ANÁLISE DE COMPLEXIDADE

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO 1º SEMESTRE DE 2007

Antonio Alfredo Ferreira Loureiro
loureiro@dcc.ufmg.br
http://www.dcc.ufmg.br/~loureiro

Algoritmos

- Os algoritmos fazem parte do dia-a-dia das pessoas. Exemplos de algoritmos:
  - instruções para o uso de medicamentos;
  - indicações de como montar um aparelho;
  - uma receita de culinária.
- Seqüência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema.
- Segundo Dijkstra, um algoritmo corresponde a uma descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações.
  - Executando a operação $a + b$ percebemos um padrão de comportamento, mesmo que a operação seja realizada para valores diferentes de a e b.

O papel de algoritmos em computação

- Definição: um **algoritmo** é um conjunto finito de instruções precisas para executar uma computação.
  - ➜ Um algoritmo pode ser visto como uma ferramenta para resolver um problema computacional bem especificado.
- Um algoritmo pode receber como entrada um conjunto de valores e pode produzir como saída um outro conjunto de valores.
  - ➜ Um algoritmo descreve uma seqüência de passos computacionais que transforma a entrada numa saída, ou seja, uma relação entrada/saída.
- O vocábulo algoritmo origina do nome *al-Khowarizmi*.

Origem do vocábulo algoritmo

Abu Ja'Far Mohammed Ibn Musa al-Khowarizmi (780–850), astrônomo e matemático árabe. Era membro da "Casa da Sabedoria", uma academia de cientistas em Bagdá. O nome al-Khowarizmi significa da cidade de Khowarizmi, que agora é chamada Khiva e é parte do Uzbequistão. al-Khowarizmi escreveu livros de matemática, astronomia e geografia. A álgebra foi introduzida na Europa ocidental através de seus trabalhos. A palavra álgebra vem do árabe al-jabr, parte do título de seu livro *Kitab al-jabr w'al muquabala*. Esse livro foi traduzido para o latim e foi usado extensivamente. Seu livro sobre o uso dos numerais hindu descreve procedimentos para operações aritméticas usando esses numerais. Autores europeus usaram uma adaptação latina de seu nome, até finalmente chegar na palavra **algoritmo** para descrever a área da aritmética com numerais hindu.

Algoritmo e modelo computacional (1)

- Modelo:
  - Esquema que possibilita a representação de uma entidade (Houaiss).
  - ➜ No modelo, só se deve incluir o que for relevante para a modelagem do objeto em questão.
- Computacional:
  - Relativo ao processamento (Houaiss.)
- Definição (nosso contexto):
  - Esquema que descreve como é o modelo abstrato do processamento de algoritmos.

Algoritmo e modelo computacional (2)

- **Importância**:
  - Um algoritmo não existe, ou seja, não é possível escrevê-lo, se antes não for definido o modelo computacional associado (como será executado).
  - ➜ Conceito básico no projeto de qualquer algoritmo.
- Questão decorrente:
  - Dado um problema qualquer, existe sempre um algoritmo que pode ser projetado para um dado modelo computacional?
  - ➜ Não! Em vários casos é possível mostrar que não existe um algoritmo para resolver um determinado problema considerando um modelo computacional.

Algoritmo e modelo computacional (3)

- Que modelos existem?
  - Literalmente dezenas deles.
  - Se não estiver satisfeito, invente o seu!
- O mais popular (usado) de todos:
  - RAM – Random Access Machine.
  - ➜ Modela o computador tradicional e outros elementos computacionais.

Algoritmo e modelo computacional: Modelo RAM (4)

- Elementos do modelo:
  - um único processador;
  - memória.
- Observações:
  - Podemos ignorar os dispositivos de entrada e saída (teclado, monitor, etc) assumindo que a codificação do algoritmo e os dados já estão armazenados na memória.
  - Em geral, não é relevante para a modelagem do problema saber como o algoritmo e os dados foram armazenados na memória.

Algoritmo e modelo computacional: Modelo RAM (5)

- Computação nesse modelo:
  - Processador busca instrução/dado da memória.
  - Uma única instrução é executada de cada vez.
  - Cada instrução é executada seqüencialmente.

- Cada operação executada pelo processador, incluindo cálculos aritméticos, lógicos e acesso a memória, implica num **custo de tempo**:
  ➜ Função de complexidade de tempo.

- Cada operação e dado armazenado na memória, implica num **custo de espaço**:
  ➜ Função de complexidade de espaço.

Complexidade de tempo e espaço

- A complexidade de tempo não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

- A complexidade de espaço representa a quantidade de memória (numa unidade qualquer) que é necessário para armazenar as estruturas de dados associadas ao algoritmo.

- Usa-se a notação assintótica para representar essas complexidades:
  - O (O grande);
  - Ω (Ômega grande);
  - Θ (Teta);
  - o (o pequeno);
  - ω (ômega pequeno).

Modelo computacional para sistemas distribuídos

- Mundo distribuído:
  - Normalmente os elementos computacionais seguem o modelo RAM que são interconectados através de algum meio e só comunicam entre si através de troca de mensagens.
  ➜ Não existe compartilhamento de memória.

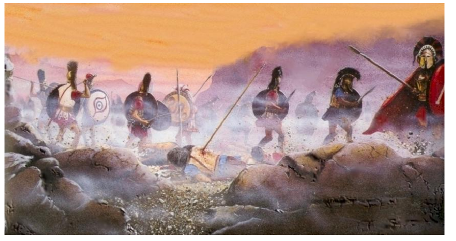
- Elementos desse modelo:
  - Nó computacional representado pelo modelo RAM.
  - Canal normalmente representado pelo modelo FIFO (first-in, first-out).

Problema dos dois exércitos (1)

Na Grécia antiga, lugares maravilhosos como este ...

Vale perto de Almfiklia, Grécia

...podiam se transformar em cenários de guerra.



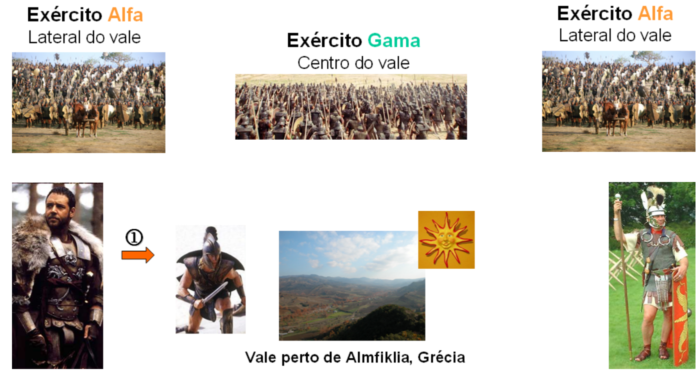
➜ É quando algum filósofo propõe o "Problema dos dois exércitos".

Problema dos dois exércitos (2)
Cenário inicial

Vale perto de Almfiklia, Grécia

- Exército **Alfa** está em maior número que o exército **Gama** mas está dividido em duas metades, cada uma numa lateral do vale.
- Cada metade do exército **Alfa** está em menor número que o exército **Gama**.
- Objetivo do exército **Alfa**: coordenar um ataque ao exército **Gama** para ganhar a guerra.

Problema dos dois exércitos (3)
O problema da coordenação



Vale perto de Almfiklia, Grécia

1. General do exército **Alfa**, do lado esquerdo do vale, chama o seu melhor soldado para levar uma mensagem para o general do exército **Alfa** do lado direito:
   Vamos atacar conjuntamente o exército **Gama** amanhã às 6:00h?
   Observações:
   - A **única** possibilidade de comunicação entre os dois generais é através de um mensageiro.
   - Os dois generais têm um "relógio perfeitamente sincronizado", ou seja, eles sabem pela posição do sol quando é 6:00h.

Problema dos dois exércitos (4)
O problema da coordenação

Vale perto de Almfiklia, Grécia

2. O soldado do exército **Alfa** atravessa as linhas inimigas e leva a mensagem até o general do outro lado.

Problema dos dois exércitos (5)
O problema da coordenação

Vale perto de Almfiklia, Grécia

3. O general do exército **Alfa** do lado direito concorda em atacar o exército **Gama** no dia seguinte às 6:00h.

Problema dos dois exércitos (6)
O problema da coordenação

4. O soldado do exército **Alfa** atravessa novamente as linhas inimigas e confirma com seu general o ataque para o dia seguinte.

Problema dos dois exércitos (7)
O problema da coordenação

➜ Após esses quatro passos terem sido realizados com sucesso, vai haver ataque amanhã às 6:00h?

O problema dos dois robôs (1)

- Imagine dois ou mais robôs que vão carregar uma mesa de tal forma que um ficará de frente para outro.
- Problema:
  - Projete um algoritmo para coordenar a velocidade e direção do movimento de cada robô para que a mesa não caia.
  - ➜ Os robôs só podem comunicar entre si através de um canal de comunicação sem fio.
  - ➜ Variante do problema anterior!

O problema dos dois robôs (2)

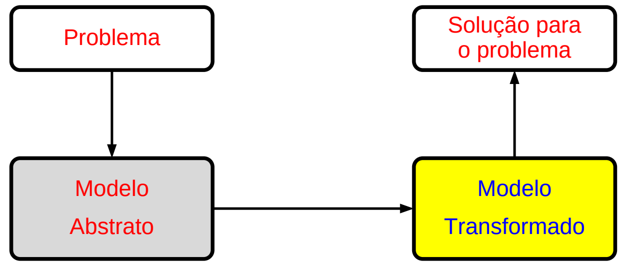
➜ É possível projetar um algoritmo distribuído para esse problema?
NÃO! Não existe um algoritmo distribuído para o problema de coordenação considerando o modelo computacional proposto!

Alguns comentários sobre algoritmos distribuídos

- São a base do mundo distribuído, ou seja, de sistemas distribuídos.
- Sistemas distribuídos podem ser:
  - Tempo real ou não;
  - Reativos ou não.
- Sistemas distribuídos podem ser especificados tomando-se como base:
  - tempo;
  - eventos.

Modelagem Matemática

- Metodologia: conjunto de conceitos que traz coesão a princípios e técnicas mostrando quando, como e porque usá-los em situações diferentes.
- A metodologia que usa matemática na resolução de problemas é conhecida como modelagem matemática.
- O processo de modelagem:



Exemplo de modelagem: Malha rodoviária (1)

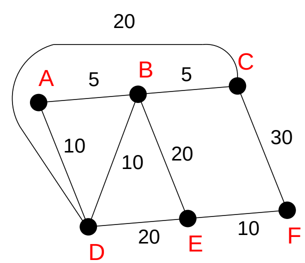
Suponha a malha rodoviária entre as seis cidades A, B, C, D, E, e F.

Problema: Achar um subconjunto da malha rodoviária representada pela tabela abaixo que ligue todas as cidades e tenha um comprimento total mínimo.

	B	C	D	E	F
A	5	–	10	–	–
B		5	10	20	–
C			20	–	30
D				20	–
E					10

Exemplo de modelagem: Malha rodoviária (2)

- Tabela já é um modelo da situação do mundo real.
- A tabela pode ser transformada numa representação gráfica chamada GRAFO, que será o modelo matemático.



Grafo G

- Grafo (definição informal): conjunto de pontos chamados de vértices ou nós, e um conjunto de linhas (normalmente não-vazio) conectando um vértice ao outro.
  - Neste caso, cidades são representadas por vértices e estradas por linhas (arestas).

Exemplo de modelagem: Malha rodoviária (3)

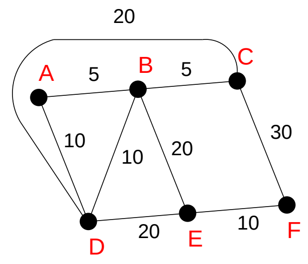
- Qual é o próximo passo?
  - Achar uma solução em termos desse modelo.
  - Nesse caso, achar um grafo G' com o mesmo número de vértices e um conjunto mínimo de arestas que conecte todas as cidades e satisfaça a condição do problema.
- Observação: o modelo matemático é escolhido, em geral, visando a solução.
- A solução será apresentada na forma de um algoritmo.

Exemplo de modelagem: Malha rodoviária (4)

Algoritmo:

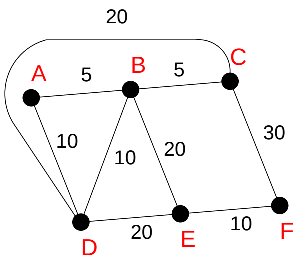
1. Selecione arbitrariamente qualquer vértice e o coloque no conjunto de vértices já conectados.
2. Escolha dentre os vértices não conectados aquele mais próximo de um vértice já conectado. Se existir mais de um vértice com essa característica escolha aleatoriamente qualquer um deles.
3. Repita o passo 2 até que todos os vértices já estejam conectados.

➜ Este é um exemplo de um "algoritmo guloso" (*greedy algorithm*).



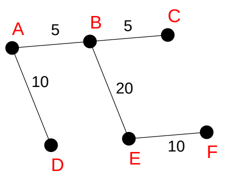
Grafo G

Exemplo de modelagem: Malha rodoviária – Soluções (5)

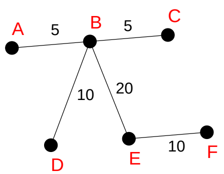


Grafo G

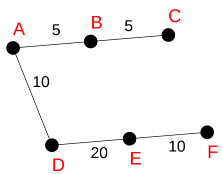
Soluções:



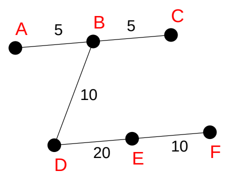
Grafo G_1



Grafo G_2



Grafo G_3



Grafo G_4

Exemplo de modelagem: Malha rodoviária (6)

- O que foi feito?
  1. Obtenção do modelo matemático para o problema.
  2. Formulação de um algoritmo em termos do modelo.

  ➜ Ou seja, essa é a técnica de resolução de problemas em Ciência da Computação.
- Nem todos os problemas considerados terão como solução um algoritmo, mas muitos terão.

Exemplo de modelagem: Sudoku e Godoku (1)

		1	8			3		
	4	9	7	1	6		8	
	2			9				
		4					2	
	5	6			1	8		
	1					5		9
						4	3	
			1	6				8
7					2			1

Sudoku

A								
				C				
	B				F	I		D
		A					B	
E				G				
		F				H		
			D		E			
I			B					
							A	

Godoku

O objetivo do Sudoku (Godoku) é preencher todos os espaços em branco do quadrado maior, que está dividido em nove grids, com os números de 1 a 9 (letras). Os algarismos não podem se repetir na mesma coluna, linha ou grid.

Sudoku: A palavra Sudoku significa "número sozinho" em japonês, o que mostra exatamente o objetivo do jogo. O Sudoku existe desde a década de 1970, mas começou a ganhar popularidade no final de 2004 quando começou a ser publicado diariamente na sessão de *puzzles* do jornal *The Times*. Entre abril e maio de 2005 o *puzzle* começou a ganhar um espaço na publicação de outros jornais britânicos e, poucos meses depois, ganhou popularidade mundial. Fonte: wikipedia.org

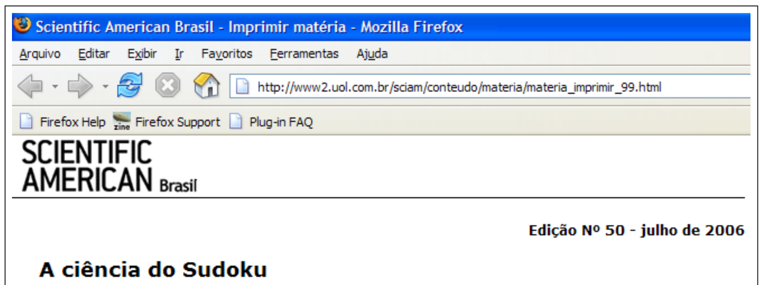
Godoku: O jogo Godoku é similar ao Sudoku mas formado apenas por letras.

Exemplo de modelagem: SuperSudoku (2)

6	1	B	2			5	3		A	D		8		C	F
		0	8		B	C			7	6	F				5
4	9		D	2			0	8			C	B		7	1
		F			D	8		3	B		1		A	0	E
3	4		5	A		1	7		F	2		C	0	9	
								6				A			B
0		E	B	D	6		9	1	4		A		5		
		6			4		5	9		8	7			E	3
				B				5	3			E	7		4
F		5				6		A		4	8			2	
	8		4				C	F	2			1	3		A
2		7				4		E			6			F	
		C	0	4			2			1	B			5	9
7	E			5		D		4	9		0	3		B	8
5		4		8		7	6		E	F	3		C		
8		1		3	0		B	C		5		7		4	D

O jogo SuperSudoku é similar ao Sudoku e Godoku formado por números e letras. Cada grid tem 16 entradas, sendo nove dos números (0 a 9) e seis letras (A a F).

Exemplo de modelagem: Mais informações sobre o Sudoku e jogos similares (3)



Para mais detalhes sobre o Sudoku e variantes desse jogo, veja o artigo "A ciência do Sudoku" por Jean-Paul Delahaye, na revista *Scientific American Brasil*, edição nº 50 de julho de 2006, ou nas páginas:

http://www2.uol.com.br/sciam/conteudo/materia/materia_99.html
http://www2.uol.com.br/sciam/conteudo/materia/materia_imprimir_99.html

Exemplo de modelagem: Kasparov × Deep Blue

In the first ever traditional chess match between a man (world champion Garry Kasparov) and a computer (IBM's Deep Blue) in 1996, Deep Blue won one game, tied two and lost three. The next year, Deep Blue defeated Kasparov in a six-game match – the first time a reigning world champion lost a match to a computer opponent in tournament play. Deep Blue was a combination of special purpose hardware and software with an IBM RS/6000 SP2 (seen here) – a system capable of examining 200 million moves per second, or 50 billion positions, in the three minutes allocated for a single move in a chess game.

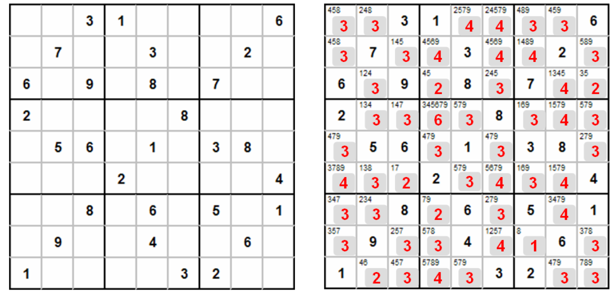
Referência: http://www-03.ibm.com/ibm/history/exhibits/vintage/vintage_4506VV1001.html

Questões sobre a modelagem (1)

- O objetivo é projetar um algoritmo para resolver o problema.
  - Veja que o Sudoku e o *Deep Blue* têm características bem diferentes!
- Esse projeto envolve dois aspectos:
  1. O algoritmo propriamente dito, e
  2. A estrutura de dados a ser usada nesse algoritmo.
- Em geral, a escolha do algoritmo influencia a estrutura de dados e vice-versa.
  - ➜ É necessário considerar diferentes fatores para escolher esse par (algoritmo e estrutura de dados).
- Nesta disciplina, estudaremos vários tópicos relacionados tanto a algoritmos quanto estruturas de dados.

Questões sobre a modelagem (2) O caso do jogo Sudoku

- Um possível algoritmo para resolver o jogo Sudoku é o "Algoritmo de Força Bruta":
  - Tente todas as possibilidades até encontrar uma solução!
- Nessa estratégia, quantas possibilidades existem para a configuração abaixo?



➜ Existem $1^1 \times 2^5 \times 3^{32} \times 4^{13} \times 6^1 =$
$23\,875\,983\,329\,839\,202\,653\,175\,808 \approx 23,8 \times 10^{24}$ possibilidades!

Estruturas de dados

- Estruturas de dados e algoritmos estão intimamente ligados:
  - Não se pode estudar estruturas de dados sem considerar os algoritmos associados a elas;
  - Assim como a escolha dos algoritmos em geral depende da representação e da estrutura dos dados.
- Para resolver um problema é necessário escolher uma abstração da realidade, em geral mediante a definição de um conjunto de dados que representa a situação real.
- A seguir, deve ser escolhida a forma de representar esses dados.

Escolha da representação dos dados

- A escolha da representação dos dados é determinada, entre outras, pelas operações a serem realizadas sobre os dados.
- Considere a operação de adição:
  - Para pequenos números, uma boa representação é por meio de barras verticais (caso em que a operação de adição é bastante simples).
  - Já a representação por dígitos decimais requer regras relativamente complicadas, as quais devem ser memorizadas.

➜ Quando consideramos a adição de grandes números é mais fácil a representação por dígitos decimais (devido ao princípio baseado no peso relativo a posição de cada dígito).

Programas

- Programar é basicamente estruturar dados e construir algoritmos.
- Programas são formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados.
- Programas representam uma classe especial de algoritmos capazes de serem seguidos por computadores.
- Um computador só é capaz de seguir programas em linguagem de máquina (seqüência de instruções obscuras e desconfortáveis).
- É necessário construir linguagens mais adequadas, que facilitem a tarefa de programar um computador.
- Uma linguagem de programação é uma técnica de notação para programar, com a intenção de servir de veículo tanto para a expressão do raciocínio algorítmico quanto para a execução automática de um algoritmo por um computador.

Tipos de dados

- Caracteriza o conjunto de valores a que uma constante pertence, ou que podem ser assumidos por uma variável ou expressão, ou que podem ser gerados por uma função.
- Tipos simples de dados são grupos de valores indivisíveis (como os tipos básicos *integer, boolean, char* e *real* do Pascal).
  - Exemplo: uma variável do tipo *boolean* pode assumir o valor **verdadeiro** ou o valor **falso**, e nenhum outro valor.
- Os tipos estruturados em geral definem uma coleção de valores simples, ou um agregado de valores de tipos diferentes.

Tipos abstratos de dados (TADs)

- Modelo matemático, acompanhado das operações definidas sobre o modelo.
  - Exemplo: o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação.
- TADs são utilizados extensivamente como base para o projeto de algoritmos.
- A implementação do algoritmo em uma linguagem de programação específica exige a representação do TAD em termos dos tipos de dados e dos operadores suportados.
- A representação do modelo matemático por trás do tipo abstrato de dados é realizada mediante uma estrutura de dados.
- Podemos considerar TADs como generalizações de tipos primitivos e procedimentos como generalizações de operações primitivas.
- O TAD encapsula tipos de dados:
  - A definição do tipo e todas as operações ficam localizadas numa seção do programa.

Implementação de TADs (1)

- Considere uma aplicação que utilize uma lista de inteiros. Poderíamos definir o TAD Lista, com as seguintes operações:
  1. Faça a lista vazia;
  2. Obtenha o primeiro elemento da lista; se a lista estiver vazia, então retorne nulo;
  3. Insira um elemento na lista.
- Há várias opções de estruturas de dados que permitem uma implementação eficiente para listas (por exemplo, o tipo estruturado arranjo).

Implementação de TADs (2)

- Cada operação do tipo abstrato de dados é implementada como um procedimento na linguagem de programação escolhida.
- Qualquer alteração na implementação do TAD fica restrita à parte encapsulada, sem causar impactos em outras partes do código.
- Cada conjunto diferente de operações define um TAD diferente, mesmo atuem sob um mesmo modelo matemático.
- A escolha adequada de uma implementação depende fortemente das operações a serem realizadas sobre o modelo.

Medida do tempo de execução de um programa

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos.
- Depois que um problema é analisado e decisões de projeto são finalizadas, é necessário estudar as várias opções de algoritmos a serem utilizados, considerando os aspectos de tempo de execução e espaço ocupado.
- Algoritmos são encontrados em todas as áreas de Ciência da Computação.

Tipos de problemas na análise de algoritmos Análise de um algoritmo particular

- Qual é o custo de usar um dado algoritmo para resolver um problema específico?
- Características que devem ser investigadas:
  - análise do número de vezes que cada parte do algoritmo deve ser executada,
  - estudo da quantidade de memória necessária.

Tipos de problemas na análise de algoritmos Análise de uma classe de algoritmos

- Qual é o algoritmo de menor custo possível para resolver um problema particular?
- Toda uma família de algoritmos é investigada.
- Procura-se identificar um que seja o melhor possível.
- Colocam-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe.

Custo de um algoritmo

- Determinando o menor custo possível para resolver problemas de uma dada classe, temos a medida da dificuldade inerente para resolver o problema.
- Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada.
- Podem existir vários algoritmos para resolver o mesmo problema.
- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado.

Medida do custo pela execução do programa em uma plataforma real

- Tais medidas são bastante inadequadas e os resultados jamais devem ser generalizados:
  - os resultados são dependentes do compilador que pode favorecer algumas construções em detrimento de outras;
  - os resultados dependem do hardware;
  - quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.
- Apesar disso, há argumentos a favor de se obterem medidas reais de tempo.
  - Por exemplo, quando há vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma mesma ordem de grandeza.
  - Assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.

Medida do custo por meio de um modelo matemático

- Usa um modelo matemático baseado em um computador idealizado.
- Deve ser especificado o conjunto de operações e seus custos de execuções.
- É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas.
- Por exemplo, algoritmos de ordenação:
  - consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam.

Função de complexidade

- Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade** f.
- $f(n)$ é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n.
- Função de **complexidade de tempo**: $f(n)$ mede o tempo necessário para executar um algoritmo para um problema de tamanho n.
- Função de **complexidade de espaço**: $f(n)$ mede a memória necessária para executar um algoritmo para um problema de tamanho n.
- Utilizaremos f para denotar uma função de complexidade de tempo daqui para a frente.
- Na realidade, a complexidade de tempo não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

Exemplo: Maior elemento (1)

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $A[1..n], n \geq 1$.

```
function Max (var A: Vetor): integer;
var i, Temp: integer;
begin
  Temp := A[1];
  for i:=2 to n do if Temp < A[i] then Temp := A[i];
  Max  := Temp;
end;
```

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de A, se A contiver n elementos.

- Logo $f(n) = n - 1$, para $n \geq 1$.

- Vamos provar que o algoritmo apresentado no programa acima é **ótimo**.

Exemplo: Maior elemento (2)

Teorema: Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n \geq 1$, faz pelo menos $n - 1$ comparações.

Prova: Cada um dos $n - 1$ elementos tem de ser mostrado, por meio de comparações, que é menor do que algum outro elemento.

Logo $n - 1$ comparações são necessárias. $\square$

➜ O teorema acima nos diz que, se o número de comparações for utilizado como medida de custo, então a função Max do programa anterior é ótima.

Tamanho da entrada de dados

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados.

- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada.

- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.

- No caso da função `Max` do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n.

- Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos.

Melhor caso, pior caso e caso médio (1)

- **Melhor caso**:
  - Menor tempo de execução sobre todas as entradas de tamanho n.

- **Pior caso**:
  - Maior tempo de execução sobre todas as entradas de tamanho n.
  - ➜ Se f é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que $f(n)$.

- **Caso médio** (ou caso esperado):
  - Média dos tempos de execução de todas as entradas de tamanho n.

Melhor caso, pior caso e caso médio (2)

- Na análise do caso esperado, supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho n e o custo médio é obtido com base nessa distribuição.
- A análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis.
- Na prática isso nem sempre é verdade.

Exemplo: Registros de um arquivo (1)

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recuperar registros do arquivo.
- O problema: dada uma chave qualquer, localize o registro que contenha esta chave.
- O algoritmo de pesquisa mais simples é o que faz a **pesquisa seqüencial**.
- Seja f uma função de complexidade tal que $f(n)$ é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - Melhor caso: $f(n) = 1$ (registro procurado é o primeiro consultado);
  - Pior caso: $f(n) = n$ (registro procurado é o último consultado ou não está presente no arquivo);
  - Caso médio: $f(n) = \frac{n+1}{2}$.

Exemplo: Registros de um arquivo (2)

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro.
- Se p_i for a probabilidade de que o i-ésimo registro seja procurado, e considerando que para recuperar o i-ésimo registro são necessárias i comparações, então
  $f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots + n \times p_n$.
- Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i.
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n, 1 \leq i \leq n$.
- Neste caso $f(n) = \frac{1}{n}(1 + 2 + 3 + \cdots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$.
- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

Exemplo: Maior e menor elementos (Versão 1)

- Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $A[1..n], n \geq 1$.
- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o maior elemento.

```
procedure MaxMin1 (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1]; Min := A[1];
  for i := 2 to n do
    begin
      if A[i] > Max then Max := A[i]; {Testa se A[i] contém o maior elemento}
      if A[i] < Min then Min := A[i]; {Testa se A[i] contém o menor elemento}
    end;
end;
```

- Seja $f(n)$ o número de comparações entre os elementos de A, se A tiver n elementos.
- Logo $f(n) = 2(n-1)$, para $n > 0$, para o melhor caso, pior caso e caso médio.

Exemplo: Maior e menor elementos (Versão 2)

- MaxMin1 pode ser facilmente melhorado:
  - a comparação $A[i] <$ Min só é necessária quando o resultado da comparação $A[i] >$ Max for falso.

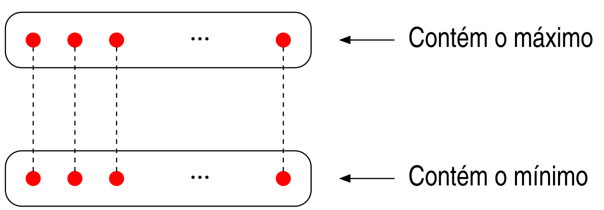
```
procedure MaxMin2 (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1];  Min := A[1];
  for i := 2 to n do
    if A[i] > Max
    then Max := A[i]
    else if A[i] < Min then Min := A[i];
end;
```


Exemplo: Maior e menor elementos (Versão 2)

- Para a nova implementação temos:
  - Melhor caso: $f(n) = n - 1$ (quando os elementos estão em ordem crescente);
  - Pior caso: $f(n) = 2(n - 1)$ (quando os elementos estão em ordem decrescente);
  - Caso médio: $f(n) = \frac{3n}{2} - \frac{3}{2}$.
- Caso médio:
  - $A[i]$ é maior do que Max a metade das vezes.
  - Logo, $f(n) = n - 1 + \frac{n-1}{2} = \frac{3n}{2} - \frac{3}{2}$, para $n > 0$.

Exemplo: Maior e menor elementos (Versão 3)

- Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente:
  1. Compare os elementos de A aos pares, separando-os em dois subconjuntos (maiores em um e menores em outro), a um custo de $\lceil n/2 \rceil$ comparações.
  2. O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.
  3. O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.



Exemplo: Maior e menor elementos (Versão 3)

```
procedure MaxMin3(var A: Vetor;
                  var Max, Min: integer);
var i,
    FimDoAnel: integer;
begin
  {Garante uma qte par de elementos no vetor para evitar caso de exceção}
  if (n mod 2) > 0
  then begin
         A[n+1]    := A[n];
         FimDoAnel := n;
       end
  else FimDoAnel := n-1;

  {Determina maior e menor elementos iniciais}
  if A[1] > A[2]
  then begin
         Max := A[1]; Min := A[2];
       end
  else begin
         Max := A[2]; Min := A[1];
       end;
```

Exemplo: Maior e menor elementos (Versão 3)

```
 i:= 3;
 while i <= FimDoAnel do
   begin
   {Compara os elementos aos pares}
   if A[i] > A[i+1]
   then begin
          if A[i] > Max then   Max := A[i];
          if A[i+1] < Min then Min := A[i+1];
        end
   else begin
          if A[i] < Min then   Min := A[i];
          if A[i+1] > Max then Max := A[i+1];
        end;
   i:= i + 2;
   end;
end;
```


Exemplo: Maior e menor elementos (Versão 3)

- Os elementos de A são comparados dois a dois e os elementos maiores são comparados com `Max` e os elementos menores são comparados com `Min`.
- Quando n é ímpar, o elemento que está na posição $A[n]$ é duplicado na posição $A[n+1]$ para evitar um tratamento de exceção.
- Para esta implementação, $f(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$, para $n > 0$, para o melhor caso, pior caso e caso médio.

Comparação entre os algoritmos MaxMin1, MaxMin2 e MaxMin3

- A tabela abaixo apresenta uma comparação entre os algoritmos dos programas MaxMin1, MaxMin2 e MaxMin3, considerando o número de comparações como medida de complexidade.
- Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1 de forma geral.
- O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e bastante próximo quanto ao caso médio.

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n-1)$	$2(n-1)$	$2(n-1)$
MaxMin2	$n-1$	$2(n-1)$	$3n/2-3/2$
MaxMin3	$3n/2-2$	$3n/2-2$	$3n/2-2$

Limite inferior: Uso de um oráculo

- É possível obter um algoritmo MaxMin mais eficiente?
- Para responder temos de conhecer o **limite inferior** para essa classe de algoritmos.
- Técnica muito utilizada:
  ➜ Uso de um oráculo.
- Dado um modelo computacional que expresse o comportamento do algoritmo, o oráculo informa o resultado de cada passo possível (no caso, o resultado de cada comparação).
- Para derivar o limite inferior, o oráculo procura sempre fazer com que o algoritmo trabalhe o máximo, escolhendo como resultado da próxima comparação aquele que cause o maior trabalho possível necessário para determinar a resposta final.

Exemplo de uso de um oráculo (1)

Teorema: Qualquer algoritmo para encontrar o maior e o menor elementos de um conjunto com n elementos não ordenados, $n \geq 1$, faz pelo menos $\lceil 3n/2 \rceil - 2$ comparações.

Prova: A técnica utilizada define um oráculo que descreve o comportamento do algoritmo por meio de um conjunto de n–tuplas, mais um conjunto de regras associadas que mostram as tuplas possíveis (estados) que um algoritmo pode assumir a partir de uma dada tupla e uma única comparação.

Uma 4–tupla, representada por (a, b, c, d), onde os elementos de:

- a nunca foram comparados;
- b foram vencedores e nunca perderam em comparações realizadas;
- c foram perdedores e nunca venceram em comparações realizadas;
- d foram vencedores e perdedores em comparações realizadas.

Exemplo de uso de um oráculo (2)

- O algoritmo inicia no estado $(n, 0, 0, 0)$ e termina com $(0, 1, 1, n - 2)$.

$$(n, 0, 0, 0) \rightarrow \vdots \rightarrow (0, 1, 1, n-2)$$

Exemplo de uso de um oráculo (3)

- Após cada comparação a tupla (a, b, c, d) consegue progredir apenas se ela assume um dentre os seis estados possíveis abaixo:
  1. $(a - 2, b + 1, c + 1, d)$ se $a \geq 2$
     ➜ Dois elementos de a são comparados.
  2. $(a - 1, b + 1, c, d)$ ou
  3. $(a - 1, b, c + 1, d)$ ou
  4. $(a - 1, b, c, d + 1)$ se $a \geq 1$
     ➜ Um elemento de a é comparado com um de b ou um de c.
  5. $(a, b - 1, c, d + 1)$ se $b \geq 2$
     ➜ Dois elementos de b são comparados.
  6. $(a, b, c - 1, d + 1)$ se $c \geq 2$
     ➜ Dois elementos de c são comparados.
- O primeiro passo requer necessariamente a manipulação do componente a.

Exemplo de uso de um oráculo (4)

- O caminho mais rápido para levar a até zero requer $\lceil n/2 \rceil$ mudanças de estado e termina com a tupla $(0, n/2, n/2, 0)$ (por meio de comparação dos elementos de a dois a dois).
- A seguir, para reduzir o componente b até um são necessárias $\lceil n/2 \rceil - 1$ mudanças de estado (mínimo de comparações necessárias para obter o maior elemento de b).
- Idem para c, com $\lceil n/2 \rceil - 1$ mudanças de estado.
- Logo, para obter o estado $(0, 1, 1, n - 2)$ a partir do estado $(n, 0, 0, 0)$ são necessárias

$$\lceil n/2 \rceil + \lceil n/2 \rceil - 1 + \lceil n/2 \rceil - 1 = \lceil 3n/2 \rceil - 2$$

comparações. □

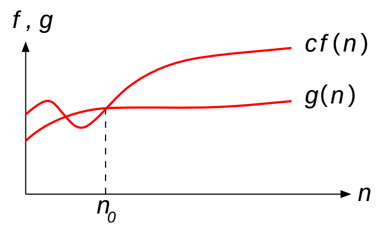
- O teorema nos diz que se o número de comparações entre os elementos de um vetor for utilizado como medida de custo, então o algoritmo MaxMin3 é **ótimo**.

Comportamento assintótico de funções

- O parâmetro n fornece uma medida da dificuldade para se resolver o problema.
- Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
- A **escolha do algoritmo** não é um problema crítico para problemas de tamanho pequeno.
- Logo, a análise de algoritmos é realizada para valores grandes de n.
- Estuda-se o comportamento assintótico das **funções de custo** (comportamento de suas funções de custo para valores grandes de n).
- O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo quando n cresce.

Dominação assintótica

- A análise de um algoritmo geralmente conta com apenas algumas operações elementares.
- A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada.
- **Definição**: Uma função $f(n)$ **domina assintoticamente** outra função $g(n)$ se existem duas constantes positivas c e n_0 tais que, para $n \geq n_0$, temos $|g(n)| \leq c \times |f(n)|$.



Exemplo:

- Sejam $g(n) = (n+1)^2$ e $f(n) = n^2$.
- As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra, já que

$$|(n+1)^2| \leq 4|n^2|$$

para $n \geq 1$ e

$$|n^2| \leq |(n+1)^2|$$

para $n \geq 0$.

Como medir o custo de execução de um algoritmo?

- **Função de Custo** ou **Função de Complexidade**
  - $f(n) =$ medida de custo necessário para executar um algoritmo para um problema de tamanho n.
  - Se $f(n)$ é uma medida da quantidade de tempo necessário para executar um algoritmo para um problema de tamanho n, então f é chamada *função de complexidade de tempo de algoritmo*.
  - Se $f(n)$ é uma medida da quantidade de memória necessária para executar um algoritmo para um problema de tamanho n, então f é chamada *função de complexidade de espaço de algoritmo*.
- Observação: tempo não é tempo!
  - É importante ressaltar que a complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada

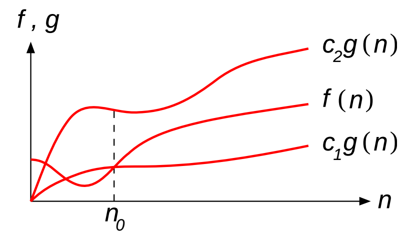
Custo assintótico de funções

- É interessante comparar algoritmos para valores grandes de n.
- O *custo assintótico* de uma função $f(n)$ representa o limite do comportamento de custo quando n cresce.
- Em geral, o custo aumenta com o tamanho n do problema.
- Observação:
  - Para valores pequenos de n, mesmo um algoritmo ineficiente não custa muito para ser executado.

Notação assintótica de funções

- Existem três notações principais na análise assintótica de funções:
  - Notação Θ
  - Notação O (“O” grande)
  - Notação Ω

Notação Θ (1)



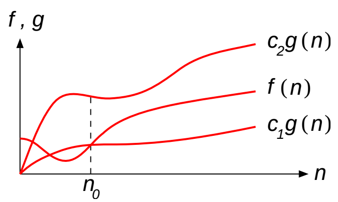
$$f(n) = \Theta(g(n))$$

Notação Θ (2)

- A notação Θ limita a função por fatores constantes.
- Escreve-se $f(n) = \Theta(g(n))$, se existirem constantes positivas c_1, c_2 e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ está sempre entre $c_1 g(n)$ e $c_2 g(n)$ inclusive.
- Neste caso, pode-se dizer que $g(n)$ é um limite assintótico firme (em inglês, *asymptotically tight bound*) para $f(n)$.

$$f(n) = \Theta(g(n)),\ \exists\, c_1 > 0, c_2 > 0 \text{ e } n_0 \mid$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n),\ \forall\, n \geq n_0$$



Notação Θ: Exemplo (3)

- Mostre que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Para provar esta afirmação, devemos achar constantes $c_1 > 0, c_2 > 0, n > 0$, tais que:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

para todo $n \geq n_o$.

Se dividirmos a expressão acima por n^2 temos:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Notação Θ: Exemplo (4)

A inequação mais a direita será sempre válida para qualquer valor de $n \geq 1$ ao escolhermos $c_2 \geq \frac{1}{2}$.

Da mesma forma, a inequação mais a esquerda será sempre válida para qualquer valor de $n \geq 7$ ao escolhermos $c_1 \geq \frac{1}{14}$.

Assim, ao escolhermos $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, podemos verificar que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Note que existem outras escolhas para as constantes c_1 e c_2, mas o fato importante é que *a escolha existe*.

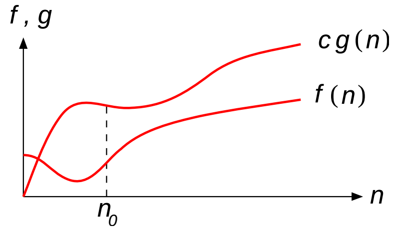
Note também que a escolha destas constantes depende da função $\frac{1}{2}n^2 - 3n$.

Uma função diferente pertencente a $\Theta(n^2)$ irá provavelmente requerer outras constantes.

Notação Θ: Exemplo (5)

- Usando a definição formal de Θ prove que $6n^3 \neq \Theta(n^2)$.

Notação O (1)



$$f(n) = O(g(n))$$

Notação O (2)

- A notação O define um limite superior para a função, por um fator constante.
- Escreve-se $f(n) = O(g(n))$, se existirem constantes positivas c e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ é menor ou igual a $cg(n)$. Neste caso, pode-se dizer que $g(n)$ é um limite assintótico superior (em inglês, *asymptotically upper bound*) para $f(n)$.

$$f(n) = O(g(n)),\ \exists\, c > 0 \text{ e } n_0 \mid 0 \leq f(n) \leq cg(n),\ \forall\, n \geq n_0$$

- Escrevemos $f(n) = O(g(n))$ para expressar que $g(n)$ domina assintoticamente $f(n)$. Lê-se $f(n)$ é da ordem no máximo $g(n)$.

Notação O: Exemplos (3)

- Seja $f(n) = (n + 1)^2$.
  - Logo $f(n)$ é $O(n^2)$, quando $n_0 = 1$ e $c = 4$, já que
$$(n + 1)^2 \leq 4n^2 \text{ para } n \geq 1.$$
- Seja $f(n) = n$ e $g(n) = n^2$. Mostre que $g(n)$ não é $O(n)$.
  - Sabemos que $f(n)$ é $O(n^2)$, pois para $n \geq 0$, $n \leq n^2$.
  - Suponha que existam constantes c e n_0 tais que para todo $n \geq n_0$, $n^2 \leq cn$. Assim, $c \geq n$ para qualquer $n \geq n_0$. No entanto, não existe uma constante c que possa ser maior ou igual a n para todo n.

Notação O: Exemplos (4)

- Mostre que $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$.
  - Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$, para $n \geq 0$.
  - A função $g(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto esta afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$.
- Mostre que $h(n) = \log_5 n$ é $O(\log n)$.
  - O $\log_b n$ difere do $\log_c n$ por uma constante que no caso é $\log_b c$.
  - Como $n = c^{\log_c n}$, tomando o logaritmo base b em ambos os lados da igualdade, temos que $\log_b n = \log_b c^{\log_c n} = \log_c n \times \log_b c$.

Notação O (5)

- Quando a notação O é usada para expressar o tempo de execução de um algoritmo no pior caso, está se definindo também o limite (superior) do tempo de execução desse algoritmo para *todas* as entradas.
- Por exemplo, o algoritmo de ordenação por inserção (a ser estudado neste curso) é $O(n^2)$ no pior caso.
  - ➜ Este limite se *aplica* para *qualquer* entrada.

Notação O (6)

- Tecnicamente é um abuso dizer que o tempo de execução do algoritmo de ordenação por inserção *é* $O(n^2)$ (i.e., sem especificar se é para o pior caso, melhor caso, ou caso médio)
  - ➜ O tempo de execução desse algoritmo depende de como os dados de entrada estão arranjados.
  - ➜ Se os dados de entrada já estiverem ordenados, este algoritmo tem um tempo de execução de $O(n)$, ou seja, o tempo de execução do algoritmo de ordenação por inserção no *melhor caso* é $O(n)$.
- O que se quer dizer quando se fala que "o tempo de execução" é $O(n^2)$" é que no pior caso o tempo de execução é $O(n^2)$.
  - ➜ Ou seja, não importa como os dados de entrada estão arranjados, o tempo de execução em qualquer entrada é $O(n^2)$.

Operações com a notação O (7)

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \times O(f(n)) &= O(f(n)) \quad c = constante \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n)) &= O(f(n)) \\
O(f(n)) + O(g(n)) &= O(max(f(n), g(n))) \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \\
f(n)O(g(n)) &= O(f(n)g(n))
\end{aligned}
$$

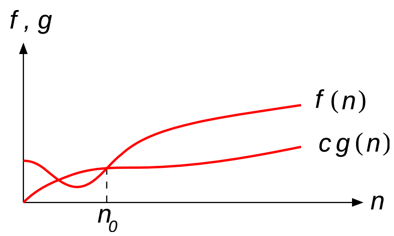
Operações com a notação O: Exemplos (8)

- Regra da soma $O(f(n)) + O(g(n))$.
  - Suponha três trechos cujos tempos de execução são
    $O(n)$, $O(n^2)$ e $O(n \log n)$.
  - O tempo de execução dos dois primeiros trechos é $O(max(n, n^2))$, que é $O(n^2)$.
  - O tempo de execução de todos os três trechos é então
    $O(max(n^2, n \log n))$,
    que é $O(n^2)$.

- O produto de $[\log n + k + O(1/n)]$ por $[n + O(\sqrt{n})]$ é

$$n \log n + kn + O(\sqrt{n} \log n).$$

Notação Ω (1)



$$f(n) = \Omega(g(n))$$

Notação Ω (2)

- A notação Ω define um limite inferior para a função, por um fator constante.

- Escreve-se $f(n) = \Omega(g(n))$, se existirem constantes positivas c e n_0 tais que para $n \geq n_0$, o valor de $f(n)$ é maior ou igual a $cg(n)$.
  - ➜ Pode-se dizer que $g(n)$ é um limite assintótico inferior (em inglês, *asymptotically lower bound*) para $f(n)$.

$$f(n) \;=\; \Omega(g(n)),\ \exists\, c > 0 \text{ e } n_0 \mid 0 \leq cg(n) \leq f(n),\ \forall\, n \geq n_0$$

Notação Ω (3)

- Quando a notação Ω é usada para expressar o tempo de execução de um algoritmo no melhor caso, está se definindo também o limite (inferior) do tempo de execução desse algoritmo para *todas* as entradas.

- Por exemplo, o algoritmo de ordenação por inserção é $\Omega(n)$ no melhor caso.
  ➜ O tempo de execução do algoritmo de ordenação por inserção é $\Omega(n)$.

- O que significa dizer que "o tempo de execução" (i.e., sem especificar se é para o pior caso, melhor caso, ou caso médio) é $\Omega(g(n))$?
  - O tempo de execução desse algoritmo é pelo menos uma constante vezes $g(n)$ para valores suficientemente grandes de n.

Notação Ω: Exemplos (4)

- Para mostrar que $f(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer $c = 1$, e então $3n^3 + 2n^2 \geq n^3$ para $n \geq 0$.

- Seja $f(n) = n$ para n ímpar $(n \geq 1)$ e $f(n) = n^2/10$ para n par $(n \geq 0)$.
  - Neste caso $f(n)$ é $\Omega(n^2)$, bastando considerar $c = 1/10$ e $n = 0, 2, 4, 6, \ldots$

Limites do algoritmo de ordenação por inserção

- O tempo de execução do algoritmo de ordenação por inserção está entre $\Omega(n)$ e $O(n^2)$.

- Estes limites são assintoticamente os mais firmes possíveis.
  - Por exemplo, o tempo de execução deste algoritmo *não é* $\Omega(n^2)$, pois o algoritmo executa em tempo $\Theta(n)$ quando a entrada já está ordenada.

- Não é contraditório dizer que o tempo de execução deste algoritmo no *pior caso* é $\Omega(n^2)$, já que existem entradas para este algoritmo que fazem com que ele execute em tempo $\Omega(n^2)$.

Teorema

Para quaisquer funções $f(n)$ e $g(n)$,

$$f(n) = \Theta(g(n))$$

se e somente se,

$$f(n) = O(g(n)), \text{ e}$$

$$f(n) = \Omega(g(n))$$

Mais sobre notação assintótica de funções

- Existem duas outras notações na análise assintótica de funções:
  - Notação o ("O" pequeno)
  - Notação ω
- Estas duas notações não são usadas normalmente, mas é importante saber seus conceitos e diferenças em relação às notações O e Ω, respectivamente.

Notação o (1)

- O limite assintótico superior definido pela notação O pode ser assintoticamente firme ou não.
  - Por exemplo, o limite $2n^2 = O(n^2)$ é assintoticamente firme, mas o limite $2n = O(n^2)$ não é.
- A notação o é usada para definir um limite superior que não é assintoticamente firme.
- Formalmente a notação o é definida como:

$$f(n) = o(g(n)), \text{ para qq } c > 0 \text{ e } n_0 \mid 0 \leq f(n) < cg(n), \; \forall\, n \geq n_0$$

- Exemplo, $2n = o(n^2)$ mas $2n^2 \neq o(n^2)$.

Notação o (2)

- As definições das notações O (o grande) e o (o pequeno) são similares.
  - A diferença principal é que em $f(n) = O(g(n))$, a expressão $0 \leq f(n) \leq cg(n)$ é válida para todas constantes $c > 0$.
- Intuitivamente, a função $f(n)$ tem um crescimento muito menor que $g(n)$ quando n tende para infinito. Isto pode ser expresso da seguinte forma:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

➜ Alguns autores usam este limite como a definição de o.

Notação ω

- Por analogia, a notação ω está relacionada com a notação Ω da mesma forma que a notação o está relacionada com a notação O.
- Formalmente a notação ω é definida como:

$$f(n) = \omega(g(n)), \text{ para qq } c > 0 \text{ e } n_0 \mid 0 \leq cg(n) < f(n), \; \forall\, n \geq n_0$$

- Por exemplo, $\frac{n^2}{2} = \omega(n)$, mas $\frac{n^2}{2} \neq \omega(n^2)$.
- A relação $f(n) = \omega(g(n))$ implica em

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty,$$

se o limite existir.

Comparação de programas

- Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade.
- Um programa com tempo de execução $O(n)$ é melhor que outro com tempo $O(n^2)$.
  - Porém, as constantes de proporcionalidade podem alterar esta consideração.
- Exemplo: um programa leva $100n$ unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?
  - Depende do tamanho do problema.
  - Para $n < 50$, o programa com tempo $2n^2$ é melhor do que o que possui tempo $100n$.
  - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$.
  - Entretanto, quando n cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa $O(n)$.

Classes de Comportamento Assintótico
Complexidade Constante

- $f(n) = O(1)$
  - O uso do algoritmo independe do tamanho de n.
  - As instruções do algoritmo são executadas um número fixo de vezes.

➜ O que significa um algoritmo ser $O(2)$ ou $O(5)$?

Classes de Comportamento Assintótico
Complexidade Logarítmica

- $f(n) = O(\log n)$
  - Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores.
  - Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande.
- Supondo que a base do logaritmo seja 2:
  - Para $n = 1\,000$, $\log_2 \approx 10$.
  - Para $n = 1\,000\,000$, $\log_2 \approx 20$.
- Exemplo:
  - Algoritmo de pesquisa binária.

Classes de Comportamento Assintótico
Complexidade Linear

- $f(n) = O(n)$
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.
  - Esta é a melhor situação possível para um algoritmo que tem que processar/produzir n elementos de entrada/saída.
  - Cada vez que n dobra de tamanho, o tempo de execução também dobra.
- Exemplos:
  - Algoritmo de pesquisa seqüencial.
  - Algoritmo para teste de planaridade de um grafo.

Classes de Comportamento Assintótico
Complexidade Linear Logarítmica

- $f(n) = O(n \log n)$
  - Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois agrupando as soluções.
  - Caso típico dos algoritmos baseados no paradigma *divisão-e-conquista*.
- Supondo que a base do logaritmo seja 2:
  - Para $n = 1\,000\,000$, $\log_2 \approx 20\,000\,000$.
  - Para $n = 2\,000\,000$, $\log_2 \approx 42\,000\,000$.
- Exemplo:
  - Algoritmo de ordenação MergeSort.

Classes de Comportamento Assintótico
Complexidade Quadrática

- $f(n) = O(n^2)$
  - Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro do outro
  - Para $n = 1\,000$, o número de operações é da ordem de $1\,000\,000$.
  - Sempre que n dobra o tempo de execução é multiplicado por 4.
  - Algoritmos deste tipo são úteis para resolver problemas de tamanhos *relativamente* pequenos.
- Exemplos:
  - Algoritmos de ordenação simples como seleção e inserção.

Classes de Comportamento Assintótico
Complexidade Cúbica

- $f(n) = O(n^3)$
  - Algoritmos desta ordem de complexidade geralmente são úteis apenas para resolver problemas *relativamente* pequenos.
  - Para $n = 100$, o número de operações é da ordem de $1\,000\,000$
  - Sempre que n dobra o tempo de execução é multiplicado por 8.
  - Algoritmos deste tipo são úteis para resolver problemas de tamanhos *relativamente* pequenos.
- Exemplo:
  - Algoritmo para multiplicação de matrizes.

Classes de Comportamento Assintótico
Complexidade Exponencial

- $f(n) = O(2^n)$
  - Algoritmos desta ordem de complexidade não são úteis sob o ponto de vista prático.
  - Eles ocorrem na solução de problemas quando se usa a *força bruta* para resolvê-los.
  - Para $n = 20$, o tempo de execução é cerca de $1\,000\,000$.
  - Sempre que n dobra o tempo de execução fica elevado ao quadrado.
- Exemplo:
  - Algoritmo do Caixeiro Viajante

Classes de Comportamento Assintótico
Complexidade Exponencial

- $f(n) = O(n!)$.
  - Um algoritmo de complexidade $O(n!)$ é dito ter complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$.
  - Geralmente ocorrem quando se usa *força bruta* na solução do problema.
- Considerando:
  - $n = 20$, temos que $20! = 2432902008176640000$, um número com 19 dígitos.
  - $n = 40$ temos um número com 48 dígitos.

Comparação de funções de complexidade

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc
3^n	0,059 s	58 min	6,5 anos	3855 séc	10^8 séc	10^{13} séc

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1000 vezes mais rápido
n	t_1	$100\ t_1$	$1000\ t_1$
n^2	t_2	$10\ t_2$	$31,6\ t_2$
n^3	t_3	$4,6\ t_3$	$10\ t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

Hierarquias de funções (1)

A seguinte hierarquia de funções pode ser definida do ponto de vista assintótico:

$$1 \prec \log\log n \prec \log n \prec n^\epsilon \prec n^c \prec n^{\log n} \prec c^n \prec n^n \prec c^{c^n}$$

onde ϵ e c são constantes arbitrárias com $0 < \epsilon < 1 < c$.

Hierarquias de funções (2)

- Usando MatLab, ou um outro pacote matemático, desenhe os gráficos dessas funções, quando $n \to \infty$

Hierarquias de funções (3)

Onde as seguintes funções se encaixam nessa hierarquia? (Mostre a sua solução)

(a) $\pi(n) = \frac{n}{\ln n}$. Esta função define o número de primos menor ou igual a n.

(b) $e^{\sqrt{\log n}}$.
Dica: $e^{f(n)} \prec e^{g(n)} \iff \lim_{n\to\infty}(f(n) - g(n)) = -\infty$

Hierarquias de Funções (4)
Preliminares

A hierarquia apresentada está relacionada com funções que vão para o infinito. No entanto, podemos ter o recíproco dessas funções já que elas nunca são zero. Isto é,

$$f(n) \prec g(n) \iff \frac{1}{g(n)} \prec \frac{1}{f(n)}.$$

Assim, todas as funções (exceto 1) tendem para zero:

$$\frac{1}{c^{c^n}} \prec \frac{1}{n^n} \prec \frac{1}{c^n} \prec \frac{1}{n^{\log n}} \prec \frac{1}{n^c} \prec \frac{1}{n^\epsilon} \prec \frac{1}{\log n} \prec \frac{1}{\log\log n} \prec 1$$

Hierarquias de Funções (5)
Solução de (a)

- $\pi(n) = \frac{n}{\ln n}$

Temos que (note que a base do logaritmo não altera a hierarquia):

$$\frac{1}{n^\epsilon} \prec \frac{1}{\ln n} \prec 1$$

Multiplicando por n, temos:

$$\frac{n}{n^\epsilon} \prec \frac{n}{\ln n} \prec n,$$

ou seja,

$$n^{1-\epsilon} \prec \pi(n) \prec n$$

Note que o valor $1 - \epsilon$ ainda é menor que 1.

Hierarquias de Funções (6)
Solução de (b)

- $e^{\sqrt{\log n}}$

Dado a hierarquia:

$$1 \prec \ln\ln n \prec \sqrt{\ln n} \prec \epsilon \ln n$$

e elevando a e, temos que:

$$e^1 \prec e^{\ln\ln n} \prec e^{\sqrt{\ln n}} \prec e^{\epsilon \ln n}$$

Simplificando temos:

$$e \prec \ln n \prec e^{\sqrt{\ln n}} \prec n^\epsilon$$

Algoritmo exponencial × Algoritmo polinomial (1)

- Funções de complexidade:
  - Um algoritmo cuja função de complexidade é $O(c^n), c > 1$, é chamado de *algoritmo exponencial* no tempo de execução.
  - Um algoritmo cuja função de complexidade é $O(p(n))$, onde $p(n)$ é um polinômio de grau n, é chamado de *algoritmo polinomial* no tempo de execução.
- A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce.
- Esta é a razão porque algoritmos polinomiais são muito mais úteis na prática do que algoritmos exponenciais.
  - Geralmente, algoritmos exponenciais são simples variações de pesquisa exaustiva.

Algoritmo exponencial × Algoritmo polinomial (2)

- Os algoritmos polinomiais são geralmente obtidos através de um entendimento mais profundo da estrutura do problema.
- Tratabilidade dos problemas:
  - Um problema é considerado <u>intratável</u> se ele é tão difícil que não se conhece um algoritmo polinomial para resolvê-lo.
  - Um problema é considerado <u>tratável</u> (bem resolvido) se existe um algoritmo polinomial para resolvê-lo.

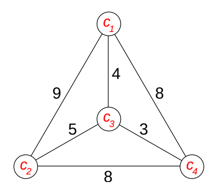
Aspecto importante no projeto de algoritmos.

Algoritmo exponencial × Algoritmo polinomial (3)

- A distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções.
- Exemplo: um algoritmo com função de complexidade $f(n) = 2^n$ é mais rápido que um algoritmo $g(n) = n^5$ para valores de n menores ou iguais a 20.
- Também existem algoritmos exponenciais que são muito úteis na prática.
  - Exemplo: o algoritmo Simplex para programação linear possui complexidade de tempo exponencial para o pior caso mas executa muito rápido na prática.
- Tais exemplos não ocorrem com freqüência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis.

Algoritmo exponencial
O Problema do Caixeiro Viajante

- Um **caixeiro viajante** deseja visitar n cidades de tal forma que sua viagem inicie e termine em uma mesma cidade, e cada cidade deve ser visitada uma única vez.
- Supondo que sempre há uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota para a viagem.
- Seja a figura que ilustra o exemplo para quatro cidades c_1, c_2, c_3, c_4, em que os números nas arestas indicam a distância entre duas cidades.



O percurso $\langle c_1, c_3, c_4, c_2, c_1\rangle$ é uma solução para o problema, cujo percurso total tem distância 24.

Exemplo de algoritmo exponencial

- Um algoritmo simples seria verificar todas as rotas e escolher a menor delas.
- Há $(n-1)!$ rotas possíveis e a distância total percorrida em cada rota envolve n adições, logo o número total de adições é $n!$.
- No exemplo anterior teríamos 24 adições.
- Suponha agora 50 cidades: o número de adições seria $50! \approx 10^{64}$.
- Em um computador que executa 10^9 adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que 10^{45} séculos só para executar as adições.
- O problema do caixeiro viajante aparece com freqüência em problemas relacionados com transporte, mas também aplicações importantes relacionadas com otimização de caminho percorrido.

Técnicas de análise de algoritmos

- Determinar o tempo de execução de um programa pode ser um problema matemático complexo.
- Determinar a ordem do tempo de execução, sem preocupação com o valor da constante envolvida, pode ser uma tarefa mais simples.
- A análise utiliza técnicas de matemática discreta, envolvendo contagem ou enumeração dos elementos de um conjunto:
  - manipulação de somas;
  - produtos;
  - permutações;
  - fatoriais;
  - coeficientes binomiais;
  - solução de **equações de recorrência**.

Análise do tempo de execução

- Comando de atribuição, de leitura ou de escrita: $O(1)$.
- Seqüência de comandos: determinado pelo maior tempo de execução de qualquer comando da seqüência.
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é $O(1)$.
- Anel: soma do tempo de execução do corpo do anel mais o tempo de avaliar a condição para terminação (geralmente $O(1)$), multiplicado pelo número de iterações.

Análise do tempo de execução

- **Procedimentos não recursivos**:
  - Cada um deve ser computado separadamente um a um, iniciando com os que não chamam outros procedimentos.
  - Avalia-se então os que são chamam os já avaliados (utilizando os tempos desses).
  - O processo é repetido até chegar no programa principal.
- **Procedimentos recursivos**:
  - É associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos.

Procedimento não recursivo

Algoritmo para ordenar os n elementos de um conjunto A em ordem ascendente.

```
    procedure Ordena (var A: Vetor);
    var  i, j, min, x: integer;
    begin
(1)   for i := 1 to n-1 do
        begin
          {min contém o índice do
           menor elemento de A[i..n]}
(2)       min := i;
(3)       for j := i+1 to n do
(4)         if A[j] < A[min]
(5)         then min := j;

          {Troca A[min] e A[i]}
(6)       x      := A[min];
(7)       A[min] := A[i];
(8)       A[i]   := x;
        end;
    end;
```

- Seleciona o menor elemento do conjunto.
- Troca este elemento com $A[1]$.
- Repete as duas operações acima com os $n-1$ elementos restantes, depois com os $n-2$, até que reste apenas um.

Análise do procedimento não recursivo Anel interno

- Contém um comando de decisão, com um comando apenas de atribuição. Ambos levam tempo constante para serem executados.
- Quanto ao corpo do comando de decisão, devemos considerar o pior caso, assumindo que será sempre executado.
- O tempo para incrementar o índice do anel e avaliar sua condição de terminação é $O(1)$.
- O tempo combinado para executar uma vez o anel é $O(max(1,1,1)) = O(1)$, conforme regra da soma para a notação O.
- Como o número de iterações é $n-i$, o tempo gasto no anel é $O((n-i)\times 1) = O(n-i)$, conforme regra do produto para a notação O.

Análise do procedimento não recursivo Anel externo

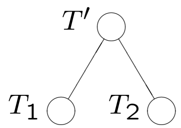
- Contém, além do anel interno, quatro comandos de atribuição:
  - $O(max(1,(n-i),1,1,1)) = O(n-i)$.
- A linha (1) é executada $n-1$ vezes, e o tempo total para executar o programa está limitado ao produto de uma constante pelo **somatório** de $(n-i)$:
  - $\sum_1^{n-1}(n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$
- Considerarmos o número de comparações como a medida de custo relevante, o programa faz $(n^2)/2 - n/2$ comparações para ordenar n elementos.
- Considerarmos o número de trocas, o programa realiza exatamente $n-1$ trocas.

Algoritmos recursivos

- Um objeto é recursivo quando é definido parcialmente em termos de si mesmo
- Exemplo 1: Números naturais
  (a) 1 é um número natural
  (b) o sucessor de um número natural é um número natural
- Exemplo 2: Função fatorial
  (a) $0! = 1$
  (b) se $n > 0$ então $n! = n \cdot (n-1)!$

Algoritmos recursivos

- Exemplo 3: Árvores
  (a) A árvore vazia é uma árvore
  (b) se T_1 e T_2 são árvores então T' é um árvore



Poder da recursão

- Definir um conjunto infinito de objetos através de um comando finito
- Um problema recursivo P pode ser expresso como $P \equiv \mathcal{P}[S_i, P]$, onde $\mathcal{P}$ é a composição de comandos S_i e do próprio P
- Importante: constantes e variáveis locais a P são duplicadas a cada chamada recursiva

Problema de terminação

- Definir um condição de terminação
- Idéia:
  - Associar um parâmetro, por exemplo n, com P e chamar P recursivamente com $n - 1$ como parâmetro
  - A condição $n > 0$ garante a terminação
  - Exemplo:

$$P(n) \equiv \text{if } n > 0 \text{ then } \mathcal{P}[S_i; P(n-1)]$$

- Importante: na prática é necessário:
  - mostrar que o nível de recursão é finito, e
  - tem que ser mantido pequeno! Por que?

Razões para limitar a recursão

- Memória necessária para acomodar variáveis a cada chamada
- O estado corrente da computação tem que ser armazenado para permitir a volta da chamada recursiva.

Exemplo:

```
function F(i : integer) : integer;
begin
  if i > 0
  then F := i * F(i-1)
  else F := 1;
end;
```

F(4) →	1	4∗F(3)
	2	3∗F(2)
	3	2∗F(1)
	4	1∗F(0)
	1	

Quando não usar recursividade

- Algoritmos recursivos são apropriados quando o problema é definido em termos recursivos
- Entretanto, uma definição recursiva não implica necessariamente que a implementação recursiva é a melhor solução!
- Casos onde evitar recursividade:
  - P $\equiv$ if condição then $(S_i; P)$
    Exemplo: P $\equiv$ **if** $i < n$ **then** $(\overbrace{i := i + 1; F := i * F}; P)$

Eliminando a recursividade de cauda *(Tail recursion)*

```
function Fat : integer;
var F, i : integer;
begin
  i := 0; F := 1;
  while i < n do
  begin
    i := i+1;
    F := F*i;
  end;
  Fat := F;
end
```

Logo,

$$P \equiv \texttt{if } B \texttt{ then } (S; P)$$

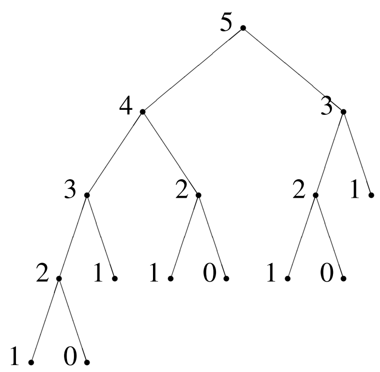
deve ser transformado em

$$P \equiv (x = x_0; \texttt{while } B \texttt{ do } S)$$

Outro exemplo

```
function Fib(n : integer) : integer;
begin
  if n = 0
  then Fib := 0
  else if n = 1
       then Fib := 1
       else Fib := Fib(n-1) + Fib(n-2);
end;  {Fib}
```

Observação: para cada chamada a Fib(n), Fib é ativada 2 vezes



Solução óbvia

```
function Fib : integer;
var i, Temp, F, Fant : integer;
begin
  i := 1;  F := 1;  Fant := 0;
  while i < n do
  begin
    Temp := F;
    F := F + Fant;
    Fant := Temp;
    i := i+1;
  end;
  Fib := F;
end;  {Fib}
```

- Complexidade de tempo: $T(n) = n - 1$
- Complexidade de espaço: $E(n) = O(1)$

Procedimento recursivo

```
    Pesquisa(n);
(1) if n ≤ 1
(2) then "inspecione elemento" e termine
    else begin
(3)       para cada um dos n elementos "inspecione elemento";
(4)       Pesquisa(n/3);
        end;
```

- Para cada procedimento recursivo é associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos para o procedimento.
- Obtemos uma equação de recorrência para $f(n)$.
- **Equação de recorrência**: maneira de definir uma função por uma expressão envolvendo a mesma função.

Análise do procedimento recursivo

- Seja $T(n)$ uma função de complexidade que represente o número de inspeções nos n elementos do conjunto.
- O custo de execução das linhas (1) e (2) é $O(1)$ e da linha (3) é $O(n)$.
- Usa-se uma **equação de recorrência** para determinar o nº de chamadas recursivas.
- O termo $T(n)$ é especificado em função dos termos anteriores $T(1)$, $T(2)$, ..., $T(n-1)$.
- $T(n) = n + T(n/3)$, $T(1) = 1$ (para $n = 1$ fazemos uma inspeção).
- Por exemplo, $T(3) = T(3/3) + 3 = 4$, $T(9) = T(9/3) + 9 = 13$, e assim por diante.
- Para calcular o valor da função seguindo a definição são necessários $k - 1$ passos para computar o valor de $T(3^k)$.

Ex de resolução de equação de recorrência (1)

Substitui-se os termos $T(k)$, $k < n$, até que todos os termos $T(k)$, $k > 1$, tenham sido substituídos por fórmulas contendo apenas $T(1)$.

$$\begin{aligned}
T(n) &= n + T(n/3) \\
T(n/3) &= n/3 + T(n/3/3) \\
T(n/3/3) &= n/3/3 + T(n/3/3/3) \\
&\vdots \\
T(n/3/3 \cdots /3) &= n/3/3 \cdots /3 + T(n/3 \cdots /3)
\end{aligned}$$

Adicionando lado a lado, temos

$$T(n) = n + n \cdot (1/3) + n \cdot (1/3^2) + n \cdot (1/3^3) + \cdots + (n/3/3 \cdots /3)$$

que representa a soma de uma série geométrica de razão $1/3$, multiplicada por n, e adicionada de $T(n/3/3 \cdots /3)$, que é menor ou igual a 1.

Ex de resolução de equação de recorrência (2)

$$T(n) = n + n \cdot (1/3) + n \cdot (1/3^2) + n \cdot (1/3^3) + \cdots + \; + (n/3/3 \cdots /3)$$

Se desprezarmos o termo $T(n/3/3 \cdots /3)$, quando n tende para infinito, então

$$T(n) = n \sum_{i=0}^{\infty} (1/3)^i = n \left(\frac{1}{1 - \frac{1}{3}} \right) = \frac{3n}{2}.$$

Se considerarmos o termo $T(n/3/3/3 \cdots /3)$ e denominarmos x o número de subdivisões por 3 do tamanho do problema, então $n/3^x = 1$, e $n = 3^x$. Logo $x = \log_3 n$.

Ex de resolução de equação de recorrência (3)

Lembrando que $T(1) = 1$ temos

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{x-1} \frac{n}{3^i} + T(\frac{n}{3^x}) \\
&= n \sum_{i=0}^{x-1} (1/3)^i + 1 \\
&= \frac{n(1 - (\frac{1}{3})^x)}{(1 - \frac{1}{3})} + 1 \\
&= \frac{3n}{2} - \frac{1}{2}.
\end{aligned}
$$

Logo, o programa do exemplo é $O(n)$.

Comentários sobre recursividade

- Evitar o uso de recursividade quando existe uma solução óbvia por iteração!
- Exemplos:
  - Fatorial
  - Série de Fibonacci

Análise de algoritmos recursivos

- Comportamento é descrito por uma equação de recorrência
- Enfoque possível:
  - Usar a própria recorrência para substituir para $T(m)$, $m < n$ até que todos os termos tenham sido substituídos por fórmulas envolvendo apenas $T(0)$ ou o caso base

Análise da função `fat`

Seja a seguinte função para calcular o fatorial de n:

```
function fat(n : integer}) : integer};
begin
  if n <= 1
  then fat := 1
  else fat := n * fat(n-1);
end;  {fat}
```

Seja a seguinte equação de recorrência para esta função:

$$T(n) = \begin{cases} d & n = 1 \\ c + T(n-1) & n > 1 \end{cases}$$

Esta equação diz que quando $n = 1$ o custo para executar `fat` é igual a d. Para valores de n maiores que 1, o custo para executar `fat` é c mais o custo para executar $T(n-1)$

Resolvendo a equação de recorrência

Esta equação de recorrência pode ser expressa da seguinte forma:

$$
\begin{aligned}
T(n) &= c + T(n-1) \\
&= c + (c + T(n-2)) \\
&= c + c + (c + T(n-3)) \\
\vdots &= \vdots \\
&= c + c + \cdots + (c + T(1)) \\
&= \underbrace{c + c + \cdots + c}_{n-1} + d
\end{aligned}
$$

Em cada passo, o valor do termo T é substituído pela sua definição (ou seja, esta recorrência está sendo resolvida pelo método da expansão). A última equação mostra que depois da expansão existem $n-1$ c's, correspondentes aos valores de 2 até n. Desta forma, a recorrência pode ser expressa como:

$$T(n) = c(n-1) + d = O(n)$$

Alguns somatórios úteis

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

$$\sum_{i=0}^{k} 2^k = 2^{k+1} - 1$$

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{1}{2^k}$$

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1}-1}{a-1} (a \neq 1)$$

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

Algumas recorrências básicas: Caso 1

$$
\begin{aligned}
T(n) &= T(\tfrac{n}{2}) + 1 \quad (n \geq 2) \\
T(1) &= 0 \quad\quad\quad\quad\; (n = 1)
\end{aligned}
$$

Vamos supor que:

$$n = 2^k \Rightarrow k = \log n$$

Resolvendo por expansão temos:

$$
\begin{aligned}
T(2^k) &= T(2^{k-1}) + 1 \\
&= (T(2^{k-2}) + 1) + 1 \\
&= (T(2^{k-3}) + 1) + 1 + 1 \\
\vdots & \quad \vdots \\
&= (T(2) + 1) + 1 + \cdots + 1 \\
&= (T(1) + 1) + 1 + \cdots + 1 \\
&= 0 + \underbrace{1 + \cdots + 1}_{k} \\
&= k
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= \log n \\
T(n) &= O(\log n)
\end{aligned}
$$

Algumas recorrências básicas: Caso 2

$$
\begin{aligned}
T(n) &= 2T(\tfrac{n}{2}) + n \quad (n \geq 2) \\
T(1) &= 0 \quad\quad\quad\quad\quad (n = 1)
\end{aligned}
$$

Vamos supor que $n = 2^k \Rightarrow k = \log n$. Resolvendo por expansão temos:

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + 2^k \\
&= 2(2T(2^{k-2}) + 2^{k-1}) + 2^k \\
&= 2(2(2T(2^{k-3}) + 2^{k-2}) + 2^{k-1}) + 2^k \\
\vdots & \quad \vdots \\
&= 2(2(\cdots(2(2T(1) + 2^2) + 2^3) + \cdots) + 2^{k-1}) + 2^k \\
&= (k-1)2^k + 2^k \\
&= k2^k
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= n \log n \\
T(n) &= O(n \log n)
\end{aligned}
$$

Teorema Mestre (1)

Recorrências da forma

$$T(n) = aT(n/b) + f(n),$$

onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva podem ser resolvidas usando o Teorema Mestre. Note que neste caso não estamos achando a forma fechada da recorrência mas sim seu comportamento assintótico.

Teorema Mestre (2)

Sejam as constantes $a \geq 1$ e $b > 1$ e $f(n)$ uma função definida nos inteiros não-negativos pela recorrência:

$$T(n) = aT(n/b) + f(n),$$

onde a fração n/b pode significar $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. A equação de recorrência $T(n)$ pode ser limitada assintoticamente da seguinte forma:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $\boxed{T(n) = \Theta(n^{\log_b a})}$.
2. Se $f(n) = \Theta(n^{\log_b a})$, então $\boxed{T(n) = \Theta(n^{\log_b a} \log n)}$.
3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$ e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e para n suficientemente grande, então $\boxed{T(n) = \Theta(f(n))}$.

Comentários sobre o teorema Mestre (3)

- Nos três casos estamos comparando a função $f(n)$ com a função $n^{\log_b a}$. Intuitivamente, a solução da recorrência é determinada pela maior das duas funções.

- Por exemplo:
  - No primeiro caso a função $n^{\log_b a}$ é a maior e a solução para a recorrência é $T(n) = \Theta(n^{\log_b a})$.
  - No terceiro caso, a função $f(n)$ é a maior e a solução para a recorrência é $T(n) = \Theta(f(n))$.
  - No segundo caso, as duas funções são do mesmo "tamanho." Neste caso, a solução fica multiplicada por um fator logarítmico e fica da forma $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$.

Tecnicalidades sobre o teorema Mestre (4)

- No primeiro caso, a função $f(n)$ deve ser não somente menor que $n^{\log_b a}$ mas ser polinomialmente menor. Ou seja, $f(n)$ deve ser assintoticamente menor que $n^{\log_b a}$ por um fator de n^{ϵ}, para alguma constante $\epsilon > 0$.

- No terceiro caso, a função $f(n)$ deve ser não somente maior que $n^{\log_b a}$ mas ser polinomialmente maior e satisfazer a condição de "regularidade" que $af(n/b) \leq cf(n)$. Esta condição é satisfeita pela maior parte das funções polinomiais encontradas neste curso.

Tecnicalidades sobre o teorema Mestre (5)

- Teorema **não** cobre todas as possibilidades para $f(n)$:
  - Entre os casos 1 e 3 existem funções $f(n)$ que são menores que $n^{\log_b a}$ mas não são polinomialmente menores.
  - Entre os casos 2 e 3 existem funções $f(n)$ que são maiores que $n^{\log_b a}$ mas não são polinomialmente maiores.

  Se a função $f(n)$ cai numa dessas condições ou a condição de regularidade do caso 3 é falsa, então não se pode aplicar este teorema para resolver a recorrência.

Uso do teorema: Exemplo 1

$$T(n) = 9T(n/3) + n$$

Temos que,

$$a = 9, b = 3, f(n) = n$$

Desta forma,

$$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$$

Como $f(n) = O(n^{\log_3 9 - \epsilon})$, onde $\epsilon = 1$, podemos aplicar o caso 1 do teorema e concluir que a solução da recorrência é

$$T(n) = \Theta(n^2)$$

Uso do teorema: Exemplo 2

$$T(n) = T(2n/3) + 1$$

Temos que,

$$a = 1, b = 3/2, f(n) = 1$$

Desta forma,

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

O caso 2 se aplica já que $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$. Temos, então, que a solução da recorrência é

$$T(n) = \Theta(\log n)$$

Uso do teorema: Exemplo 3

$$T(n) = 3T(n/4) + n \log n$$

Temos que,

$$a = 3, b = 4, f(n) = n \log n$$

Desta forma,

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

Como $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, onde $\epsilon \approx 0.2$, o caso 3 se aplica se mostrarmos que a condição de regularidade é verdadeira para $f(n)$.

Uso do teorema: Exemplo 3

Para um valor suficientemente grande de n

$$af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n\log n = cf(n)$$

para $c = 3/4$. Conseqüentemente, usando o caso 3, a solução para a recorrência é

$$T(n) = \Theta(n\log n)$$

Uso do teorema: Exemplo 4

$$T(n) = 2T(n/2) + n\log n$$

Temos que,

$$a = 2, b = 2, f(n) = n\log n$$

Desta forma,

$$n^{\log_b a} = n$$

Aparentemente o caso 3 deveria se aplicar já que $f(n) = n\log n$ é assintoticamente maior que $n^{\log_b a} = n$. Mas no entanto, não é polinomialmente maior. A fração $f(n)/n^{\log_b a} = (n\log n)/n = \log n$ que é assintoticamente menor que n^{ϵ} para toda constante positiva ϵ. Conseqüentemente, a recorrência cai na situação entre os casos 2 e 3 onde o teorema não pode ser aplicado.

Uso do teorema: Exercícios

1. $T(n) = 4T(n/2) + n$
2. $T(n) = 4T(n/2) + n^2$
3. $T(n) = 4T(n/2) + n^3$
4. O tempo de execução de um algoritmo A é descrito pela recorrência $T(n) = 7T(n/2) + n^2$. Um outro algoritmo A' tem um tempo de execução descrito pela recorrência $T'(n) = aT'(n/4) + n^2$. Qual é o maior valor inteiro de a tal que A' é assintoticamente mais rápido que A?

Notação assintótica em funções (1)

Normalmente, a notação assintótica é usada em fórmulas matemáticas. Por exemplo, usando a notação O pode-se escrever que $n = O(n^2)$. Também pode-se escrever que

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$

Como se interpreta uma fórmula como esta?

Notação assintótica em funções (2)

- Notação assintótica sozinha no lado direito de uma equação, como em $n = O(n^2)$
  - Sinal de igualdade significa que o lado esquerdo é um membro do conjunto $O(n^2)$
  - $n \in O(n^2)$ ou $n \subseteq n^2$
- Nunca deve-se escrever uma igualdade onde a notação O aparece sozinha com os lados trocados
  - Caso contrário, poderia se deduzir um absurdo como $n^2 = n$ de igualdades como em $O(n^2) = n$
- Quando se trabalha com a notação O e em qualquer outra fórmula que envolve quantidades não precisas, o sinal de igualdade é unidirecional
  - Daí vem o fato que o sinal de igualdade (`"="`) realmente significa $\in$ ou $\subseteq$, usados para inclusão de conjuntos

Notação assintótica em funções (3)

Se uma notação assintótica aparece numa fórmula, isso significa que essa notação está substituindo uma função que não é importante definir precisamente (por algum motivo). Por exemplo, a equação

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$

significa que

$$2n^2 + 3n + 1 = 2n^2 + f(n)$$

onde $f(n)$ é alguma função no conjunto $\Theta(n)$. Neste caso, $f(n) = 3n + 1$ que de fato está em $\Theta(n)$.

Notação assintótica em funções (4)

O uso da notação assintótica desta forma ajuda a eliminar detalhes que não são importantes. Por exemplo, pode-se expressar uma equação de recorrência como:

$$T(n) = 2T(n-1) + \Theta(n).$$

Se se deseja determinar o comportamento assintótico de $T(n)$ então não é necessário determinar exatamente os termos de mais baixa ordem. Entende-se que eles estão incluídos numa função $f(n)$ expressa no termo $\Theta(n)$.

Notação assintótica em funções (5)

Em alguns casos, a anotação assintótica aparece do lado esquerdo de uma equação como em:

$$2n^2 + \Theta(n) = \Theta(n^2).$$

A interpretação de tais equações deve ser feita usando a seguinte regra:

- É possível escolher uma função $f(n)$ para o lado esquerdo da igualdade de tal forma que existe uma função $g(n)$ para o lado direito que faz com que a equação seja válida
- O lado direito da igualdade define um valor não tão preciso quanto o lado esquerdo. Por exemplo,

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \qquad (1) \\ 2n^2 + \Theta(n) &= \Theta(n^2). \qquad (2) \end{aligned}$$

Notação assintótica em funções (6)

As equações (1) e (2) podem ser interpretadas usando a regra acima:

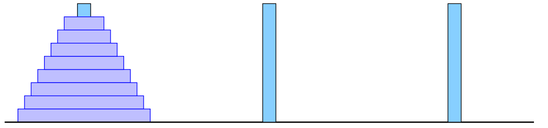
- A equação (1) diz que existe alguma função $f(n) \in \Theta(n)$ tal que $2n^2 + 3n + 1 = 2n^2 + f(n)$ para todo n.
- A equação (2) diz que para qualquer função $g(n) \in \Theta(n)$, existe uma função $h(n) \in \Theta(n^2)$ tal que

$$2n^2 + g(n) = h(n)$$

para todo n. Note que esta interpretação implica que $2n^2+3n+1 = \Theta(n^2)$, que é o que estas duas equações querem dizer.

Modelagem usando equação de recorrência (1) *Torre de Hanoi*

- Inventor:
  - Em 1883, o matemático francês Edouard Lucas criou um jogo chamado *Torre de Hanoi*
- Configuração inicial:
  - O jogo começa com um conjunto de oito discos empilhados em tamanho decrescente em uma das três varetas



Modelagem usando equação de recorrência (2) *Torre de Hanoi*

- Objetivo:
  - Transferir toda a torre para uma das outras varetas, movendo um disco de cada vez, mas nunca movendo um disco maior sobre um menor
- Soluções particulares:
  - Seja $T(n)$ o número mínimo de movimentos para transferir n discos de uma vareta para outra de acordo com as regras definidas no enunciado do problema.
  - Não é difícil observar que:

$$\begin{aligned} T(0) &= 0 \quad \text{[nenhum movimento é necessário]} \\ T(1) &= 1 \quad \text{[apenas um movimento]} \\ T(2) &= 3 \quad \text{[três movimentos usando as duas varetas]} \end{aligned}$$

Modelagem usando equação de recorrência (3) *Torre de Hanoi*

- Generalização da solução:
  - Para três discos, a solução correta é transferir os dois discos do topo para a vareta do meio, transferir o terceiro disco para a outra vareta e, finalmente, mover os outros dois discos sobre o topo do terceiro.
  - Para n discos:
    1. Transfere-se os $n-1$ discos menores para outra vareta (por exemplo, a do meio), requerendo $T(n-1)$ movimentos.
    2. Transfere-se o disco maior para a outra vareta (1 movimento).
    3. Transfere-se os $n-1$ discos menores para o topo do disco maior, requerendo-se $T(n-1)$ movimentos novamente.

Modelagem usando equação de recorrência (4)
Torre de Hanoi

- Equação de recorrência para este problema pode ser expressa por:

$$\begin{aligned} T(0) &= 0 \\ T(n) &= 2T(n-1)+1, \quad \text{para } n > 0 \end{aligned}$$

Modelagem usando equação de recorrência (5)
Torre de Hanoi

Para pequenos valores de n temos:

n	0	1	2	3	4	5	6
$T(n)$	0	1	3	7	15	31	63

Esta recorrência pode ser expressa por

$$T(n) = 2^n - 1$$

Modelagem usando equação de recorrência (6)
Torre de Hanoi

Provando por indução matemática temos:

Caso base. Para $n = 0$ temos que $T(0) = 2^0 - 1 = 0$, que é o valor presente na equação de recorrência.

Indução. A indução será feita em n. Vamos supor que a forma fechada seja válida para todos os valores até $n-1$, ou seja, $T(n-1) = 2^{n-1} - 1$. Vamos provar que esta forma fechada é de fato válida para $T(n)$.

$$T(n) = 2T(n-1)+1 = 2(2^{n-1}-1)+1 = 2^n - 2 + 1 = 2^n - 1.$$

$\therefore$ A forma fechada proposta também é válida para n.

Modelagem usando equação de recorrência (7)
Estratégia para resolução da equação

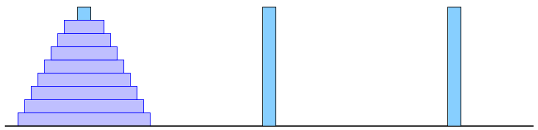
A recorrência da Torre de Hanoi aparece em várias aplicações de todos os tipos. Normalmente, existem três etapas para achar uma forma fechada para o valor de $T(n)$:

1. Analisar pequenos casos. Com isto podemos ter um entendimento melhor do problema e, ao mesmo tempo, ajudar nos dois passos seguintes.
2. Achar e provar uma recorrência para o valor de $T(n)$.
3. Achar e provar uma forma fechada para a recorrência.

Modelagem usando equação de recorrência (8)
Linhas no plano

- Problema:
  - Qual é o número máximo de regiões L_n determinado por n retas no plano?

Lembre-se que um plano sem nenhuma reta tem uma região, com uma reta tem duas regiões e com duas retas têm quatro regiões.



Análise Amortizada: Introdução

- Cenário:
  - Manter uma estrutura de dados sobre uma seqüência de n operações.
- Custo por operação:
  - Pode ser alto, por exemplo, $\Theta(n)$.
- Custo total:
  - Pode não ser $n \times$ "custo no pior caso de uma operação".
- Questão:
  - Como fazer uma análise mais precisa num cenário como esse?
- Solução:
  - Análise amortizada.

Custo amortizado

- Definição:
  - Custo médio de uma operação sobre uma seqüência de n operações, maximizado sobre todos n e todas seqüências.
- Observações importantes:
  - O custo amortizado não é a mesma coisa que a análise do caso médio.
  - Não é feita nenhuma suposição sobre a seqüência de entrada.
  - Amortização é ainda um princípio de pior caso.

Técnicas de análise amortizada

Três técnicas que podem ser usadas para analisar tais cenários são:

- Método Agregado (*Aggregate Method*).
- Método Contábil (*Accounting Method*).
- Método Potencial (*Potential Method*).

Referência:

➜ Introduction to Algorithms, 2nd edition. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. MIT Press, Hardcover, Published July 2001, ISBN 0070131511.

Exemplo: Incrementar um contador binário

- Exemplo didático (*toy example*).
- Estrutura de dados:
  - Arranjo de k bits, usado para armazenar/contar números de 0 a $2^k - 1$ e depois novamente a 0.
- Operação:
  - Incrementar.
- Custo:
  - Número de bits trocados.
  - Exemplo: arranjo de 5 bits.

Exemplo: Incrementar um contador binário

Bits	Custo
00000	
00001	1
00010	2
00011	1
00100	3

➜ Custo varia!

➜ Custo máximo: $O(\log n)$.

Contador binário: Código

- Contador binário de k bits.
- Vetor $A[0 \ldots k-1]$
  - Bit menos significativo na posição $A[0]$.
  - Bit mais significativo na posição $A[k-1]$.
  - Um número binário x é dado por $\sum_{i=0}^{k-1} A[i] \cdot 2^i$.
- length$[A] = k$.
- Para somar 1 (módulo 2^k) ao valor do contador, pode-se usar o procedimento `Increment`(*A*).

Contador binário: Código

```
Increment(A)
  i ← 0
  while (i < length[A]) ∧ (A[i] = 1) do
  begin
    A[i] ← 0
    i ← i + 1
  end
  if i < length[A]
  then A[i] ← 1
EndIncrement
```

➜ Essencialmente o mesmo algoritmo implementado pelo contador *Ripple-Carry* em hardware.

Método agregado

- Idéia:
  - Conte o custo total para as n operações.
- Técnica de contagem:
  - Ad hoc, ou seja, cada caso é um caso.

Contador binário

- Custo por linha varia.
  - Difícil de somar.
- Solução:
  - Conte por coluna, ou seja, quantas vezes o i-ésimo bit é trocado.
  - Some os custos.
  - Exemplo: arranjo de 5 bits ($k = 5$).

Contador binário

Bits					Custo
0	0	0	0	0	
0	0	0	0	1	1
0	0	0	1	0	2
0	0	0	1	1	1
0	0	1	0	0	3
		⋮			
1	1	1	1	1	1
0	0	0	0	0	5
↑	↑	↑	↑	↑	
2	4	8	16	32	

Contador binário

- Custo:
  - Para uma seqüência de n incrementos (2^k), temos:

Bit de ordem				
2^4	2^3	2^2	2^1	2^0
2	4	8	16	32

ou seja, o bit de ordem 2^i é trocado 2^{k-i} vezes. Logo, o custo total de trocas é

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

➜ Custo amortizado por troca = $O(n)/n = O(1)$.

Método contábil

- Estrutura de dados vem com uma "conta bancária".
- À cada operação é alocado um custo fixo (custo amortizado).
  - Custos devem ser escolhidos cuidadosamente.
- Se o custo real é menor que o custo alocado, deposite a diferença na conta bancária.
- Se o custo real é maior que o custo alocado, retire a diferença da conta bancária para pagar pela operação.
- Prove que o saldo nunca fica negativo.

Método contábil

- Qual o significado do saldo ficar negativo?
  - ➜ Para a seqüência de operações até aquele momento, o custo amortizado total não representa um limite superior para o custo real total.
- Conclusão:
  - Seqüência de n operações custa no máximo $n \times$ "custo amortizado".

Contador binário

- Custo amortizado para trocar $0 \to 1 = 2$ créditos.
  - Pague 1 crédito pela operação e deposite 1 crédito na conta.
- Custo amortizado para trocar $1 \to 0 = 0$ créditos
  - Retire da conta 1 crédito para pagar pela operação.
- Invariante:
  - Cada bit 1 no contador gerou um crédito.
  - Logo, sempre existe crédito para pagar pela operação de trocar $1 \to 0$.

Análise amortizada

- Custo para resetar os bits no **while** é pago pelos créditos dos bits que foram setados.
- No máximo um bit é setado.
  - Custo da operação de incremento é no máximo 2.
- O número de bits 1 no contador nunca é negativo.
  - Saldo nunca é negativo.
- Para n operações de incremento, o custo amortizado total é $O(n)$, que é um limite para o custo real total.

Método potencial

- Associa-se uma energia potencial a cada estrutura de dados.
- Energia potencial é o "potencial para fazer um estrago".
- Custo amortizado = Custo atual +
  Novo potencial −
  Potencial anterior
  - Deve-se pagar para incrementar o potencial da estrutura de dados.
- Se a operação tem um custo cumulativo alto mas reduz bastante o potencial, então o custo amortizado é baixo.
- Como encontrar o potencial?
  ➜ Determine o que torna uma estrutura de dados ruim.

Regras básicas para funções potenciais

- Devem ser sempre não negativas.
- Devem começar de zero.
- Implica uma seqüência de n operações que custam no máximo $n\times$ "custo amortizado".

Contador binário

- Estrutura de dados é "ruim" se tem vários 1's.
- Seja Φ(Contador)= # 1's no contador.
- Potencial aumenta quando há um incremento:

  = #(0 $\rightarrow$ 1) − #(1 $\rightarrow$ 0)
  = 1 − #(1 $\rightarrow$ 0)
- Custo amortizado do incremento:

  = Custo real (atual) + Aumento do potencial
  = (1 + #(1 $\rightarrow$ 0)) + (1 − #(1 $\rightarrow$ 0))
  = 2
- Custo amortizado é 2 e o custo de n incrementos é no máximo $2n$.