Algoritmos e Estruturas de Dados 2

Introdução

Prof. Fábio M. Costa

Instituto de Informática
Universidade Federal de Goiás

1o. Semestre / 2019

Relembrando definições básicas

Dado

Um item qualquer de informação
Dado atômico: um item de informação único não passível de decomposição. Ex.: inteiros
Dado composto: pode ser subdividido em sub-campos com significado próprio. Ex.: registros

Tipos de Dados

- Tipos primitivos (built-in data types). Ex.: `int`, `float`, `char`
- Tipos construídos (User-Defined Data Types). Ex.: `class`, `struct`, `union`

$\rightarrow$ Um tipo de dados define o conjunto de valores que uma variável pode assumir.

Relembrando definições básicas

Objeto de Dados

Contêiner (em memória) para valores de dados.
Uma instância de uma estrutura de dados em tempo de execução
Criado por meio de declaração de variáveis
Caracterizado por um conjunto de atributos:

- tipo de dados
- número de valores de dados que o objeto contém
- organização lógica dos valores

Relembrando definições básicas

Estrutura de Dados

Refere-se à representação dos dados em um programa

Uma coleção de tipos de dados, atômicos e/ou compostos, com relacionamentos estruturados bem definidos

Estrutura definida por meio de um conjunto de regras que mantém os dados coesos

Em suma:

- uma combinação de elementos cujos valores pertencem a tipos de dados pré-definidos ou são instâncias de outras estruturas de dados
- um conjunto de associações ou relacionamentos entre os elementos de dados combinados

Definição formal de uma estrutura de dados

Domínio (D): Faixa de valores que os dados podem assumir
Funções (F): Conjunto de operações sobre os dados
Axiomas (A): Conjunto de regras com as quais as operações em F devem ser implementadas

Exemplo:

```
Integer
    Domain D = {Integer, Boolean}
    Set of functions F = {zero, ifzero, add, increment}
    Set of axioms A = {
                        ifzero(zero()) -> true;
                        ifzero(increment(zero()) -> false
                        add(zero(), x) -> x
                        add(increment(x), y) = increment(add(x, y))
                        equal(increment(x), increment(y)) = equal(x, y)
                      }
end Integer
```

Abstração

Permite lidar com a complexidade de uma tarefa ou problema a ser resolvido

- focando nas propriedades lógicas dos dados e ações envolvidos (ao invés de seus detalhes de implementação)

Abstração de Dados: separação entre as propriedades lógicas dos dados e a forma como eles são representados
Abstração Procedural: separação entre as propriedades lógicas das ações e sua implementação

Tipo Abstrato de Dados

Um TAD é uma declaração de estrutura de dados juntamente com as operações que a ela se aplicam
Um TAD é definido por meio do uso de **abstração de dados** e **abstração procedural**
$\rightarrow$ Encapsula os dados e as operações sobre eles de maneira que a representação dos dados e a implementação das operações são ocultas ao usuário – P: Qual o benefício disto?

Exemplo

Considere um TAD que representa uma fila

- a fila pode ser implementada usando um vetor, uma lista ligada ou um arquivo
- contudo, o usuário não deve estar ciente de qual estrutura de dados é usada
- apenas o comportamento lógico das operações de inserção e remoção interessa ao usuário

Tipo Abstrato de Dados

Exemplo: um TAD para a estrutura de dados `Integer`:

```
Abstract data type Integer
  Operations
     zero() -> int
     ifzero(int) -> boolean
     increment(int) -> int
     add(int, int) -> int
     equal(int, int) -> boolean
  Rules/axioms for operations
     for all x, y OE integer let
       ifzero(zero()) -> true;
       ifzero(increment(zero()) -> false
       add(zero(), x) -> x
       add(increment(x), y) -> increment(add(x, y))
       equal(increment(x), increment(y) -> equal(x, y)
end Integer
```

Como poderia ser a implementação deste TAD?

Formas de classificar estruturas de dados

- primitivas e não-primitivas
- lineares e não-lineares
- estáticas e dinâmicas
- persistentes e efêmeras
- sequenciais e de acesso direto

Estruturas de dados primitivas e não-primitivas

Estruturas de dados primitivas

- definem conjuntos de elementos que não envolvem sub-partes
- exemplo: inteiro e caractere
- geralmente correspondem a tipos primitivos em linguagens de programação

Estruturas de dados não-primitivas

- definem conjuntos de elementos derivados a partir de tipos preexistentes
- ex. 1: vetores – conjunto de elementos do mesmo tipo
- ex. 2: `class` e `struct` – conjunto de elementos que podem ser de tipos diferentes (juntamente com funções para operar sobre esses elementos)

Estruturas de dados lineares e não-lineares

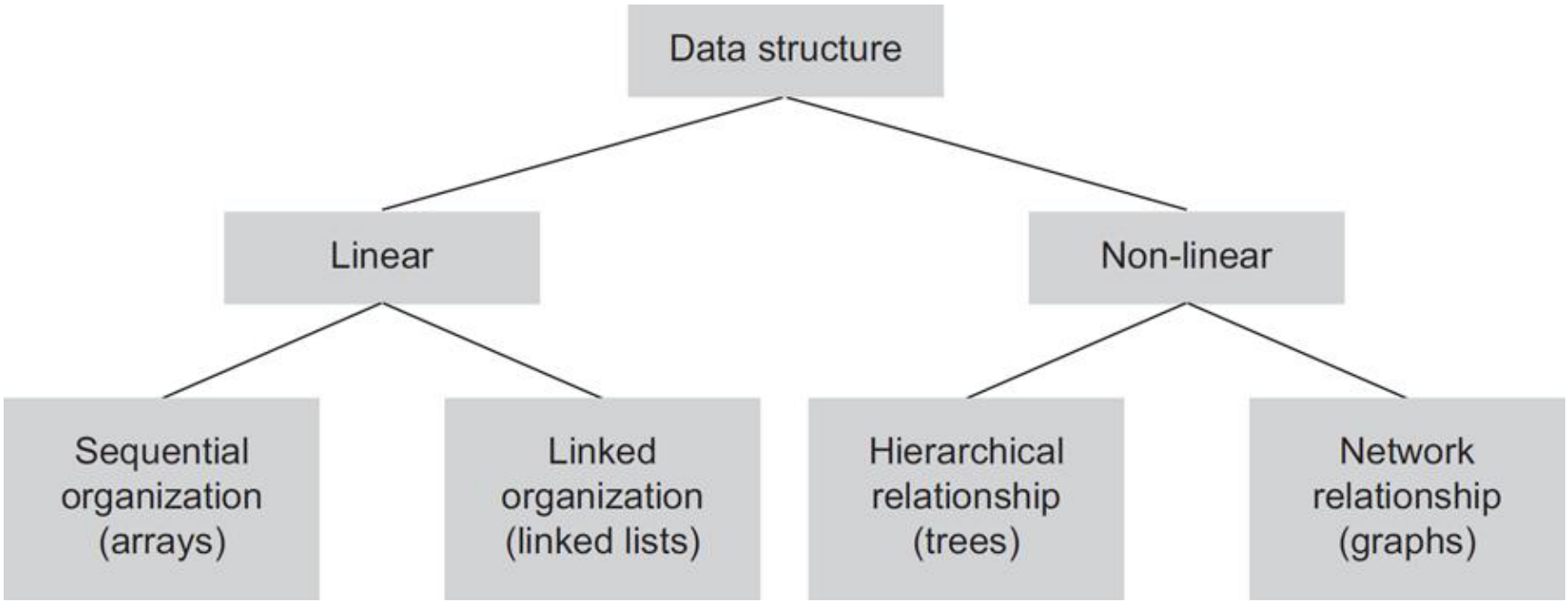
Estrutura de dados linear

- elementos formam uma sequência
- todo elemento tem um único predecessor e/ou sucessor
- duas formas de representação em memória: listas ligadas e vetores

Estrutura de dados não-linear

- usadas para representar dados que contém algum tipo de relacionamento hierárquico ou de rede entre seus elementos
- exemplos: árvores e grafos
- cada elemento de dados pode ter mais de um sucessor ou predecessor
- elementos não formam qualquer sequência linear em particular

Estruturas de dados lineares e não-lineares



Estruturas de dados estáticas e dinâmicas

Estrutura de dados estática

- criada (i.e., alocada) em tempo de compilação
- referenciada por meio de um nome de variável
- exemplo clássico: vetor

Estrutura de dados dinâmica

- criada em tempo de execução, por meio de alocação dinâmica de memória
- referenciadas indiretamente por meio de ponteiros
- exemplo: listas ligadas
- árvores e grafos podem ser implementados como estruturas de dados dinâmicas

Obs.: Não é o mesmo que conjunto de dados estático ou dinâmico.

Estruturas de dados persistentes e efêmeras

Estrutura de dados persistente

- uma alteração na estrutura de dados (ex.: acrescentar um novo elemento em uma lista) causa a criação de uma nova versão da estrutura
- versões diferentes coexistem e são acessíveis individualmente
- caso geral em linguagens funcionais

Estruturas de dados efêmeras

- alterações na estrutura de dados são realizadas na mesma cópia da estrutura – versões anteriores são perdidas
- apenas a última versão é acessível
- caso geral em linguagens imperativas

Estruturas de dados de acesso sequencial e de acesso direto

Acesso sequencial

- para acessar o n-ésimo elemento, todos os $n-1$ elementos predecessores devem ser acessados
- exemplo: listas ligadas

Acesso direto

- qualquer elemento pode ser acessado sem a necessidade de acessar os predecessores
- exemplo: vetores

Análise de Algoritmos

Motivação: Comparação entre diferentes soluções algorítmicas para o mesmo problema

Objetivo

Medir o desempenho de um algoritmo

- Complexidade de tempo de programação – pouco usada
- Complexidade de tempo (de execução) – quantidade de tempo necessária para executar uma tarefa
- Complexidade de espaço – quantidade de memória necessária para executar uma tarefa

Classificar algoritmos com base nas quantidades relativas de tempo e memória necessárias
Especificando o crescimento da quantidade de tempo e memória gastos **como uma função do tamanho da entrada**

Complexidade de Algoritmos

Taxa de crescimento da quantidade de tempo ou memória consumida pelo algoritmo em relação ao tamanho da entrada

- Mensurada por meio de funções matemáticas padronizadas

Tradeoff: economia de tempo em detrimento de um maior consumo de memória (ou vice-versa)
Geralmente, o tempo de execução é o gargalo – ênfase na complexidade de tempo

Função de tempo

$T(n)$: quanto tempo é necessário para executar o algoritmo com n valores de entrada

- Ex.: algoritmo de ordenação para ordenar n valores de dados
- Ordenação por inserção: $\mathcal{O}(n^2)$ – o tempo de execução cresce, no máximo, com o quadrado do número de elementos a serem ordenados; neste caso, a função de tempo é $T(n) = n^2$

Complexidade de Espaço (Memória)

Quantidade de memória necessária durante a execução do programa em função do tamanho da entrada
Pode ser medida em dois momentos:

- Tempo de compilação – memória alocada estaticamente
- Tempo de execução – memória alocada dinamicamente ou como resultado de chamadas recursivas

Componentes da complexidade de espaço:

- Espaço do programa (código)
- Espaço de dados (pilha + heap)

Complexidade de Tempo

Para o tempo de execução, poderíamos considerar o tempo absoluto (em minutos, segundos, etc.). Mas não é interessante em termos gerais, pois depende da máquina em que o programa for executado.

Em Análise de Algoritmos conta-se o número de operações consideradas relevantes realizadas pelo algoritmo e expressa-se esse número como uma função de n.
Essas operações podem ser comparações, operações aritméticas, movimentações de dados, etc.
Em geral, estamos interessados no pior caso, embora possamos analisar também o melhor caso e o caso médio.

Exemplo: Busca sequencial de um dado elemento em um vetor que armazena n elementos em ordem aleatória.
$\rightarrow$ Qual/quais operações são relevantes para o tempo de execução?
$\rightarrow$ Discuta o pior caso, melhor caso e o caso médio.

Complexidade de Tempo

Como exemplo, considere o número de operações de dois algoritmos que resolvem o mesmo problema, como função de n.

Algoritmo 1: $f1(n) = 2n^2 + 5n$ operações
Algoritmo 2: $f2(n) = 500n + 4000$ operações

Dependendo do valor de n, o Algoritmo 1 pode requerer mais ou menos operações que o Algoritmo 2.
(Compare as duas funções para n = 10 e n = 100.)

Comportamento Assintótico

Algoritmo 1: $f1(n) = 2n^2 + 5n$ operações
Algoritmo 2: $f2(n) = 500n + 4000$ operações

Um caso de particular interesse: quando n tem valor muito grande, denominado comportamento assintótico.

Os termos inferiores e as constantes multiplicativas contribuem pouco na comparação assintótica entre os algoritmos e podem ser descartados.

- Ou seja, o comportamento assintótico é determinado pelo termo de maior grau em $f(n)$.

No exemplo, o importante é observar que $f1(n)$ cresce com n^2, ao passo que $f2(n)$ cresce com n.
Um crescimento quadrático é considerado pior que um crescimento linear. Assim, vamos preferir o Algoritmo 2 ao Algoritmo 1.

Notação O (Big-O)

Dada uma função $g(n)$, denotamos por $\mathcal{O}(g(n))$ o conjunto das funções

$$\{f(n) : \exists \text{ constantes } c \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq c \cdot g(n) \text{ para } n \geq n_0\}$$

Isto é, para valores de n suficientemente grandes, $f(n)$ é igual ou menor que $g(n)$. Ou seja, $\mathcal{O}$ denota um limite superior.
Como abuso de notação, vamos escrever $f(n) = \mathcal{O}(g(n))$ ao invés de $f(n) \in \mathcal{O}(g(n))$.

Algoritmo 1: $f1(n) = 2n^2 + 5n = \mathcal{O}(n^2)$ — i.e., $g(n) = n^2$
Algoritmo 2: $f2(n) = 500n + 4000 = \mathcal{O}(n)$ — i.e., $g(n) = n$

Um polinômio de grau d é de ordem $\mathcal{O}(n^d)$. Como uma constante pode ser considerada como um polinômio de grau 0, então dizemos que uma constante é $\mathcal{O}(n^0)$ ou seja $\mathcal{O}(1)$.

Notação O (Big-O)

Alguns termos comuns (em ordem de significância assintótica):

$$\log_2 n \ldots n \ldots n \log_2 n \ldots n^2 \ldots n^3 \ldots n^k \ldots 2^n \ldots n!$$

Exemplo:

$$f(n) = n \times \frac{(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

Removendo os coeficientes, temos:

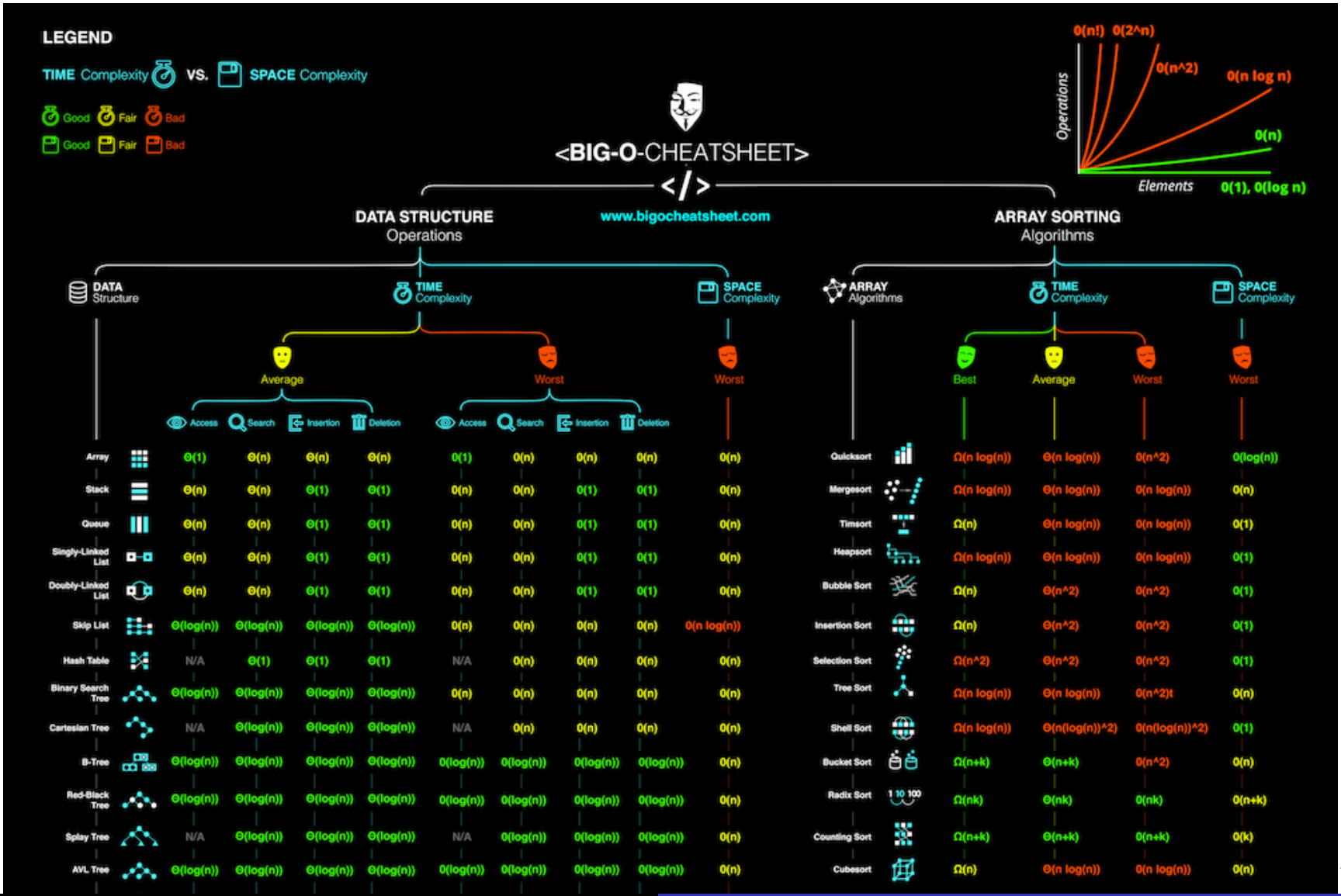
$$n^2 + n$$

Removendo o termo menor, sobra n^2
Logo: $\mathcal{O}(f(n)) = \mathcal{O}(n^2)$

Exercícios sobre a Notação O

1. É verdade que $2n^2 + 100n = \mathcal{O}(n^2)$? Prove.
2. É verdade que $10 + \frac{4}{n} = \mathcal{O}(n^0) = \mathcal{O}(1)$? Prove.
3. Escreva a seguinte função em notação $\mathcal{O}$: $4n^2 + 10 \log n + 500$.
4. O mesmo para a função $5n^n + 102^n$
5. Idem para a função $2(n-1)^n + n^{n-1}$

Notação O – Cheat Sheet



Notação Ω

Dada uma função $g(n)$, denotamos por $\Omega(g(n))$ o conjunto das funções

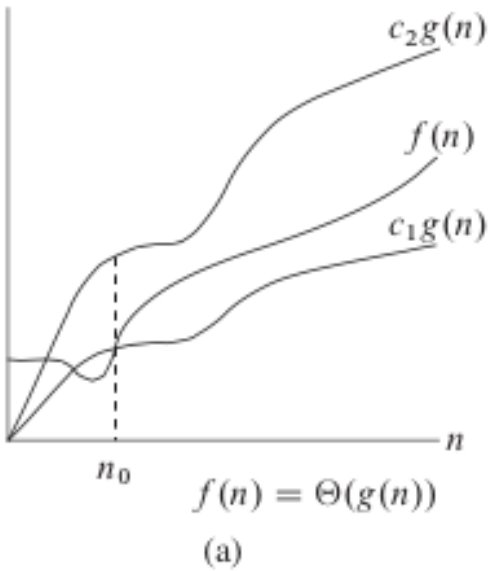
$$\{f(n) : \exists \text{ constantes } c \text{ e } n_0 \text{ tais que } 0 \leq c \cdot g(n) \leq f(n) \text{ para } n \geq n_0.\}$$

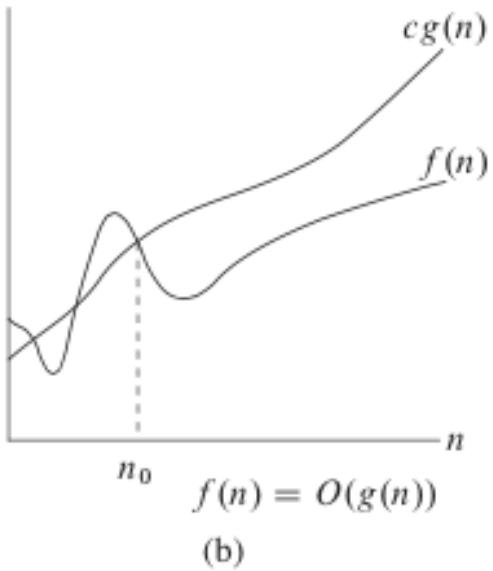
Isto é, para valores de n suficientemente grandes, $f(n)$ é igual ou maior que $g(n)$. Ou seja, Ω denota um limite inferior.
Novamente, abusando a notação, vamos escrever $f(n) = \Omega(g(n))$.

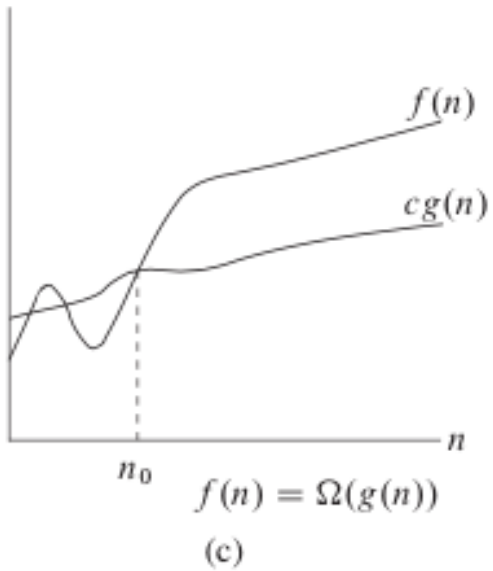
Notação Θ

Dadas duas funções $f(n)$ e $g(n)$, temos
$f(n) = \Theta(g(n))$
se e somente se
$f(n) = \mathcal{O}(g(n))$ e
$f(n) = \Omega(g(n))$

Notações O, Ω e Θ (graficamente)







Importância da Análise de Complexidade

Considere 5 algoritmos com as complexidades de tempo mostradas abaixo. Suponhamos que uma operação leve 1ms.

n	$f_1(n) = n$	$f_2(n) = n \log n$	$f_3(n) = n^2$	$f_4(n) = n^3$	$f_5(n) = 2^n$
16	0.016s	0.064s	0.256s	4s	1m 4s
32	0.032s	0.16s	1s	33s	46 dias
521	0.512s	9s	4m 22s	1 dia 13h	10^{137} séculos

Importância da Análise de Complexidade

Considere 5 algoritmos com as complexidades de tempo mostradas abaixo. Suponhamos que uma operação leve 1ms.

n	$f_1(n) = n$	$f_2(n) = n \log n$	$f_3(n) = n^2$	$f_4(n) = n^3$	$f_5(n) = 2^n$
16	0.016s	0.064s	0.256s	4s	1m 4s
32	0.032s	0.16s	1s	33s	46 dias
521	0.512s	9s	4m 22s	1 dia 13h	10^{137} séculos

Resolveria usar uma máquina mais rápida onde uma operação leve 1ps (pico segundo) ao invés de 1ms? R: Ao invés de 10^{137} séculos, seriam 10^{128} séculos :-)

Importância da Análise de Complexidade

Considere 5 algoritmos com as complexidades de tempo mostradas abaixo. Suponhamos que uma operação leve 1ms.

n	$f_1(n) = n$	$f_2(n) = n \log n$	$f_3(n) = n^2$	$f_4(n) = n^3$	$f_5(n) = 2^n$
16	0.016s	0.064s	0.256s	4s	1m 4s
32	0.032s	0.16s	1s	33s	46 dias
521	0.512s	9s	4m 22s	1 dia 13h	10^{137} séculos

Resolveria usar uma máquina mais rápida onde uma operação leve 1ps (pico segundo) ao invés de 1ms? R: Ao invés de 10^{137} séculos, seriam 10^{128} séculos :-)

Podemos, muitas vezes, melhorar o tempo de execução de um programa otimizando o código (Ex.: usar x + x ao invés de 2*x , evitar re-cálculo de expressões já calculadas, etc.).

Importância da Análise de Complexidade

Considere 5 algoritmos com as complexidades de tempo mostradas abaixo. Suponhamos que uma operação leve 1ms.

n	$f_1(n) = n$	$f_2(n) = n \log n$	$f_3(n) = n^2$	$f_4(n) = n^3$	$f_5(n) = 2^n$
16	0.016s	0.064s	0.256s	4s	1m 4s
32	0.032s	0.16s	1s	33s	46 dias
521	0.512s	9s	4m 22s	1 dia 13h	10^{137} séculos

Resolveria usar uma máquina mais rápida onde uma operação leve 1ps (pico segundo) ao invés de 1ms? R: Ao invés de 10^{137} séculos, seriam 10^{128} séculos :-)

Podemos, muitas vezes, melhorar o tempo de execução de um programa otimizando o código (Ex.: usar x + x ao invés de 2*x , evitar re-cálculo de expressões já calculadas, etc.).

Entretanto, melhorias muito mais substanciais podem ser obtidas se usarmos um algoritmo diferente, com menor complexidade de tempo. Ex.: obter um algoritmo $\mathcal{O}(n \log n)$ ao invés de um $\mathcal{O}(n^2)$.

Exemplo de uso: Sequência de Fibonacci

Para projetar um algoritmo eficiente, é fundamental preocupar-se com sua complexidade. Como exemplo: considere a sequência de Fibonacci.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

A sequência pode ser definida recursivamente:

$$F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

Dado o valor de n, queremos obter o n-ésimo elemento da sequência. Vamos apresentar dois algoritmos aternativos e considerar sua complexidade.

Algoritmo 1:

Seja a função fibo1(n) que calcula o n-ésimo elemento da sequência de Fibonacci.

```
1       Input: Valor de n
2       Output: n-ésimo elemento da sequência de Fibonacci
3       Function fibo1(n)
4       if n == 0 then
5           return 0
6       else
7           if n == 1 then
8               return 1
9           else
10              return fibo1(n - 1) + fibo1(n - 2)
11          end if
12      end if
```

Experimente rodar este algoritmo para n = 100 :-)

Algoritmo 1:

Seja a função fibo1(n) que calcula o n-ésimo elemento da sequência de Fibonacci.

```
1      Input: Valor de n
2      Output: n–ésimo elemento da sequência de Fibonacci
3      Function fibo1(n)
4      if n == 0 then
5          return 0
6      else
7          if n == 1 then
8              return 1
9          else
10             return fibo1(n - 1) + fibo1(n - 2)
11         end if
12     end if
```

Experimente rodar este algoritmo para n = 100 :-)
A complexidade é $\mathcal{O}(2^n)$. Demonstre.
(Mesmo que uma operação demore apenas 1ps, 2^{100} operações levariam 3×10^{13} anos = 30.000.000.000.000 anos.)

Algoritmo 2

```
1  Function fibo2(n)
2      if n == 0 then
3          return 0
4      else
5          if n == 1 then
6              return 1
7          else
8              penultimo = 0
9              ultimo = 1
10             for i = 2 until n do
11                 atual = penultimo + ultimo
12                 penultimo = ultimo
13                 ultimo = atual
14             end for
15             return atual
16         end if
17     end if
```

A complexidade agora passou de $\mathcal{O}(2^n)$ para $\mathcal{O}(n)$. Demonstre.
Obs.: É possível fazer em $\mathcal{O}(\log n)$. Pesquise.

Perguntas?

Atividade Supervisionada

1. Faça um resumo das estruturas de dados e algoritmos estudados em AED1 (ou na disciplina equivalente a Estruturas de Dados 1).
2. Para o problema da ordenação, apresente:
   a. um algoritmo de complexidade $\mathcal{O}(n^2)$
   b. um algoritmo de complexidade $\mathcal{O}(n \log n)$
   c. uma demonstração do limite (cota) inferior de complexidade para o problema da ordenação

Entrega via SIGAA

Até aqui, fizemos apenas uma breve recapitulação

Apenas para termos certeza de que podemos nos entender :-)

Agora o conteúdo do curso em si:

1. Árvores: conceito, formas de representação, árvores binárias, caminhamento em árvores binárias, árvores binárias de busca.
2. Aplicações de árvores: heaps e filas de prioridade, heapsort, Union & Find.
3. Árvores binárias balanceadas (árvores AVL e árvores Rubro Negras): definição, operações de busca, inserção e remoção de elementos.
4. Árvores B: definição, operações de busca, inserção e remoção de elementos.
5. Tabelas Hash: Funções de hashing, tratamento de colisões, representação em memória, hashing universal e hashing perfeito.
6. Grafos: conceitos fundamentais, representação de grafos (listas e matrizes de adjacências); algoritmos básicos em grafos (buscas em largura e em profundidade, caminhos mínimos).

Exercícios

Listas de exercícios ao final de cada tópico (ou sub-tópico)

- via SIGAA

Exercícios de programação com correção automática

- via julgador online (mais sobre isso na próxima aula)