with identity element 0
2. $(F \backslash \{0\}, *)$ is an abelian group with identity element 1
3. Multiplication distributes over addition
   - $a*(b + c) = (a*b) + (a*c)$
   - $(a + b)*c = (a*c) + (b*c)$

Polynomial

A *polynomial* in the variable x over an algebraic field F is a representation of a function $A(x)$ as a formal sum

$$A(x) = a_0 + a_1x^1 + a_2x^2 + \ldots + a_nx^n$$

Coefficients

Values $a_0, a_1, \ldots, a_n$ are *coefficients* of polynomial, and drawn from a field F, typically set of complex numbers.

Degree

A polynomial $A(x)$ is said to have *degree n* if its highest coefficient a_n is nonzero

Degree Bound

Any integer strictly greater than the degree of a polynomial is a degree-bound of that polynomial.

Addition of two Polynomials: Brute Force

Addition of two polynomials of degree n takes $\Theta(n)$ time,

Example 1

$$A(x) = a_0 + a_1x^1 + a_2x^2 + \ldots + a_nx^n$$

$$B(x) = b_0 + b_1x^1 + b_2x^2 + \ldots + b_nx^n$$

$$C(x) = (a_0 + b_0) + (a_1 + b_1)x^1 + \ldots + (a_n + b_n)x^n$$

https://studyc.info/

Multiplication of Two Polynomial: Brute Force

Multiplication of two polynomials of degree n takes $\Theta(n^2)$

Example 2

$A(x) = a_0 + a_1x^1 + a_2x^2 + \ldots + a_nx^n$

$B(x) = b_0 + b_1x^1 + b_2x^2 + \ldots + b_nx^n$

$a_0b_0 + a_1b_0x^1 + \ldots + (a_nb_0)x^n$

$a_0b_1x^1 + a_1b_1x^2 + \ldots + (a_nb_1)x^{n+1}$

. . .

$a_0b_n x^n + a_1b_nx^{n+1} + \ldots + (a_nb_n)x^{n+n}$

$C(x) = (a_0b_0) + (a_1b_0 + a_0b_1)x^1 + \ldots + (a_nb_n)x^{n+n}$

Polynomial Representation

1. The Coefficient Representation
2. Point Value Presentation

Note

- The method for multiplying polynomials equations as above take $\Theta(n^2)$ time when the polynomials are represented in coefficient form
- But $\Theta(n)$ time when represented in point value form
- We can multiply using coefficient representation in only $\Theta(n \log n)$ time converting between two forms
- This lecture make much use of complex numbers, the symbol i has same meaning, you know sqr(-1)

Coefficient Representation

A coefficient representation of a polynomial degree bound n is a vector of coefficients: a = $(a_0, a_1, \ldots, a_{n-1})$

Vectors as Column

- In this lecture, we will treat vector as column vector

Convenient Way

- The coefficients representation is convenient for certain operations on polynomials

Example:

Computing A(x) at x_0

https://studyc.info/

- Operation of evaluating polynomials A(x) at given point x_0 consists of computing value of $A(x_0)$.
- Evaluation takes time $\Theta(n)$ using Horner's rule

Evaluation and Addition using Coefficient Form

Operation 1: **Horner's Rule**

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots x_0(a_{n-2} + x_0(a_{n-1}))\ldots)$$

Operation 2: Addition of Two Polynomials

- Similarly adding two polynomials represented by the coefficient vectors:
  $a = (a_0, a_1, \ldots, a_{n-1})$ and
  $b = (b_0, b_1, \ldots, b_{n-1})$ takes $\Theta(n)$ times
- We just produce the coefficient vector:
  $c = (c_0, c_1, \ldots, c_{n-1})$ where

$$c_j = a_j + b_j, \quad \forall\ j = 1, \ldots, n - 1$$

Multiplication using Coefficient Representation

Operation 3: Multiplication of Two Polynomials

- Consider multiplication of A(x) and B(x), with degree bounds n, represented in coefficient form
- If we use the method described above polynomials multiplication takes time $O(n^2)$.
- Since each coefficient in vector a must be multiplied by each coefficients in the vector b.
- Operation of multiplying polynomials in coefficient form seems to be considerably more difficult than that of evaluating or adding two polynomials.
- The resulting coefficient vector c, also called the convolution of the input vectors a and b.

Point–value Representation

- A point value representation of a polynomial A(x) of degree bound n is a set of n point value pairs.
  $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$,
  all of the x_k are distinct and $y_k = A(x_k)$, for $k = 0, 1, \ldots, n-1$.
- Polynomial has various point value representations, since any set of n distinct points $x_0, x_1, \ldots, x_{n-1}$ can be used as a basis for the representation.

Conversion: From a Coefficient Form to Point Value Form

- Computing a point value representation for a polynomials given in coefficient form is in principle straight forward ,
- This is because select n distinct points $x_0, x_1, \ldots, x_{n-1}$ and then evaluate $A(x_k)$ for $k = 0, 1, \ldots, n-1$.

https://studyc.info/

- With Horner's rule, n-point evaluation takes q(n^2).
- This is because for $x = x_0$, evaluation cost is q(n). And since there are n number of points, hence there will be q(n^2) cost for evaluating all of the n number of points using Horner's rule.

Clever Choose of x_k

- We shall see that if we choose x_k cleverly, this computation can be accelerated to run in q(n log n)
- Inverse of evaluating coefficient form of polynomial from point value representation called interpolation.

Theorem

Uniqueness of an interpolating polynomial

For any set { $(x_0, y_0),(x_1, y_1),\ldots, (x_{n-1}, y_{n-1})$ } of n point–value pairs such that all x_k values distinct, there is a unique polynomial A(x) of degree bound n such that $y_k = A(x_k)$ for k = 0,1, . . ., n-1.

Proof

- Proof is based on existence of inverse of a matrix.
- Let us suppose that A(x) is required polynomial
  $A(x) = a_0 + a_1x^1 + a_2x^2 + \ldots + a_nx^n$
- Equation: $y_k = A(x_k)$ is equivalent to the matrix equation given in the next slide.

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ \vdots \\ y_{n-1} \end{pmatrix}$$

This matrix on the left side is called vander-monde matrix and is denoted $V(x_0, x_1, \ldots x_{n-1})$

- The determinant of this this matrix is $\prod_{0 \le j < k \le n-1} (x_k - x_j)$

If x_k are distinct then it is nonsingular. The coefficient a_j can be uniquely determined $a = V(x_0, x_1, \ldots x_{n-1})^{-1} y$

Solving the Equation in Proof of Theorem

- Using LU decomposition algorithms, we can solve these equation in O(n^3)
- A faster algorithm, in $\Theta(n^2)$, for *n*-point interpolation is based on *Lagrange's formula*:

$$\sum_{k=0}^{n-1} y_k \frac{\prod_{j \ne k} \left(x - x_j\right)}{\prod_{j \ne k} \left(x_k - x_j\right)}$$

https://studyc.info/

Addition using Point Value Form

- The point-value representation is quite convenient for many operations on polynomials.
- For addition:
  $C(x) = A(x) + B(x) \Rightarrow C(x_k) = A(x_k) + B(x_k)$
- More precisely, if point value representation for A
  - $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$,
- And for B: $\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$,
- Then a point-value representation for C is
  $\{(x_0, y_0+y'_0), (x_1, y_1+y'_1), \ldots, (x_{n-1}, y_{n-1}+y'_{n-1})\}$,
- Thus, the time to add two polynomials of degree-bound n in point-value form is $\Theta(n)$.

Multiplication using Point Value Form

- Similarly, point-value representation is convenient for multiplying polynomials as well.

$C(x) = A(x)\,B(x) \Rightarrow C(x_k) = A(x_k)B(x_k)$ for any x_k,

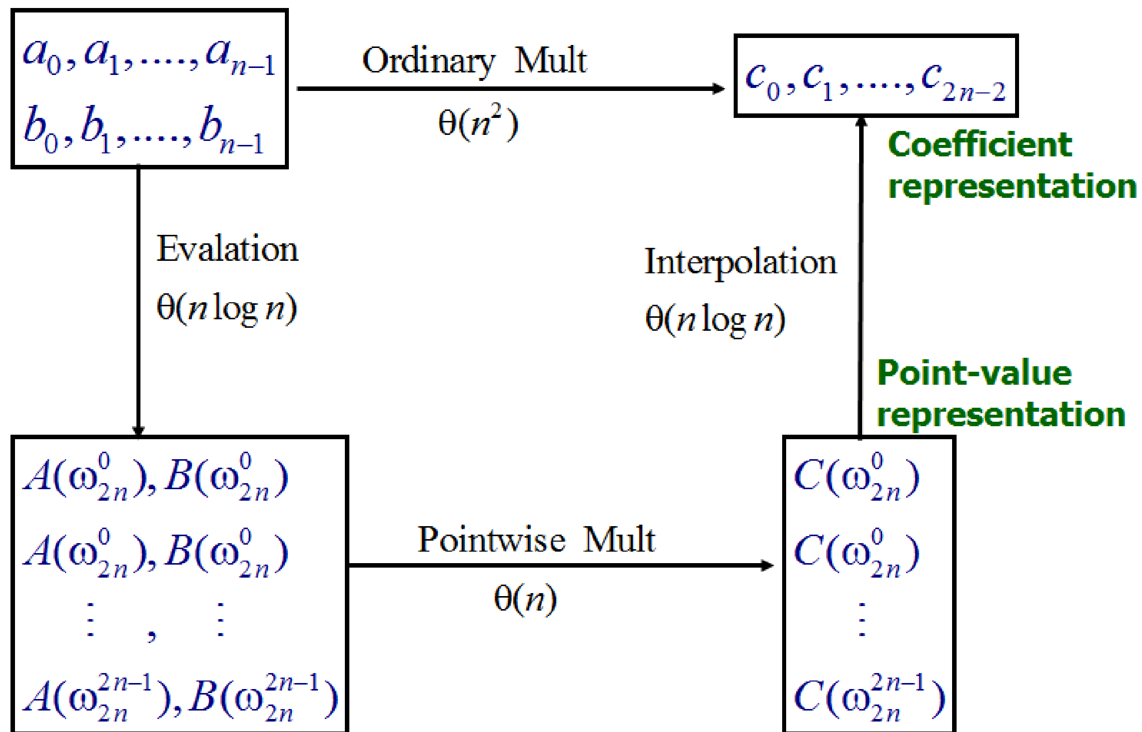
- We can multiply a point value representations for *A and B* to obtain a point-value representation for *C*.
- A standard point-value representation for *A* and *B* consists of *n* point-value pairs for each polynomial
- Multiplying these, we must extended point-value representations for *A* and *B* of $2n$ point-value each.
- Given an extended point-value representation for *A*,
  $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\}$,
- And extended point-value representation for *B*,
  $\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\}$,
- Then a point-value representation for *C* is
  $\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{n-1}, y_{n-1} y'_{n-1})\}$
- Finally, we consider how to evaluate a polynomial given in point-value form at a new point.
- Apparently no simpler approach than converting polynomial to coefficient form, and then evaluating it

Discrete Fourier Transform

- We can use any points as evaluation points, but by choosing evaluation points carefully, we can convert between representations in only $\Theta(n \lg n)$ time.
- If we take "complex roots of unity" evaluation points, we can produce a point-value representation taking Discrete Fourier Transform of coefficient vector.
- The inverse operation, interpolation, can be performed by taking "inverse DFT" of point-value pairs, yielding a coefficient vector.
- We will show how FFT performs the DFT and inverse DFT operations in $\Theta(n \lg n)$
- Multiplication procedure is shown in the next slide

Fast multiplication of polynomials in coefficient form

https://studyc.info/



Procedure: Multiplication of Polynomials in n lg n

We assume n is a power of 2; this requirement can always be met by adding zero coefficients.

1. *Double degree-bound:*

Create coefficient representations of $A(x)$ and $B(x)$ as degree bound $2n$ polynomials by adding n high-order zero coefficients to each.

2. *Evaluate:*

Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of FFT of order $2n$. These representations contain the values of the two polynomials at the $(2n)$th roots of unity.

3. *Point wise multiply:*

Compute point-value form for polynomial $C(x) = A(x)B(x)$ by multiplying these together point wise. This representation contains the value of $C(x)$ at each $(2n)$th root of unity.

4. *Interpolate:*

Create coefficient representation of $C(x)$ through a single application of an FFT on $2n$ point-value pairs to compute inverse DFT.

Steps (1) and (3) take time $\Theta(n)$, and steps (2) and (4) take time $\Theta(n \lg n)$.

https://studyc.info/

Complex Roots of Unity and Their Properties

- We claimed that if we use complex roots of unit we can evaluate and interpolate polynomials in Θ(nlgn) time.
- Here, we define complex roots of unity and study their properties.
- Define the DFT,and then show how the FFT computes the DFT and its inverse in just Θ(nlgn) time.

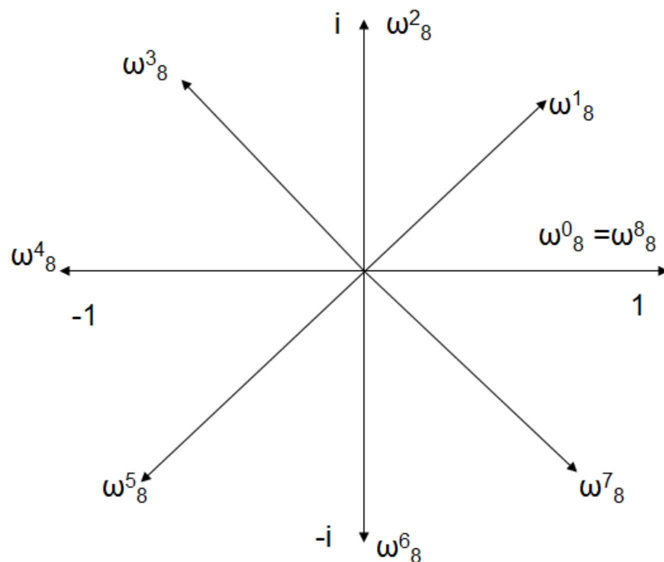
Complex root of unity

A complex nth root of unity is a complex number ω such that

$$\omega^n = 1$$

There are exactly n complex nth roots of unity: $e^{2\pi ik/n}$ for k=0,1,…,n-1

$$e^{iu} = \cos(u) + i\sin(u).$$



Values of $\omega^0_8, \omega^1_8, \ldots, \omega^7_8$ in complex plane are shown where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity. Complex roots of unity form a cyclic group. Complex roots of unity have interesting properties.

Properties: Complex Roots of Unity

Lemma 1 (cancellation lemma)

For any integers $n \geq 0, k \geq 0$,and $d > 0$,

$\omega^{dk}_{dn} = \omega^k_n$.

Proof : the lemma follows directly from $\omega_n = e^{2\pi i/n}$,since

$\omega^{dk}_{dn} = (e^{2\pi i/dn})^{dk}$

https://studyc.info/

$=(e^{2\pi i/n})^k$

$= \omega^k_n$

Corollary 1 (cancellation lemma)

For any even integer n > 0,

$\omega^{n/2}_n = -1$

Proof: We know that $\omega_n = e^{2\pi i/n}$

Now $\omega^{n/2}_n = \omega^{n/2}_{2.n/2} = \omega_2 = \omega_n = e^{2\pi i/2} = \omega_n = e^{\pi i} = -1$

Lemma 2 (Halving Lemma)

If $n > 0$ is even, then squares of n complex nth root of unity are the $n/2$ complex ($n/2$)th root of unity.

Proof:

- By the cancellation lemma, we have: $(\omega_n^k)^2 = \omega^k_{n/2}$
- For any nonnegative integer k, note that if we square all of complex nth root of unity, then each ($n/2$)th root of unity is obtained exactly twice, since

$(\omega_n^{k+n/2})^2 = \omega^{2k+n}_{n/2}$

$= \omega^{2k}_n \omega_n^n = \omega^{2k}_n$

Halving Lemma is Essential in Reducing Cost

- Thus ω^k_n *and* $\omega^{k+n/2}_{n/2}$ have the same square.
- This property can also be proved using corollary,

  $\omega_n^{n/2} = \omega_2 = -1$
- Since $\omega^{n/2}_n = -1$ *implies* $\omega^{k+n/2}_n = -\omega^k_n$ *and thus*

  $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$
- As we shall see, the halving lemma is essential to our divide-and-conquer approach of converting between coefficient and point-value representation of polynomials
- Since it guarantees that the recursive sub problems are only half as large.

Lemma 3 (Summation Lemma)

For any integer $n \geq 1$ and nonnegative integer k not divisible by n, $\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j = 0$

Proof:

https://studyc.info/

$$\sum_{j=0}^{n-1}\left(\omega_n^{\,k}\right)^j = \frac{\left(w_n^{\,k}\right)^n - 1}{w_n^{\,k} - 1}$$
$$= \frac{\left(w_n^{\,n}\right)^k - 1}{w_n^{\,k} - 1}$$
$$= \frac{(1)^k - 1}{w_n^{\,k} - 1}$$
$$= 0$$

Requiring that k not be divisible by n ensures that the denominator is not 0, since $\omega_n^k = 1$ only when k is divisible by n.

The DFT

Recall that we wish to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree-bound n at $\omega_n^0, \omega_n^1, \omega_n^2, \ldots\ldots \omega_n^{n-1}$ *(that is, at the n complex nth roots of unity)*
Without loss of generality, we assume that n is a power of 2, since a given degree-bound can always be raised we can always add new high-order zero coefficients as necessary.
We assume that A is given in coefficient form $a = (a_0, a_1, \ldots, a_{n-1})$.
Let us define the results y_k for $k = 0,1, \ldots, n-1$, *by*

$$y_k = A\left(\omega_n^{\,k}\right)$$
$$= \sum_{j=0}^{n-1} a_j \omega^k j_n$$

The vector y = $(y_0, y_1, \ldots, y_{n-1})$ is Discrete Fourier Transform (DFT) of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$.
We can also write y = $DFT_n(a)$

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots.. + a_{n-2} x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots.. + a_{n-1} x^{n/2-1}$$

And hence $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$ (A)

Thus evaluating A(x) at $\omega_n^0, \omega_n^1, \ldots.., \omega_n^{n-1}$ reduce to

1. evaluating $A^{[0]}(x)$ and $A^{[1]}(x)$ at
   $(\omega_n^0)^2, (\omega_n^1)^2, \ldots.., (\omega_n^{n-1})^2$
2. combining the results according to (A)

https://studyc.info/

Let $\begin{cases} y_k^{[0]} = A^{[0]}(\omega_{n/2}^k) \\ y_k^{[1]} = A^{[1]}(\omega_{n/2}^k) \end{cases}$

$$y_k = A(\omega_n^k) = A^{[0]}(\omega_n^{2k}) + \omega_n^{2k} A^{[1]}(\omega_n^{2k+n})$$
$$= A^{[0]}(\omega_{n/2}^k) - \omega_n^k A^{[1]}(\omega_{n/2}^k)$$
$$= y_k^{[0]} + \omega_n^k y_k^{[1]}$$

$$y_{k+n/2} = A(\omega_n^{k+n/2}) = A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+n/2} A^{[1]}(\omega_n^{2k})$$
$$= A^{[0]}(\omega_{n/2}^k) + \omega_n^{k+n/2} A^{[1]}(\omega_{n/2}^k)$$
$$= y_k^{[0]} + \omega_n^{k+n/2} y_k^{[1]} = y_k^{[0]} - \omega_n^k y_k^{[1]}$$

FFT Recursive Algorithm

```
Recursive-FFT(a)
{  n=length[a];    /* n: power of 2  */
   if n=1 the return a;
   ωn = e^(2πi/n);
   ω=1
   a[0] = (a0, a2,....., an−2);
   a[1] = (a1, a3,....., an−1);
   y[0] = Recursive-FFT(a[0]);
   y[1] = Recursive-FFT(a[1]);
   for  k=0  to  (n/2 - 1)  do
   {
        yk =  yk[0] + ωyk[1];
        yk+n/2 =  yk[0] − ωyk[1];
        ω=ωωn;                  }
}
```

https://studyc.info/

Lecture 44 NP Completeness

Polynomial Time Algorithms

- On any inputs of size n, if worst-case running time of algorithm is $O(n^k)$, for constant k, then it is called polynomial time algorithm
- Every problem can not be solved in polynomial time
- There exists some problems which can not be solved by any computer, in any amount of time, e.g. Turing's Halting Problem
- Such problems are called un-decidable
- Some problems can be solved but not in $O(n^k)$

Polynomial Time (Class P)

- These problems are solvable in polynomial time
- Problems in class P are also called **tractable**

Intractable Problems

- Problems not in P are called **intractable**
- These problems can be solved in reasonable amount of time only for small input size

Decision problems

The problems which return yes or no for a given input and a question regarding the same problem

Optimization problems

- Find a solution with best value, it can be maximum or minimum. There can be more than one solutions for it
- Optimization problems can be considered as decision problems which are easier to study.
- Finding a path between u and v using fewest edges in an un-weighted directed graph is O. P. but does a path exist from u to v consisting of at most k edges is D. P.?

NP: Nondeterministic Problems

Class NP

- Problems which are verifiable in polynomial time.
- Whether there exists or not any polynomial time algorithm for solving such problems, we do not know.
- Can be solved by nondeterministic polynomial
- However if we are given a certificate of a solution, we could verify that certificate is correct in polynomial time
- P = NP?

Nondeterministic algorithm: break in two steps

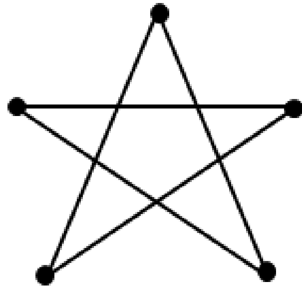
1) Nondeterministic step
   generates a candidate solution called a certificate.

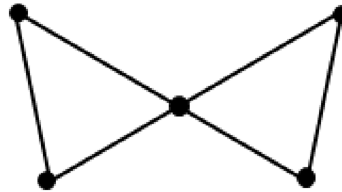
https://studyc.info/

1) Deterministic (verification) Step
   It takes certificate and an instance of problem as input, returns yes if certificate represents solution

In NP problems, verification step is polynomial

Example: Hamiltonian Cycle





Given: a directed graph G = (V, E), determine a simple cycle that contains each vertex in V, where each vertex can only be visited once

- **Certificate**:
  - Sequence: $\langle v_1, v_2, v_3, \ldots, v_n \rangle$
  - Generating certificates
- **Verification**:
  1) $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, n-1$
  2) $(v_n, v_1) \in E$

It takes polynomial time

Reduction in Polynomial Time Algorithm

Given two problems A, B, we say that A is **reducible** to B in polynomial time ($A \leq_p B$) if

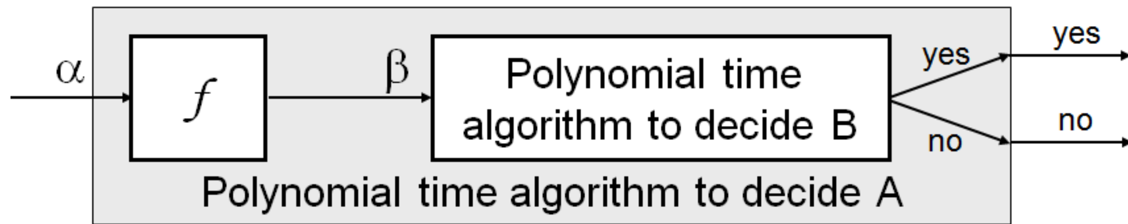
1. There exists a function f that converts the input of A to inputs of B in polynomial time
2. $A(x) = YES \Leftrightarrow B(f(x)) = YES$

where x is input for A and f(x) is input for B

Solving a decision problem A in polynomial time

- Use a polynomial time reduction algorithm to transform A into B
- Run a known polynomial time algorithm for B
- Use the answer for B as the answer for A

https://studyc.info/

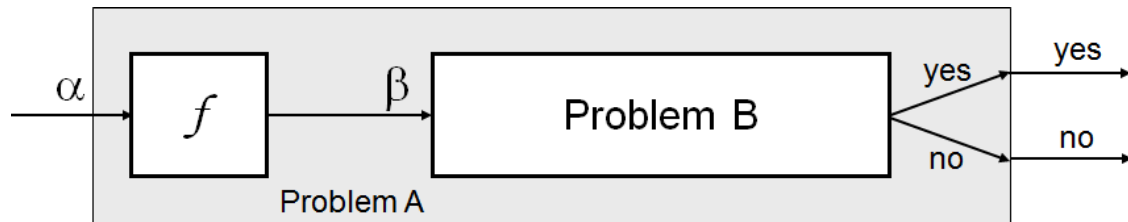


NP Complete

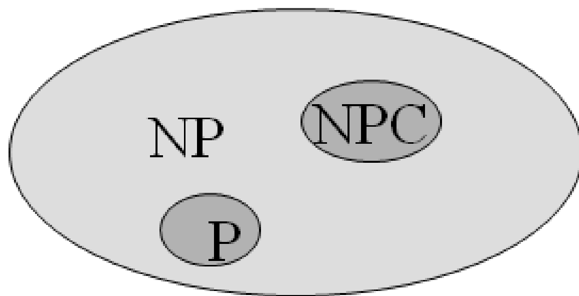
- A problem A is **NP-complete** if
  1) A $\in$ **NP**
  2) B $\leq_p$ A for all B $\in$ **NP**
- If A satisfies only property No. 2 then B is **NP-hard**
- No polynomial time algorithm has been discovered for an **NP-Complete** problem
- No one has ever proven that no polynomial time algorithm can exist for any **NP-Complete** problem

Reduction and NP Completeness

- Let A and B are two problems, and also suppose that we are given
  - No polynomial time algorithm exists for problem A
  - If we have a polynomial reduction f from A to B
- Then no polynomial time algorithm exists for B



Relation in Between P, NP, NPC



- P $\subseteq$ NP (Researchers Believe)
- NPC $\subseteq$ NP (Researchers Believe)
- P = NP (or P $\subset$ NP, or P $\neq$ NP) ???
- NPC = NP (or NPC $\subset$ NP, or NPC $\neq$ NP) ???
- P $\neq$ NP

https://studyc.info/

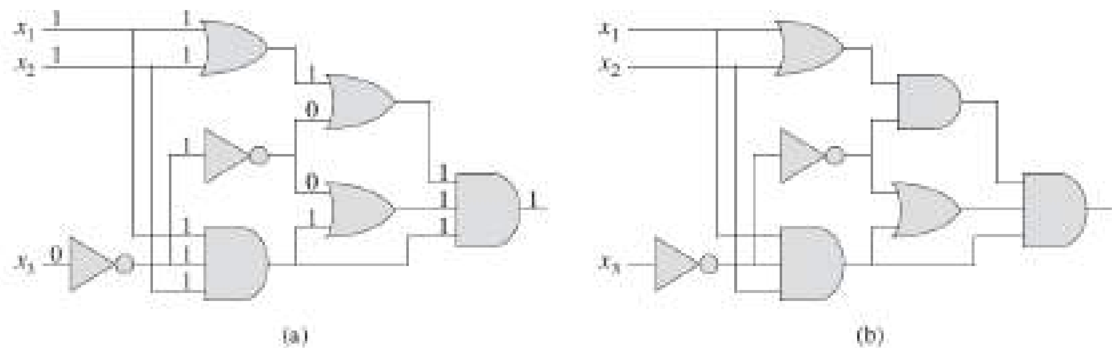
- One of the deepest, most perplexing open research problems in theoretical computer science since 1971

Problem Definitions: Circuit Satisfiability

Boolean Combinational Circuit

- *Boolean combinational elements* wired together
- *Each element* takes a set of inputs and produces a set of outputs in constant number, assume binary
- *Limit* the number of outputs to 1
- *Logic gates:* NOT, AND, OR
- *Satisfying assignment:* a true assignment causing the output to be 1.
- *A circuit is satisfiable* if it has a satisfying assignment.

Two Instances: Satisfiable and Un-satisfiable



Two instances of the circuit-satisfiability problem. (a) The assignment $<x_1 = 1, x_2 = 1, x_3 = 0>$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. (b) No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore satisfiable.

Problem: Circuit Satisfiability

Statement: Given a boolean combinational circuit composed of AND, OR, and NOT, is it satisfiable?

Intuitive solution

- For each possible assignment, check whether it generates 1.
- Suppose the number of inputs is k, then the total possible assignments are 2^k.
- So the running time is $\Omega(2^k)$.
- When the size of the problem is $\Theta(k)$, then the running time is not polynomial

Lemma 2: CIRCUIT-SAT is NP Hard

Proof:

- Suppose X is *any problem* in NP

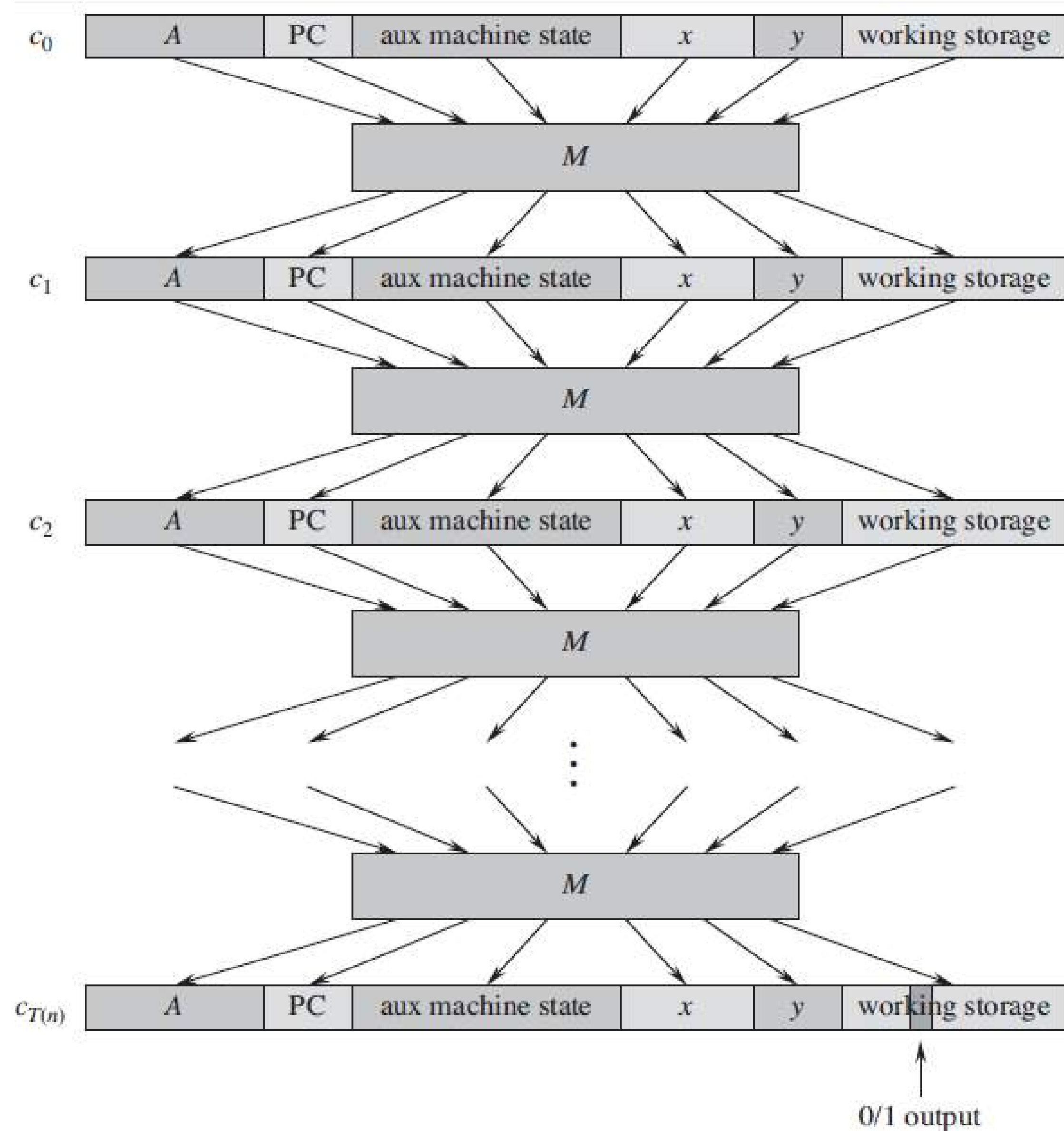
https://studyc.info/

- Construct polynomial time algorithm *F* that maps every instance x in X to a circuit C $= f(x)$ such that x is YES $\Leftrightarrow$ C $\in$ CIRCUIT-SAT (is satisfiable).
- Since X $\in$ NP, there is a polynomial time algorithm *A* which verifies X.
- Suppose the input length is n and Let $T(n)$ denote the worst-case running time.
- Let k be the constant such that $T(n) = O(n^k)$ and the length of the certificate is $O(n^k)$.

Circuit Satisfiability Problem is NP-complete

- Represent computation of *A* as a sequence of configurations, $c_0, c_1, \ldots, c_i, c_{i+1}, \ldots, c_{T(n)}$, each c_i can be broken into various components
- c_i is mapped to c_{i+1} by the combinational circuit M implementing the computer hardware.
- It is to be noted that A(x, y) = 1 or 0.
- Paste together all T(n) copies of the circuit M. Call this as, F, the resultant algorithm
- Please see the overall structure in the next slide

https://studyc.info/



The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate y, the initial configuration c_0 is constant. A Boolean combinational circuit M maps each configuration to the next configuration. The output is a distinguished bit in the working storage.

- Now it can be proved that:
    1. F correctly constructs reduction, i.e., C is satisfiable if and only if there exists a certificate y, such that $A(x, y) = 1$.
    2. F runs in polynomial time

https://studyc.info/

    - Construction of C takes $O(n^k)$ steps, a step takes polynomial time
- F takes polynomial time to construct C from x.

NP-completeness Proof Basis

Lemma 3

If X is problem such that $P' \leq_p X$ for some $P' \in NPC$, then X is NP-hard. Moreover, $X \in NP \Rightarrow X \in NPC$.

Proof:

- Since P' is NPC hence for all P" in NP, we have
  $P'' \leq_p P'$ (1)
- And $P' \leq_p X$ given (2)
- By (1) and (2)
- $P'' \leq_p P' \leq_p X \Rightarrow P'' \leq_p X$ hence X is NP-hard
- Now if $X \in NP \Rightarrow X \in NPC$

Formula Satisfiability: Notations and Definitions

- SAT Definition
  - n boolean variables: $x_1, x_2, \ldots, x_n$.
  - m boolean connectives: $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$, and
  - Parentheses.
- A SAT ϕ is satisfiable if there exists a true assignment which causes ϕ to evaluate to 1.

In Formal Language

- SAT={< ϕ>: ϕ is a satifiable boolean formula}.

SAT is NP Complete

Theorem: SAT is NP-complete.

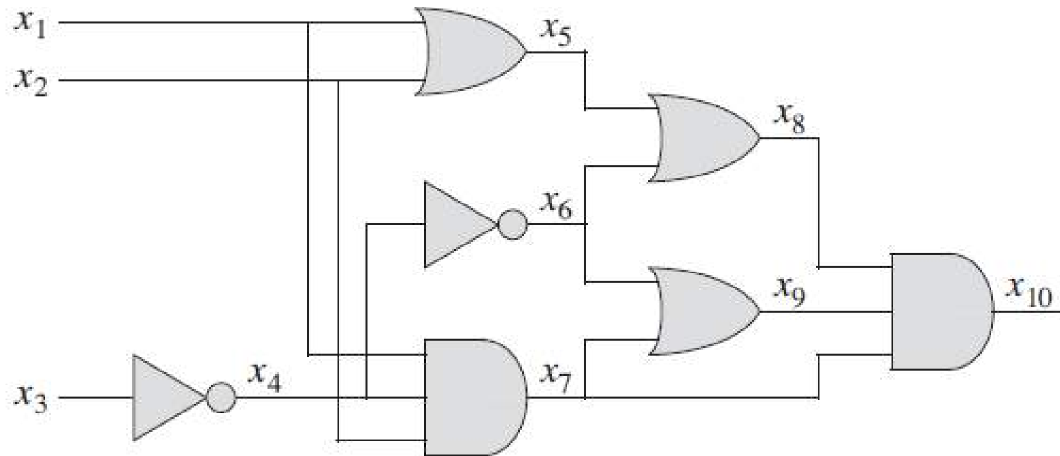
Proof:

- SAT belongs to NP.
  - Given a satisfying assignment
  - Verifying algorithm replaces each variable with its value, and evaluates formula in polynomial time.
- SAT is NP-hard
  - Sufficient to show that CIRCUIT-SAT$\leq_p$ SAT
- CIRCUIT-SAT$\leq_p$ SAT, i.e., any instance of circuit satisfiability can be reduced in polynomial time to an instance of formula satisfiability.
- Intuitive induction:
  - Look at the gate that produces the circuit output.
  - Inductively express each of gate's inputs as formulas.
  - Formula for circuit is obtained by writing an expression that applies gate's function to its input formulas.

https://studyc.info/

Unfortunately, this is not a polynomial time reduction. This is because the gate whose output is fed to 2 or more inputs of other gates, cause size to grow exponentially.

Example of Reduction of CIRCUIT-SAT to SAT



Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

$\phi= x_{10} \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$
$\wedge (x_9 \leftrightarrow (x_6 \vee x_7))$
$\wedge (x_8 \leftrightarrow (x_5 \vee x_6))$
$\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$
$\wedge (x_6 \leftrightarrow \neg x_4))$
$\wedge (x_5 \leftrightarrow (x_1 \vee x_2))$
$\wedge (x_4 \leftrightarrow \neg x_3)$

INCORRECT REDUCTION:

$\phi= x_{10}= x_7 \wedge x_8 \wedge x_9=(x_1 \wedge x_2 \wedge x_4) \wedge (x_5 \vee x_6) \wedge (x_6 \vee x_7)$

$=(x_1 \wedge x_2 \wedge x_4) \wedge ((x_1 \vee x_2) \vee \neg x_4) \wedge (\neg x_4 \vee (x_1 \wedge x_2 \wedge x_4)) = \ldots.$

NPC Proof: 3 CNF Satisfiability

Definitions:

A ***literal*** in a boolean formula is an occurrence of a variable or its negation.

Clause, OR of one or more literals.

CNF (Conjunctive Nornal Form) is a boolean formula expressed as AND of clauses.

3-CNF is a CNF in which each clause has exactly 3 distinct literals.

(a literal and its negation are distinct)

https://studyc.info/

3-CNF-SAT: whether a given 3-CNF is satiafiable?

3-CNF-SAT is NP Complete

Proof:

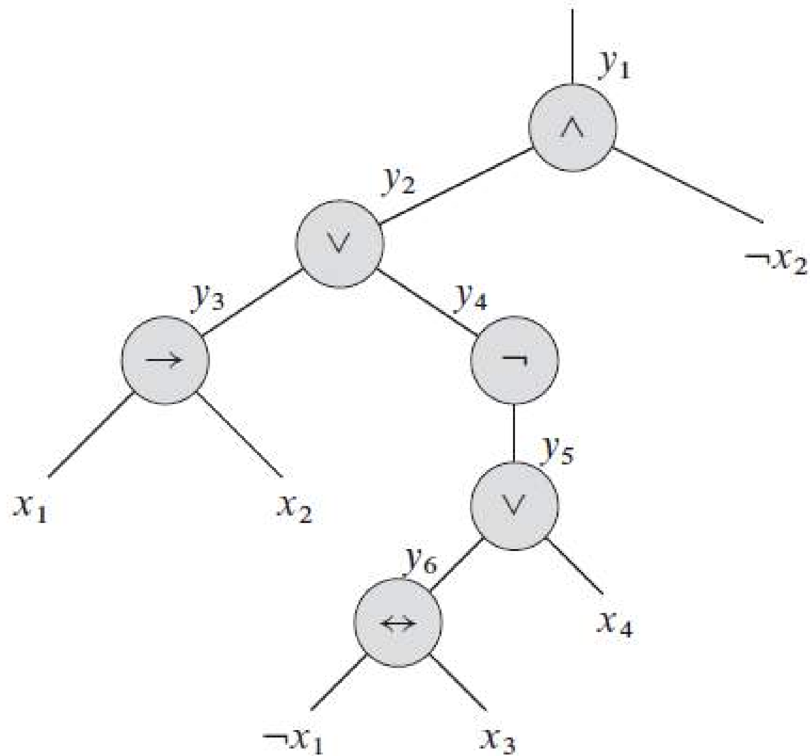
3-CNF-SAT $\in$ NP

- 3-CNF-SAT is NP-hard.
- SAT $\leq_p$3-CNF-SAT?
  - Suppose ϕ is any boolean formula, Construct a binary 'parse' tree, with literals as leaves and connectives as internal nodes.
  - Introduce y_i for output of each internal node.
  - Rewrite formula to ϕ': AND of root and conjunction of clauses describing operation of each node.
  - In ϕ', each clause has at most three literals.
- Change each clause into conjunctive normal form as:
  - Construct a true table, (at most 8 by 4)
  - Write disjunctive normal form for items evaluating 0
  - Using DeMorgan law to change to CNF.
- Result: ϕ'' in CNF but each clause has 3 or less literals.
- Change 1 or 2-literal clause into 3-literal clause as:
  - Two literals:

  $(l_1 \vee l_2)$, change it to $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$.
  - If a clause has one literal l, change it to $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$.

Binary parse tree for $\phi=((x1 \rightarrow x2) \vee \neg((\neg x1 \leftrightarrow x3) \vee x4)) \wedge \neg x2$

https://studyc.info/



The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \vee ((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

<u>Example of Converting a 3-literal clause to CNF format</u>

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

The truth table for the clause $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$

Disjunctive Normal Form:

$\phi_i' = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$

Conjunctive Normal Form:

$\phi_i''=(\neg y_1\vee\neg y_2\vee\neg x_2)\wedge(\neg y_1\vee y_2\vee\neg x_2)\wedge(\neg y_1\vee y_2\vee x_2)\wedge(y_1\vee\neg y_2\vee x_2)$

CLIQUE: NPC Proof

Definition:

- A **clique** in an undirected graph G = (V, E) is a subset V' $\subseteq$ V, each pair of V' is connected by an edge in E, i.e., clique is a complete subgraph of G.
- Size of a clique is number of vertices in the clique.
- Optimization problem: Find maximum size clique.
- Decision problem: whether a clique of given size k exists in the graph?
- CLIQUE = {<G, k>: G is a graph with a clique of size k.}

CLIQUE is NP Complete

Theorem: CLIQUE problem is NP-complete.

- Proof:
  - CLIUEQE $\in$NP: given G = (V, E) and a set V' $\subseteq$ V as a certificate for G. The verifying algorithm checks for each pair of u, $v \in$ V', whether <u, v> $\in$ E. time: $O(|V'|^2|E|)$.
  - CLIQUE is NP-hard:
    - Show 3-CNF-SAT $\leq_p$CLUQUE.
    - Surprise: from boolean formula to graph.
- Reduction from 3-CNF-SAT to CLUQUE.
  - Suppose $\phi = C_1\wedge C_2\wedge\ldots\wedge C_k$ be a boolean formula in 3-CNF with k clauses.
  - We construct a graph G = (V, E) as follows:
    - For each clause $C_r =(l_1^r\vee l_2^r\vee l_3^r)$, place triple of v_1^r, v_2^r, v_3^r into V
    - Put edge between vertices v_i^r and v_j^s when:
      - $r \neq s$, i.e. v_i^r and v_j^s are in different triples, and
      - corresponding literals are consistent, i.e, l_i^r is not negation of l_j^s
  - Then ϕ is satisfiable $\Leftrightarrow$ G has a clique of size k.

$\phi=(x1\vee\neg x2\vee\neg x3)\wedge(\neg x1\vee x2\vee x3)\wedge(x1\vee x2\vee x3)$ and its reduced graph G

https://studyc.info/

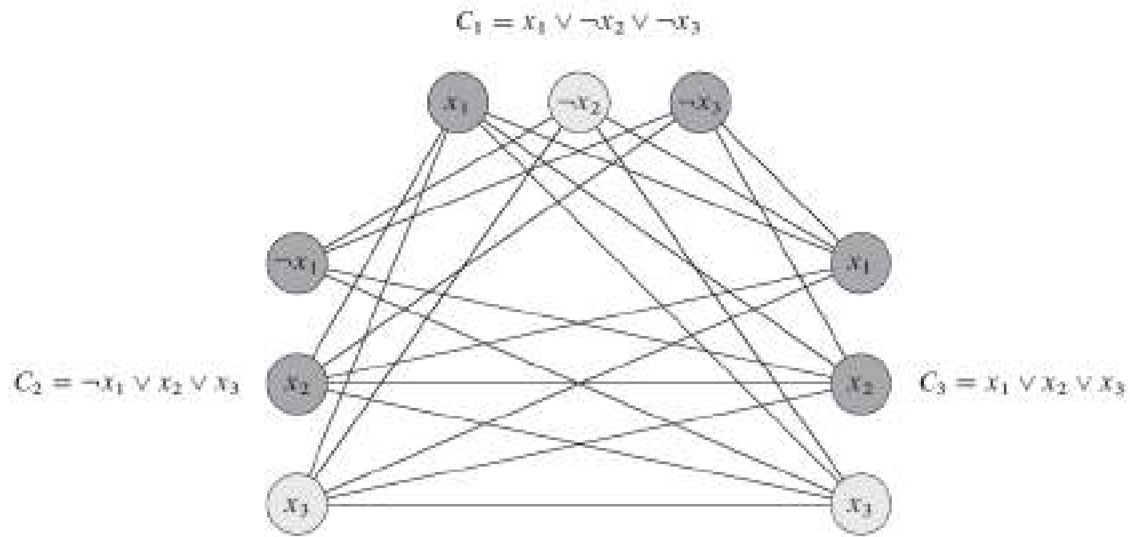
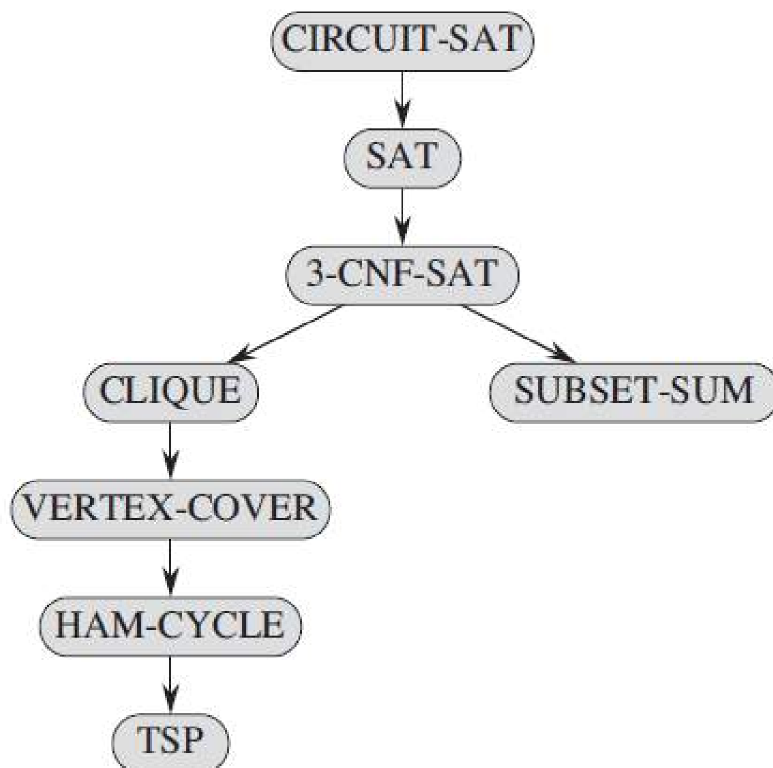


Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with lightly shaded vertices.

NP-completeness proof structure



https://studyc.info/

Lecture 45 Review of Lectures 01-44

https://studyc.info/