Advanced Algorithm Analysis

Dr. A . Sattar

Semester: Fall 2008
Computer Science Department
International Islamic University

Course Contents

The following topics will be covered in this semester:

- *Introduction to Algorithm Analysis*
- *Mathematical Preliminaries*
- *Asymptotic Analysis*
- *Recurrences*
- *Sorting Algorithm Analysis*
- *Graph Algorithms*
- *Dynamic Programming*
- *String Processing Algorithms*
- *Theory of NP-Completeness*
- *Approximation Algorithms*
- *Parallel and Distributed Algorithms*

➢ The prerequisites are *Discrete Mathematics (Combinotrics and probability theory)*, *Calculus*, and *Elementary Data Structures*.

Reference Texts

The course material will be based on, or adapted from, the following sources:

- T.Cormen, E.Leiserson, L.Rivest, C.Stein *Introduction to Algorithms*, Prentice Hall
- S.Basse,V.A.Gelder,*Computer Algorithms*, Pearson Education Inc.
- A.Levitin*, Introduction to Design and Analysis of Algorithms*, Pearson Education Inc.
- R.E.Neapolitan, K.Naimipour, *Fundamentals of Algorithms*, Heath and Company
- R.Sedgewick , P.Flajot, *Analysis of Algorithms*, National Book Foundation

Introduction to Algorithm Analysis

Introduction to Algorithm Analysis

Topics

- *Computer Algorithms*
- *Application Domains*
- *Algorithm notation*
- *Algorithm Analysis*
  - *time efficiency*
  - *space efficiency*
  - *correctness*
- *Classification of time efficiencies*
  - *Best case analysis*
  - *Worst case analysis*
  - *Average case analysis*
- *Case Study*

Computer Algorithms

Definition

An *algorithm* is an orderly step-by-step procedure to solve a problem. A *computer algorithm* has the following essential characteristics:

1) It accepts one or more inputs

2) It returns at least one output

3) It terminates after finite steps

- The term *algorithm* is derived from the title *Khowrizmi* of ninth-century Persian mathematician *Abu Musa al-Khowrizmi* , who is credited with systematic study and development of important algebraic procedures.

Algorithm Applications

Problem Domains

A large variety of problems in computer science, mathematics and other disciplines depend on the use of algorithms for their solutions. The broad categories of applications types are:

- Searching Algorithms *(Linear and non-linear)*
- Sorting Algorithms *(Elementary and Advanced)*
- Strings Processing *(Pattern matching, Parsing, Compression, Cryptography)*
- Optimization Algorithms *(Shortest routes, minimum cost)*
- Geometric Algorithms*(Triangulation, Convex Hull)*
- Image Processing *(Compression, Matching, Conversion)*
- Data Mining Algorithms*(Clustering, Cleansing, Rules mining)*
- Mathematical Algorithms *(Random number generator, matrix operations, FFT, etc)*

Algorithm Applications

Hard Problems

• There are several classic problems in computer science for which *efficient algorithms are not known* . Such problems are referred to as *hard* or *intractable*.

• Currently, over a thousand such problems have been identified. Two celebrated examples are *Hamiltonian Problem* and *Traveling Salesperson Problem*

Hamiltonian Problem

A Hamiltonian circuit , also called *tour*, is path in a graph (network) that starts and ends at the same vertex, and *passes through all other vertices exactly once*. The Hamiltonian problem is to find whether or not a given graph has Hamiltonian circuit.

Traveling Salesperson Problem

A salesperson has to travel to a given number of cities such that (1) tour starts at one city and ends up at the same city, (2) each city is visited exactly once , (3) the tour consists of *minimum total distance* .

Algorithm Notation

Specifying Algorithm Steps

Conventions

For the purpose of design and analysis the algorithms are often specified in either of the two ways:

Natural Language Specification
In this representation English language words, phrases and terminology are used to describe the crucial steps involved in an algorithm. This form is meant to highlight the design features of the algorithm

Pseudo Code Specification
In this form symbol and phrases of some high level programming to state the working of an algorithm. This form is more useful for the purpose of analysis.

Natural Language Specification

Example

This example illustrates the natural language description of an algorithm for inorder traversal of a binary tree.

Step #1: *Push tree root to stack*

Step #2: *Pop the stack. If stack is empty exit, else process the node*

Step #3: *Traverse down the tree following the left-most path, and pushing each right child onto to the stack*

Step #4: *When leaf node is reached, go back to Step #2.*

➢ The phrases and words used in natural language are not formalized. The style and choice of words vary with the algorithm design. However, all important steps and branching points must be specified unambiguously

Pseudo Code

Notation

There is no standardized notation used for the pseudo code. The following notation, based on *T . Cormen et al,* will be followed .

- ***Procedures:*** Name followed by input parameters enclosed in parentheses **FIND-MAX (*A, n*)**
- ***Assignments to variables:*** left arrows (←) e.g j← k ← p
- ***Comparisons of variables :*** symbols $x \le y$, $x \ge y$, $x \ne y$, $x = y$, $x > y$, $x < y$
- ***Logical expressions:*** connectives *exp1* **AND** *exp2* , *exp1* **OR** *exp2,* **NOT** *exp*
- ***Computations:*** arithmetic symbols +, -, *, /
- ***Exchanges (Swapping) :*** symbol ↔ e.g A[i] ↔ A[k]
- ***Loops and iterations***: Words ***for***, ***do***, ***repeat***, *while*, ***unti***l are used to describe loops, such as
  for ---- do ----, ***for-----downto ---do---***
  while ---- do---
  do ----- until---
- ***Conditionals:*** Words ***if*** and ***then*** are used to specify conditional statements, such as
  if ---- then-----else-----
- ***Block structure:*** Using *indentation* ie an inner block is displaced with respect to the outer block, as depicted below

```
do---               (start of outer most blob)
    do----          (start of next block)
        if--- else ---- ( start of a new block)
```

- ***Comments:*** symbol ► (Different from the one used in the book)
- ***Array :*** Name followed by size in brackets, e.g A[1..n]

Pseudo Code Notation

Example

Figure (*i*) shows an example of pseudo code for ***insertion sort algorithm*** . Figure *(ii)* shows the corresponding code in C++ / Java

```
for j ← 2 to n
   do key ← A[j]
      ▶Insert A[j] into sorted sequence A[1..j-1]
      i ← j - 1
      while i > 0 AND A[i] > key
              do A[i+i] ← A[i]
                    i ← i-1
      A[i+1] ← key
```

(i) Pseudo code for insertion sort

```
for ( j=2;  j <= n;  j++)
 {     key  = A[j];
       // Insert  A[j] into sorted sequence A[1..j-1]
         i =j-1;
          while ( i > 0  && A[i] > key)
                { A[i+i] = A[i];  i--; }
          A[i+1]= key;
 }
```

(ii) C++/Java code for insertion sort

Algorithm Analysis

Algorithm Analysis

Objectives

- The purpose of algorithm analysis is, in general , to determine the performance of the algorithm in terms of time taken and storage requirements to solve a given problem.

- An other objective can be to check whether the the algorithm produces consistent, reliable, and accurate outputs for all instances of the problem .It may also ensured that algorithm is robust and would prove to be failsafe under all circumstances.

- The common metrics that are used to gauge the performance are referred to as *time efficiency*, *space efficiency*, and *correctness*.

Algorithm Analysis

Input Size

• The performance is often expressed in terms *problem size*, or more precisely, by the number of data item items processed by an algorithm.

• The key parameters used in the analysis of algorithms for some common application types are:

i. Count of data items in data collections such as arrays, queues

ii. Number of nodes in trees and linked lists

iii. Number of vertices and number of edges in a graph

iv. Number of rows and columns in an input table

v. Character count in an input text block

Analysis of Algorithm

Time Efficiency

- The *time efficiency* determines how fast an algorithm solves a given problem. Quantitatively, it is the measures of time taken by the algorithm to produce the output with a given input. The time efficiency is denoted by a mathematical function of the input size.

- Assuming that an algorithm takes *T(n)* time to process *n* data items. The function *T(n)* is referred to as the *running time* of the algorithm. It is also known as *time complexity.* The time complexity can depend on more than one parameter. For example, the running time of a graph algorithm can depend both on the number of vertices and edges in the graph.

- The running time is an important indicator of the behavior of an algorithm for varying inputs. It tells, for example, whether the time taken to process would increase in direct proportional to input size, would increase four fold if size is doubled, or increase exponentially.

- It would be seen that that time efficiency is the most significant metric for algorithm analysis. For this reason, the main focus of algorithm analysis would be on determining the time complexity.

Time Efficiency
Approaches

The time efficiency can be determined in several ways . The common methods are categorized as *empirical*, *analytical*, and *visualization*

- In empirical approach , the running time is determined experimentally. The performance is measured by testing the algorithm with inputs of different sizes

- The analytical method uses *mathematical* and *statistical* techniques to examine the time efficiency. The running time is expressed as mathematical function of input size

- The visualization technique is sometimes used to study the behavior and performance of an algorithm by generating graphics and animation, based on interactive inputs .

Empirical Analysis

Methodology

The empirical methodology broadly consists of following steps

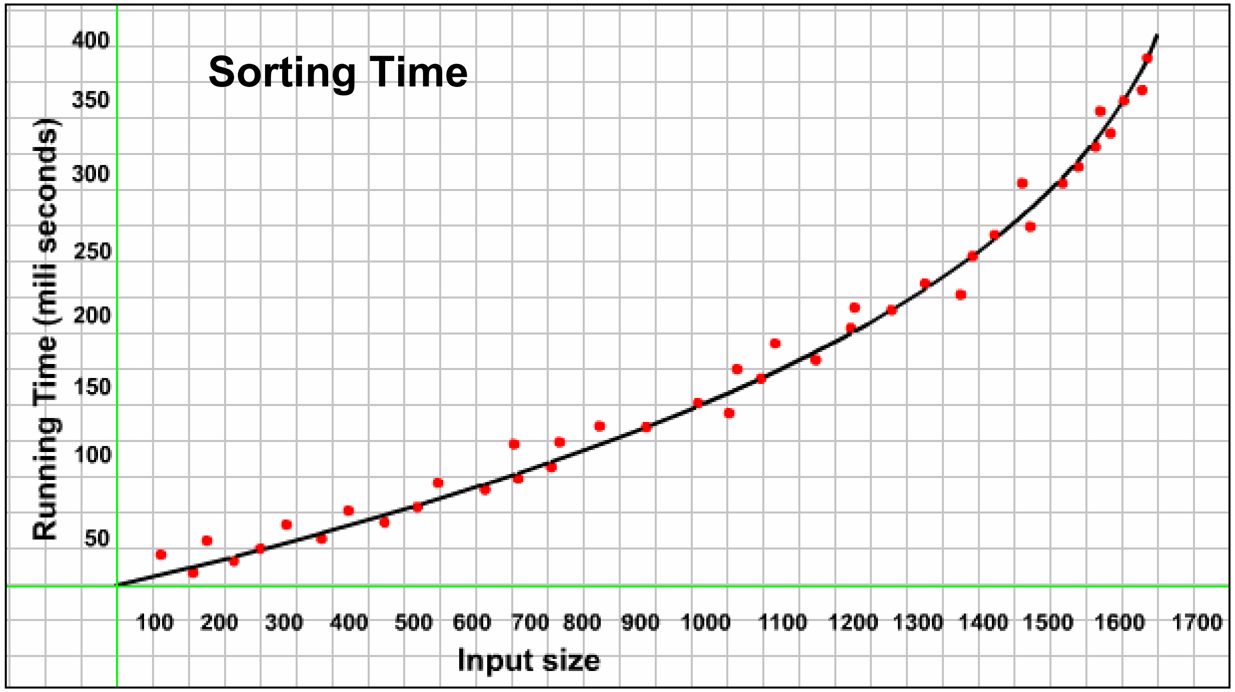
1) The algorithm is coded and run on a computer. The running times are measured, by using some timer routine , with inputs of different sizes..

2) The output is logged and analyzed with the help of *graphical* and *statistical* tools to determine the behavior and growth rate of running time in terms of input size.

3) A *best curve* is fitted to depict trend of the algorithm in terms input sizes

Empirical Analysis
Example

• The graph illustrates the results of an *empirical analysis.* The running times of a sorting algorithm are plotted against the number input sort keys. The measured values are shown in red dots. The graph shows the *best-fit curve* to scattered points.



• The analysis indicates that time increases roughly in proportion to the square of input size, which means that doubling the input size increases the running time four fold.

Empirical Analysis

Limitations

The empirical analysis provides estimates of running times in a real life situation. It has, however, several limitations, because the running time crucially depends on several factors. Some key factors that influence the time measurements are:

- Hardware types *(CPU speed, IO throughput, RAM size etc.)*
- Software environment *(Compiler, Programming Language etc.)*
- Program design *(Conventional, Structured, Object Oriented)*
- Composition of data set sets (*Choice of data values and the range of input)*

Analytical Analysis

Methodology

In analytic approach the running time is estimated by studying and analyzing the *basic* or *primitive operations* involved in an algorithm. Broadly, the following methodology is adopted:

- The code for the algorithm is examined to identify basic operations
- The number of times each basic operation is executed, for a given input, is determined.
- The running time is estimated by taking into consideration the frequency and cost of significant operations
- The total time is expressed as a function of the input size

Analytical Analysis

Basic Operations

• The code for an algorithm, generally, consists of a mix of following basic operations:.

i. *Assigning value to a variable*

ii. Comparing a pair of data items

iii. Incrementing a variable

iv. Performing arithmetic and floating point operations

v. Moving a data item from one storage location to another location

vi. Calling a procedure

vii. Returning a value

viii. Accessing an array element

• The time taken by the basic operation to complete a single step is often referred to as the *cost* of the operation. Some operations are relatively more expensive than others. For example, operations involving data movement and arithmetical computations are costly compared to logical or assignment operations .

Running Time Classification

Worst, Best, Average Cases

- We have seen that the running time of an algorithm depends on the *input size*. For some applications, the running time also depends on the *order* in which the data items are input to the algorithm.

- Let k denote *all possible orderings of input of size n*, and $T_1(n), T_2(n), .. T_k(n))$ be the running times for each instance . We can formally, define the ***best, worst*** and ***average*** running times, $T_{best}(n)$, $T_{worst}(n)$, $T_{average}(n)$, as follows

Best Case: In this case the algorithm has *minimum* running time.

: $T_{best}(n) = minimum(T_1, T_2, \ldots T_k)$

This is also called the *optimistic* time

Worst Case: In this case the algorithm has *maximum* running time

$T_{worst}(n) = maximum(T_1, T_2, \ldots T_k)$

This is also known as *pessimistic* time

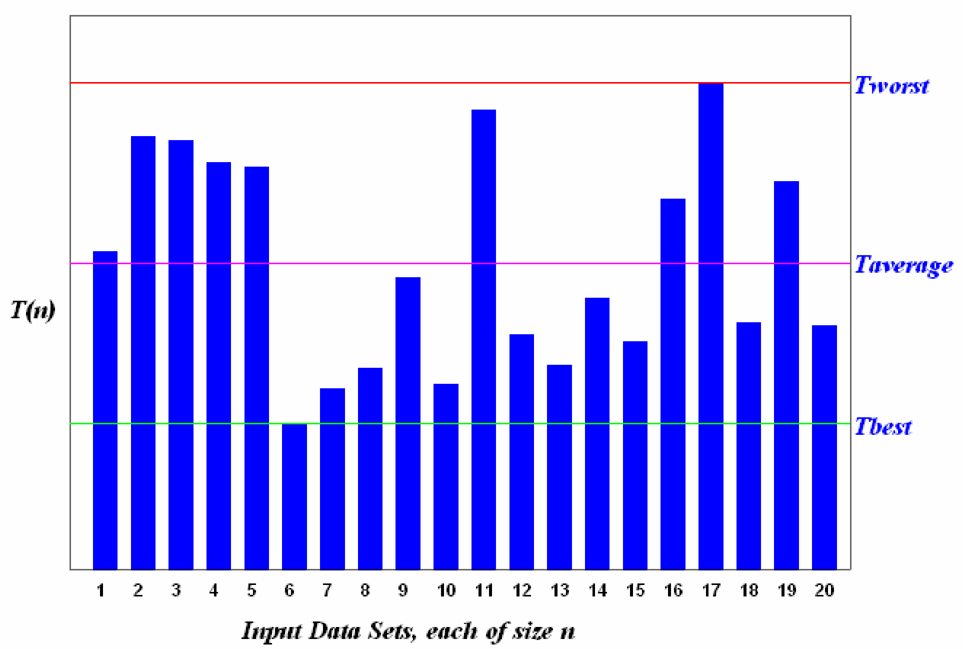
Average Case: The *average* running time is the *average of running times for all possible ordering of inputs of the same size:*

$T_{average}(n) = (T_1 + T_2 + \ldots. + T_k) / k$

Best, Worst, Average Cases

Example

• The graph shows *best*, *worst*, and *average* running times of an algorithm. In this example, times are shown for *20* inputs of *same size* but *in different order*. The algorithm takes *minimum time* to process *Input* #6 , and *maximum time* to process *Input* #17 , which are referred to as *best* and *worst* times. The *average* of all the running times for *20* inputs is also shown.



• For an accurate analysis , *all possible arrangements* of input should be considered to identify the best, worst, and average running times

Algorithm Analysis

Space Efficiency

• The space efficiency determines total memory requirement of RAM and disk storage to run an algorithm with given input.

• The space requirement consists of the amount of real storage needed to execute the algorithm code and the storage to hold the application data. The algorithm code occupies a fixed amount of space, which is independent of the input size. The storage requirement for the application depends on the nature of data structure used to provide faster and flexible access to stored information . For an arrays and linked lists, for example, the space requirement is directly proportional to the input size.

• For most algorithms, the space efficiency is not of much concern. However, some algorithms require extra storage to hold results of intermediate computations. This is the case, for example, with merge sort and dynamic programming techniques.

Algorithm Correctness

Loop Invariant Method

- There are no standard methods for proving the correctness of a given algorithm. However, many useful algorithms consist of one or more *iterative computations*, using loop structures. The correctness of such algorithms can be formally established by a technique called *Loop Invariant Method* .

- A *loop invariant* is set of *conditions* and *relationships* that remain true *prior to*, *during* , and *after* the execution of a loop.

- The loop invariant condition / statement depends on the nature of problem being analyzed In a *sorting problem*, for example, the condition might be the order of keys in a *sub-array*, which should remain in ascending/descending order ,prior to, and after the execution of each iteration

Algorithm Correctness

Formal Proof

The loop invariant method establishes the correctness of algorithm in three steps, which are known as *initialization, maintenance*, and *termination* *(* Reference*: T. Cormen et al)* At each step, the *loop invariant* is examined. If the loop conditions at each of the steps hold true, then algorithm is said be *correct*.

Initialization: Loop invariant is true prior to execution of *first iteration* of the loop

Maintenance: If loop invariant is assumed to be true at *some iteration***,** it remains true *after the next iteration*

Termination: After the termination of the loop, the invariant holds true for the problem size

Algorithm Analysis

Visualization

Algorithm visualization techniques are used to study

- *Studying inner working of an algorithm through trace of basic operations*
- *Illustrating algorithm steps with animations*
- *Studying algorithm performance with interactive inputs*
- *Counting primitive operations for analysis*
- *Doing simulations with a variety of data sets*
- *Reporting on the performance*

Case Study
Algorithm Analysis

Case Study

Algorithm Analysis

This case study is meant to demonstrate the salient features of algorithm design and analysis, as discussed previously. A simple example is used to systematically describe the following steps.:

- *Problem statement*
- *Algorithm design*
- *Implementation using pseudo code*
- *Analysis of best, worst and average running times*
- *Space complexity*
- *Correctness of algorithm*
- *Visualization of Analysis*

Case Study

Problem Statement

- Design an algorithm to find maximum element in an array of size ***n***
- Analyze the algorithm to determine :
  - *Time efficiency*
  - *Space efficiency*
  - *Correctness*

Case Study

Algorithm Design

The design features are expressed in plain language. The algorithm for the solution of the problem consists of the following steps:

Step #1: *Store the first array element in variable* ***max***

Step #2: *Scan array to compare* ***max*** *with other elements*

Step #3: *Replace* ***max*** *with a larger element, when found during the scan*

Step #4: *Return value held by* ***max***

Algorithm Analysis

Pseudo Code

The procedure **FIND-MAX** returns maximum element in an array. The array ***A*** *and its size* ***n*,** are passed as arguments. The following pseudo code describes the essential steps, together with comments which are identified by the symbol ►*:*

```
FIND-MAX ( A, n )

1   max ← A[1]                   ► Store first array element into variable max
2   for j ← 2 to n do            ► Scan remaining elements
3       if ( A[j] > max )        ► Compare an element with max
4           then max ← A[j]      ► Replace max with a larger element
5   return max                   ► Return maximum element
```

Running Time

Costs of Basic Operations

First, we identify *basic operations* and their associated costs . The table below lists various operations and costs. Here the term 'cost' refers to the time consumed in executing an operation.

#	*Statement*	*Unit costs*	*Remarks*
1	*max ←A[1]*	*Ca* *Cs*	*Cost of **accessing** A[1]* *Cost of **storing** A[1] into **max***
2	***for** j←2 to n **do***	*Cs* *Cc* *Ci*	*Cost of **storing** 2 into **j*** *Cost of **comparing** index **j** with **n**, and branching* *Cost of **incrementing j***
3	***if** (A[j] > max)*	*Ca* *Cc*	*Cost of **accessing** A[j]* *Cost of **comparing** A[j] with **max,** and branching*
4	***then** max ← A[j]*	*Ca* *Cs*	*Cost of **accessing** A[j]* *Cost of **storing** A[j] into **max***
5	***return** max*	*Cr*	*Cost of **returning** maximum element*

Table of costs of basic operations

Running Time

Counts of Basic Operations

Next we count the number of the *basic operations,* and total cost of executing each statement. The frequency of execution of statement *4* depends on the outcome of statement *3*; it will be executed when the condition *A[j]>max* turns out to be true. This condition ,in turn, depends on the order of data in the array *A*. For the purpose of analysis, we assume that statement *4* is executed *k* times, where $0 \leq k \leq n-1$.

#	***Statement***	***Unit Cost***	***Operations Count***	***Total Cost***
1	*max ←A[1]*	*Ca* *Cs*	*1* *1*	*Ca + Cs*
2	***for*** *j←2 to n* ***do***	*Cs* *Cc* *Ci*	*1* *n-1* *n-1*	*Cs + (n-1).Cc +(n-1).Ci*
3	***if*** *(A[j] > max)*	*Ca* *Cc*	*n-1* *n-1*	*(n-1).Ca + (n-1).Cc*
4	***then max ← A[j]***	***Ca*** ***Cs***	***k*** ***k***	***(Ca + Cs). k, where k depends on condition in statement 3 . In general, 0≤ k ≤ n -1***
5	***return*** *max*	*Cr*	*1*	*Cr*

Table of frequency and total cost of basic operations

Running Time

Aggregate Cost

- The *running time T(n)* is obtained by adding the costs in the last column of the table

  $Tn) = Ca + Cs$

  $\quad + \; Cs + (n-1).Cc$

  $\quad + \; (n-1).Ci + (n-1).Ca$

  $\quad + \; (n-1).Cc \; + \; (Ca + Cs).\,k$

  $\quad + \; Cr$

- Simplifying and rearranging, we get following expression .

  $\boldsymbol{T(n) = A + B.k + C.n}$,

where $A = 2Cs - 2Cc - Ci + Cr,$

$B = Ca + Cs,$

$C = 2Cc + Ci + Ca$

- The constants *A, B, C* depend on the computing environment

Running Time Classification
Best, Worst, Average Cases

The running time *T(n)* of the algorithm for finding the maximum element of array of size *n* is given by

$$T(n) = A + B.k + C.n, \quad \text{where } 0 \leq k \leq n-1$$

Here ***k*** is the number of times the statement ***max ← A[j]*** will be executed**.** The following possibilities can arise:

Best Case: Best case occurs when the statement is not executed at all. This happens when the array maximum element occurs in the *first* cell**.** In this case $k = 0$, and *best (minimum) running time* is given by

$$T_{best}(n) = A + C.n$$

Average Case: In this case, the statement is executed on an average $n / 2$ times so that $k = n / 2$. Thus, *average time running time* is given by

$$T_{average}(n) = A + (B / 2 + C).n$$

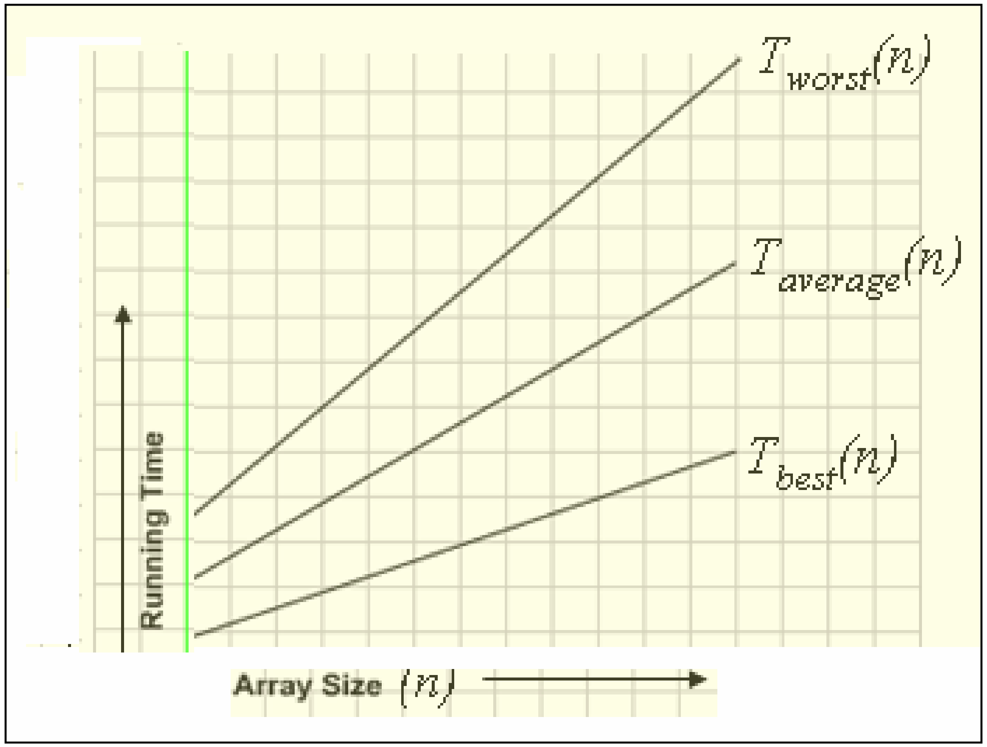
Worst Case: In this case, the statement is executed $n-1$ times; so $k = n-1$. This happens when the array is *sorted in ascending order* .Thus, *worst (maximum) running time* is given by

$$T_{worst}(n) = A-B + (B + C).n$$

Running Time Classification

Plot of Best Worst and Average Cases

A plot of running times for different cases is shown below. The running time in all of the three cases increases linearly. However, the *slope of line* (*rate of increase)* is different. In the case of worst time, for example, the running time increases at a greater rate.



Algorithm Analysis

Correctness

- The ***correctness*** of FIND-MAX algorithm, listed below, is established by the ***loop invariant*** method.

```
FIND-MAX(A)
1   max ← A[1]
2   for j←2 to n do
3     if( A[j] > max)
4     then    max ← A[j]
1   return max
```

- First, we define the *loop invariant S* as the following statement:

Variable max holds the largest value at all stages of loop execution

• Next, we consider the steps of *initialization*, *maintenance*, and *termination*

.

Algorithm Analysis

Correctness

• The ***Initialization*** condition requires that prior to first iteration the statement *S* should be true**.** This is trivially (also called ***vacuously***) true, because at this stage ***max*** contains the single element *A[1]*.

• The ***Maintenance*** condition requires that if S is true before an iteration of loop, it should remain true after the iteration It can be easily verified that if ***max holds the largest of k elements***, ***after k^th^ iteration,*** then it holds ***largest of k+1 elements after the next iteration. ie.(k+1)^st^ iteration***

• The ***Termination*** condition requires that post-condition should be true for problem size i.e, ***max*** should return maximum array element. The loop terminates when index *j* exceeds *n*. This implies that just after the ***last iteration max holds the largest of the first n elements*** of the array. Since ***array has size n***, it means that ***max returns the largest of array elements***.

Analysis of Algorithm

Space Efficiency

The *space analysis of algorithm* for finding maximum element is simple and straightforward. It amounts to determining space utilization as function of data structure size.

- The total space requirement consists of memory used by the *program statements* and array element. The former is a fixed and does not depend on array size.

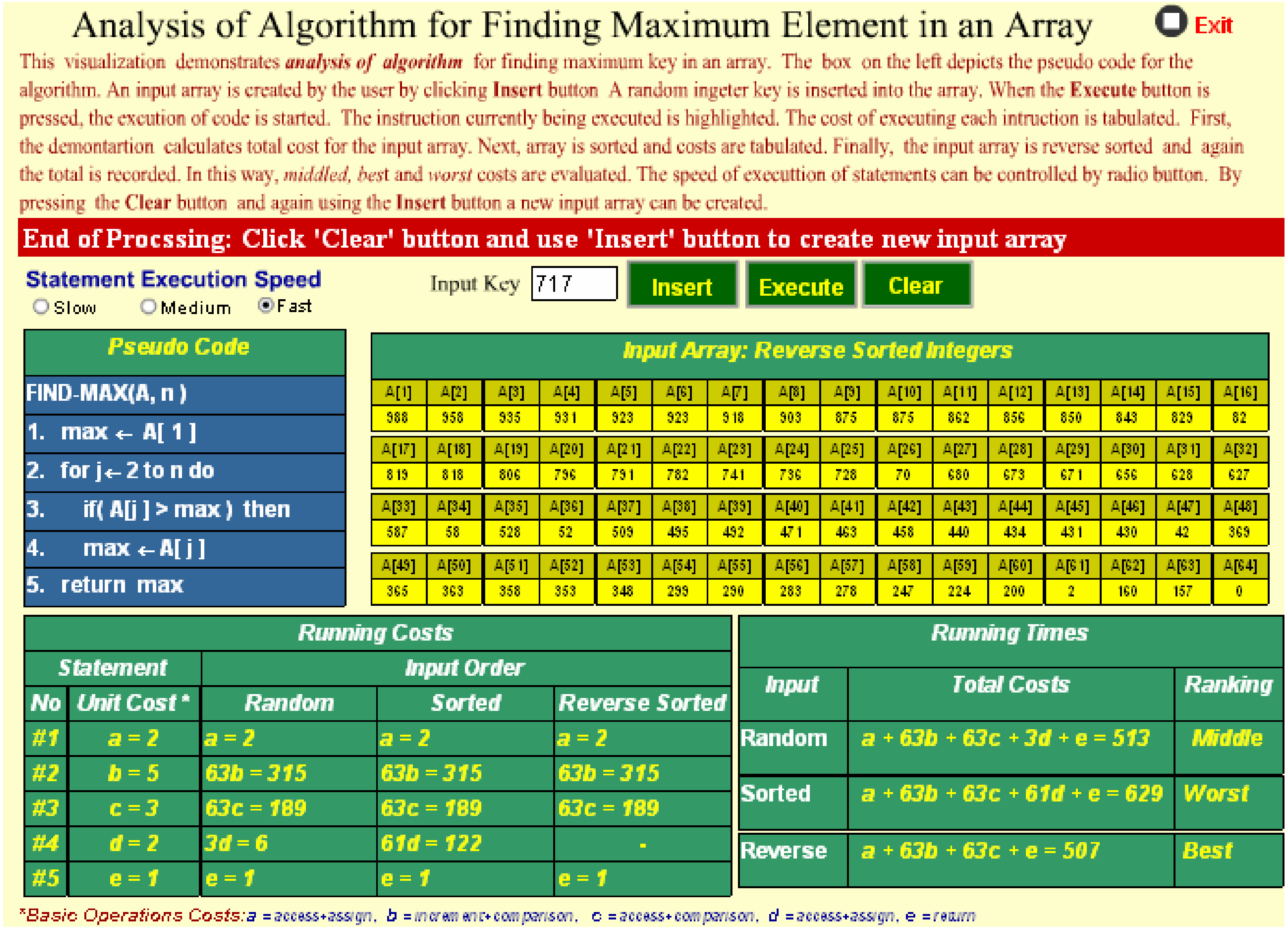
- The amount of storage requirement for the *array depends on the nature of data type* (integer, floating point, strings). It increases in direct proportion to the array size

- Thus, *space efficiency* is given by

$$S(n) = \underbrace{A}_{\textit{Program space requirement}} + \underbrace{B.n}_{\textit{Array space requirement}}$$

Visualization

Visualization



Analysis of Algorithm for Finding Maximum Element in an Array

Exit

This visualization demonstrates *analysis of algorithm* for finding maximum key in an array. The box on the left depicts the pseudo code for the algorithm. An input array is created by the user by clicking **Insert** button A random ingeter key is inserted into the array. When the **Execute** button is pressed, the excution of code is started. The instruction currently being executed is highlighted. The cost of executing each intruction is tabulated. First, the demontartion calculates total cost for the input array. Next, array is sorted and costs are tabulated. Finally, the input array is reverse sorted and again the total is recorded. In this way, *middled, best* and *worst* costs are evaluated. The speed of executtion of statements can be controlled by radio button. By pressing the **Clear** button and again using the **Insert** button a new input array can be created.

End of Procssing: Click 'Clear' button and use 'Insert' button to create new input array

Statement Execution Speed

○ Slow ○ Medium ● Fast

Input Key 717 | Insert | Execute | Clear

Pseudo Code

```
FIND-MAX(A, n )
1.  max ← A[ 1 ]
2.  for j ← 2 to n do
3.      if( A[j ] > max ) then
4.      max ← A[ j ]
5.  return max
```

Input Array: Reverse Sorted Integers

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]	A[16]
988	958	935	931	923	923	918	903	875	875	862	856	850	843	829	82
A[17]	**A[18]**	**A[19]**	**A[20]**	**A[21]**	**A[22]**	**A[23]**	**A[24]**	**A[25]**	**A[26]**	**A[27]**	**A[28]**	**A[29]**	**A[30]**	**A[31]**	**A[32]**
819	818	806	796	791	782	741	736	728	70	680	673	671	656	628	627
A[33]	**A[34]**	**A[35]**	**A[36]**	**A[37]**	**A[38]**	**A[39]**	**A[40]**	**A[41]**	**A[42]**	**A[43]**	**A[44]**	**A[45]**	**A[46]**	**A[47]**	**A[48]**
587	58	528	52	509	495	492	471	463	458	440	434	431	430	42	369
A[49]	**A[50]**	**A[51]**	**A[52]**	**A[53]**	**A[54]**	**A[55]**	**A[56]**	**A[57]**	**A[58]**	**A[59]**	**A[60]**	**A[61]**	**A[62]**	**A[63]**	**A[64]**
365	363	358	353	348	299	290	283	278	247	224	200	2	160	157	0

Running Costs

Statement		Input Order		
No	Unit Cost *	Random	Sorted	Reverse Sorted
#1	a = 2	a = 2	a = 2	a = 2
#2	b = 5	63b = 315	63b = 315	63b = 315
#3	c = 3	63c = 189	63c = 189	63c = 189
#4	d = 2	3d = 6	61d = 122	-
#5	e = 1	e = 1	e = 1	e = 1

Running Times

Input	Total Costs	Ranking
Random	a + 63b + 63c + 3d + e = 513	Middle
Sorted	a + 63b + 63c + 61d + e = 629	Worst
Reverse	a + 63b + 63c + e = 507	Best

*Basic Operations Costs: a = access+assign, b = increment+comparison, c = access+comparison, d = access+assign, e = return