Análise de algoritmos

Programação II – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

mello@ifsc.edu.br

INSTITUTO FEDERAL
Santa Catarina
Campus São José

Licenciamento

Algoritmos

Definição

Sequência bem definida de passos que transforma a entrada na saída

- Um algoritmo é dito **correto** se, para cada instância de entrada, ele finaliza com a saída correta
- Um algoritmo **incorreto**, pode não finalizar para algumas instâncias de entrada ou pode finalizar com uma saída que não a desejada

Algoritmos

(CORMEN et al., 2012)

*Se os computadores fossem **infinitamente rápidos** e com espaço infinito na memória, **qualquer algoritmo** com método **correto** para resolver um problema **seria suficiente**.*

- *Provalmente você gostaria de escrever aplicativos que respeitassem as boas práticas da engenharia de software*
- *Mas, na maioria das vezes, você usaria o método que fosse **mais fácil de implementar***

Algoritmos

(CORMEN et al., 2012)

*Se os computadores fossem **infinitamente rápidos** e com espaço infinito na memória, **qualquer algoritmo** com método **correto** para resolver um problema **seria suficiente**.*

- *Provalmente você gostaria de escrever aplicativos que respeitassem as boas práticas da engenharia de software*
- *Mas, na maioria das vezes, você usaria o método que fosse **mais fácil de implementar***

Qualquer problema é trivial desde que o tamanho da entrada seja pequeno. Mas cuidado, pois sempre estará a sua disposição algoritmos $O(2^n)$ – entenderemos isso depois!

Eficiência dos algoritmos

- Diferentes algoritmos, criados para resolver o mesmo problema, muitas vezes diferem drasticamente em sua eficiência
  - Desempenho (tempo de execução)
  - Consumo de recursos (quantidade de memória)
- A diferença pode ser até mais significativa que as diferenças relacionadas a *hardware* (i.e. *clock* do CPU) e *software* (i.e. S.O., linguagem de programação)

Algoritmos eficientes, em termos de tempo e consumo de memória, permitirão o uso sensato dos recursos que são limitados

Sequência de Fibonacci de tamanho `n`

Algoritmos iterativo e recursivo

- Use os algoritmos (iterativo e recursivo) que você desenvolveu na aula sobre recursividade

Qual algoritmo é mais rápido?[1]

1. Para uma sequência de tamanho 8
2. Para uma sequência de tamanho 32

[1] Em https://emersonmello.me/ensino/prg2/tempo tem exemplos como fazer tomada de tempo de execução de um programa em C

Obter o n-ésimo termo de uma sequência de Fibonacci

Algoritmo iterativo

```
long fib_ite(int n){
    if (n == 0){
        return 0;
    }
    long a = 0;
    long b = 1;

    for (int i = 1; i < n; ++i) {
        int k = a+b;
        a = b;
        b = k;
    }
    return b;
}
```

Algoritmo recursivo

```
long fib_rec(int n){
    if ((n == 0) || (n == 1)){
        return n;
    }
    return (fib_rec(n-1) + fib_rec(n-2));
}
```

Análise de algoritmos

Necessária para determinar o algoritmo mais adequado para resolver um dado problema

Qual o custo de um algoritmo para resolver um problema específico?

Determinar o esforço computacional (**custo**) do algoritmo em relação ao **tempo de execução** e **espaço** (quantidade de memória)

- **tempo** – analisar o número de vezes que cada parte do algoritmo é executada
- **espaço** – verificar a quantidade de memória ocupada

Método empírico

Execute os algoritmos e veja qual é o mais rápido

Medidas não confiáveis e o resultado não pode ser generalizado

- **Variáveis que podem influenciar os resultados**
  - Hardware, Sistema Operacional
  - Linguagem de programação
  - Compilador e parâmetros de compilação
  - Entradas não expressivas (i.e sequência Fibonacci de 8 *vs* 32)

Método analítico

Análise matemática

Estima o consumo de recursos (**tempo** e **memória**) em função do tamanho, de sua entrada (em função de n), oferecendo assim uma análise independente do hardware e software

- Considera somente o **comportamento assintótico**, ou seja, quando **n** for suficientemente grande
- Constantes aditivas e multiplicativas na expressão matemática não são levadas em consideração

Complexidade em tempo

- **Tempo requirido** sobre uma **dada entrada** pode ser medido pelo **número de execução** de algumas operações, como:
  - comparações
  - operações aritméticas
  - movimentação de dados, etc
- Para medir a quantidade de trabalho é necessário escolher uma operação, chamada de **operação fundamental**

Complexidade de um algoritmo

A contagem do número de vezes que a **operação fundamental** é executada

Complexidade em tempo

Melhor caso, pior caso e caso médio

- O esforço computacional (a eficiência de alguns algoritmos) depende **não apenas do tamanho da entrada**, mas também de **uma entrada em particular**
- Exemplos
  1. Buscar por um elemento em um vetor **depende do tamanho do vetor** e em **qual posição encontra-se o elemento desejado**
  2. A quantidade de trabalho para ordenar um vetor de inteiros **depende do tamanho do vetor** e do **nível de ordem** que ele se encontra

Complexidade em tempo

Melhor caso

- Uma entrada de tamanho **n** para qual o algoritmo terá o **menor tempo de execução** entre todas as entradas de mesmo tamanho
- Verifica-se qual entrada resulta no menor número de execuções da **operação fundamental**

Ex: busca por um elemento que encontra-se na 1ª posição do vetor

Não importa se o vetor tem 3 ou 10.000 elementos, o melhor caso sempre será quando o elemento desejado estiver na 1ª posição

Complexidade em tempo

Pior caso

- Uma entrada de tamanho **n** para qual o algoritmo terá o **maior tempo de execução** entre todas as entradas de mesmo tamanho
- Verifica-se qual entrada resulta no maior número de execuções da **operação fundamental**

Complexidade em tempo

Pior caso

- Uma entrada de tamanho **n** para qual o algoritmo terá o **maior tempo de execução** entre todas as entradas de mesmo tamanho
- Verifica-se qual entrada resulta no maior número de execuções da **operação fundamental**

Qual seria o pior caso na busca por um elemento em um vetor?

Complexidade em tempo

Pior caso

- Uma entrada de tamanho **n** para qual o algoritmo terá o **maior tempo de execução** entre todas as entradas de mesmo tamanho
- Verifica-se qual entrada resulta no maior número de execuções da **operação fundamental**

Qual seria o pior caso na busca por um elemento em um vetor?

a) Elemento na 1ª posição

b) Elemento na metade do vetor

c) Elemento na última posição

d) Elemento não está presente no vetor

Complexidade em tempo

Pior caso

- Uma entrada de tamanho **n** para qual o algoritmo terá o **maior tempo de execução** entre todas as entradas de mesmo tamanho
- Verifica-se qual entrada resulta no maior número de execuções da **operação fundamental**

Qual seria o pior caso na busca por um elemento em um vetor?

a) Elemento na 1ª posição

b) Elemento na metade do vetor

c) Elemento na última posição ← **aqui!**

d) Elemento não está presente no vetor ← **aqui!**

Complexidade em tempo

Caso médio

- Média do número de execuções da **operação fundamental** para todas as entradas de tamanho **n**
  - Leva em consideração a distribuição de probabilidade sobre o conjunto de entradas de tamanho **n**
  - Para simplificação, assume-se uma distribuição uniforme, de forma que todas as entradas possíveis são igualmente prováveis
- O cálculo do caso médio é pouco mais complexo

Caso médio

Exemplo: Buscar por um elemento no vetor

Considerando que todo elemento (i) possui a mesma probabilidade (p) de ser buscado que todos os demais, logo $p_i = \frac{1}{n}$

- Para o elemento na 1ª posição o algoritmo faz 1 comparação
- Para o elemento na 2ª posição o algoritmo faz 2 comparações
- Para o elemento na $n^{ésima}$ posição o algoritmo faz n comparações
- Logo, a complexidade média é:

$$
\begin{aligned}
&= \frac{1}{n} \times (1 + 2 + 3 + \ldots + n) \qquad (1)\\
&= \frac{1}{n} \times \left(\frac{n(n+1)}{2} \right)\\
&= \frac{n+1}{2}
\end{aligned}
$$

Complexidade em tempo

Melhor caso, pior caso e caso médio

- No exemplo *buscar por um elemento no vetor*, o pior caso e caso médio são lineares em **n**

Melhor caso	Pior caso	Caso médio
1	n	$\frac{n+1}{2}$

Método analítico

Função de complexidade

- Considere **f** como a função de custo ou complexidade
- $f(n)$ **mede o tempo ou espaço** para executar um algoritmo em um problema de tamanho **n**
- **Complexidade de tempo** não indica o tempo de relógio necessário, mas sim o número de vezes que as **operações fundamentais** são executadas

Complexidade em tempo

Exemplo: Encontrar o maior valor em um vetor de inteiros com n elementos

Algoritmo 1: Maior elemento no vetor

Entrada: vetor[1..n]
Saída: o maior valor contido no vetor

```
1 max ← vetor[1];
2 para i de 2 até n faça
3 |   se max < vetor[i] então
4 |   |   max ← vetor[i]
5 |   fim
6 fim
7 retorna (max)
```

- Considere como **operação fundamental** a comparação na linha 3 a qual é executada $n - 1$ vezes
- $f(n)$ consiste no número de comparações dos elementos do vetor, ou seja, $f(n) = n - 1$

Complexidade em tempo

Exemplo: Determinar se um valor `x` está presente em um vetor de inteiros com n elementos

Algoritmo 2: Determinar se o elemento está contido no vetor

Entrada: vetor[1..n], x
Saída: verdade ou falso

```
1 para i de 1 até n faça
2     se x = vetor[i] então
3         retorna (verdade)
4     fim
5 fim
6 retorna (falso)
```

- Considere como **operação fundamental** a comparação na linha 2
- **No pior caso** executa a linha 2 n **vezes**, **se encontrar na 1ª posição**, então executa **apenas** 1 **vez**
- Assim, no pior caso $f(n) = n$ (linear no tamanho de n), e no melhor caso $f(n) = 1$

Complexidade em tempo

- **Algoritmo 1:** o custo é uniforme para todos os problemas de tamanho **n**
- **Algoritmo 2:** o custo depende da posição no vetor do elemento que queira encontrar

Comparando algoritmos: complexidade de tempo

Total de operações que cada um realiza para resolver o mesmo problema

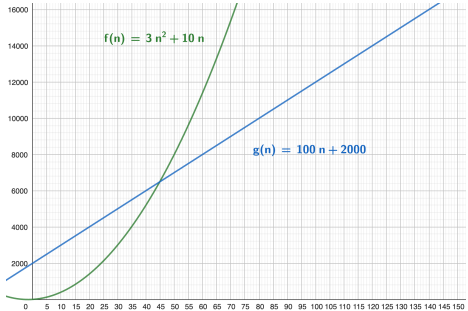
- **Algoritmo 1**: $f(n) = 3n^2 + 10n$
- **Algoritmo 2**: $g(n) = 100n + 2000$
- Para valores pequenos de n, o algoritmo 1 requer menos operações que o algoritmo 2, mas isso não se mantém valores grandes de n

Algoritmo	Total de operações (tamanho de n)			
	1	10	100	1.000
1	13	400	31.000	3.010.000
2	2.100	3.000	12.000	102.000

Comparando algoritmos: complexidade de tempo

Total de operações que cada um realiza para resolver o mesmo problema

- **Algoritmo 1**: $f(n) = 3n^2 + 10n$
- **Algoritmo 2**: $g(n) = 100n + 2000$
- Para valores pequenos de n, o algoritmo 1 requer menos operações que o algoritmo 2, mas isso não se mantém valores grandes de n



Assintótica

Concentra-se apenas nos valores enormes de n e ignora os valores pequenos

- **Complexidade assintótica** é definida pelo crescimento da complexidade para entradas suficientemente grandes
- **Comportamento assintótico** é o comportamento das funções para valores grandes de n ($n \to \infty$)

Um **algoritmo assintoticamente mais eficiente** é melhor para todas as entradas, exceto talvez para entradas relativamente pequenas

Análise assintótica

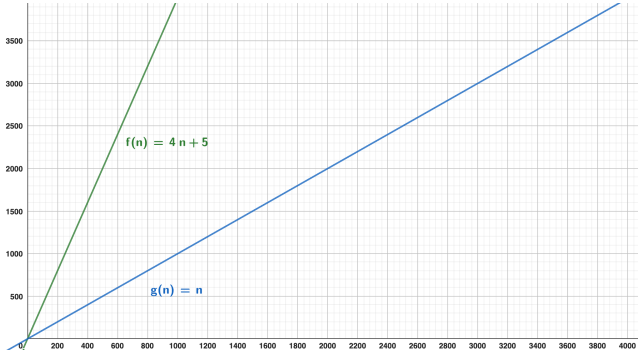
Termos inferiores e constantes multiplicativas podem ser descartadas

Considere um **algoritmo de busca cuja complexidade é dada por** $4 \times n + 5$ e desejamos simplificar para entradas suficientemente grandes

- Para um n grande, a parcela linear ($4 \times n$) torna-se dominante face à parcela constante 5
  - A diferença entre $4 \times n + 5$ e $4 \times n$ é apenas 5
  - Para $n = 100$, 5 representa apenas $1,25\%$
  - Para $n = 1000$, 5 presentaria apenas $0,125\%$
- $4 \times n$ cresce mais rapidamente do que n, mas sua razão é uma constante 4, sendo que essa razão constante é similar à que ocorre por uma mudança de escala
- Assim, $4 \times n$ **pode ser simplificado para** n

Análise assintótica

Termos inferiores e constantes multiplicativas podem ser descartadas



- $f(n)$ e $g(n)$ não são iguais, porém possuem a mesma ordem de crescimento.
- Na análise assintótica é possível simplificar $4 \times n + 5$ para n

Análise assintótica

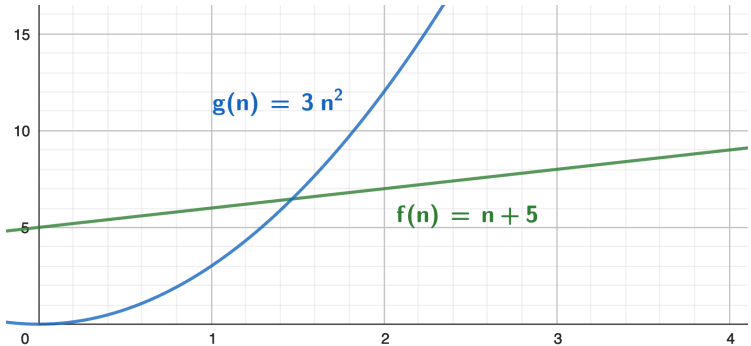
Termos inferiores e constantes multiplicativas podem ser descartadas

- Para dois algoritmos (1 e 2), considere o total de operações que cada um realiza para resolver o mesmo problema, como $f(n)$ e $g(n)$, respectivamente
  - **Algoritmo 1**: $f(n) = 3n^2 + 10n$
  - **Algoritmo 2**: $g(n) = 100n + 2000$
- $f(n)$ tem crescimento quadrático (n^2) e $g(n)$ tem crescimento linear (n)
- Logo o Algoritmo 2 é melhor que o Algoritmo 1 no pior caso

Cota assintótica superior

Função que cresce mais rapidamente do que outra função

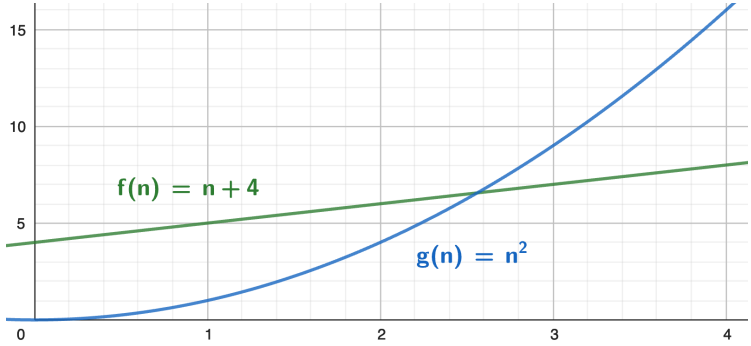
- A função quadrática $g(n) = 3 \times n^2$ cresce mais rapidamente do que a função linear $f(n) = n + 5$,
- Assim a função $g(n)$ é a **cota assintótica superior** para a função $f(n)$



Cota assintótica superior

- Basta que $g(n)$ domine $f(n)$ para um n suficientemente grande para $g(n)$ ser **cota assintótica superior**

- **Exemplo**
  - $f(n) = n + 4$ e $g(n) = n^2$,
  - Para $n = 0, 1, 2$, temos $f(n) > g(n)$, porém para $n \geq 3$ temos $f(n) < g(n)$



Cota assintótica superior

Usualmente o **cálculo da complexidade** concentra-se em **determinar a ordem de magnitude** do número de **operações fundamentais** na execução do algoritmo

- A **cota assintótica superior** é a mais comumente usada para medir a complexidade de algoritmos

Complexidade assintótica

Ordem de crescimento – notações

Para avaliar quanto o tempo de execução aumenta de acordo com o aumento do tamanho de **n**

- *O* – Big O
  - limite superior da complexidade
  - cota assintótica superior
- Ω – Ômega
  - limite inferior da complexidade
  - cota assintótica inferior
- Θ – Theta
  - limite exato da complexidade (inferior e superior)
  - limite assintótico exato

Big O

Notação assintótica: $f(n)$ é $O(g(n))$ ou f é $O(g)$

- **Definição**: $f(n)$ é $O(g(n))$ se e somente se para alguma constante $c \in \mathbb{R}_+ : c \times g(n)$ é cota assintótica superior para $f(n)$ para todo n maior que n_0

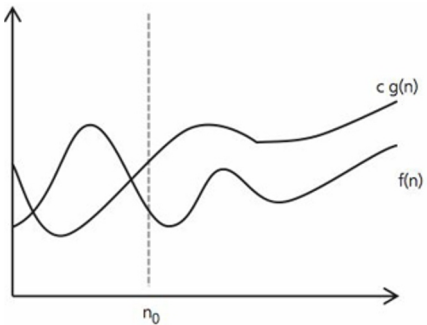
$$(\exists c \in \mathbb{R}_+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0) : f(n) \leq c \times g(n) \tag{2}$$

- Ou seja, para comparar o comportamento assintótico de f e g é necessário encontrar valores de c e n_0 tais que $f(n) \leq c \times g(n)$

Big O

Notação assintótica: $f(n)$ é $O(g(n))$ ou f é $O(g)$

- Para um **n** suficientemente grande, $g(n)$ é cota assintótica superior para $f(n)$ com um fator constante, isto é, existem constantes positivas c e n_0 tais que, para valores de n não menores que n_0, o valor de f(n) é sempre menor ou igual ao produto $c \times g(n)$



Fonte: (TOSCANI; VELOSO, 2012)

- $f(n) = 10n$ e $g(n) = n^2$
- $10n \leq c\ n^2$
- Com $c = 1$ e $n_0 = 10$, temos $10n = n^2$, acima disso $f(n)$ sempre será menor

Big O

Notação assintótica: $f(n)$ é $O(g(n))$ ou f é $O(g)$

- Para dois algoritmos (1 e 2), considere o total de operações que cada um realiza para resolver o mesmo problema, como $f(n)$ e $g(n)$, respectivamente
  - **Algoritmo 1**: $f(n) = 100n + 2000 = O(n)$
  - **Algoritmo 2**: $g(n) = 3n^2 + 10n = O(n^2)$
- Uma constante pode ser considerada como polinômio de grau 0, logo $O(n^0) = O(1)$

Big $O(n)$

Exemplos

$$
\begin{aligned}
f(n) &= 403 && \Rightarrow && f(n) = O(1) \\
f(n) &= n^2 - 1 && \Rightarrow && f(n) = O(n^2) \\
f(n) &= n^3 + n^2 + 1 && \Rightarrow && f(n) = O(n^3) \\
f(n) &= 5 + 2 \log n && \Rightarrow && f(n) = O(\log n) \\
f(n) &= 3n + 5 \log n + 20 && \Rightarrow && f(n) = O(n) \\
f(n) &= 2^n + 5n^{10} && \Rightarrow && f(n) = O(2^n)
\end{aligned}
$$

Big O

Ordem de complexidade dos algoritmos

Ordem	Descrição
$O(1)$	Sempre demora o mesmo tempo independente do tamanho de n
$O(\log n)$	Aumenta logaritmicamente conforme o tamanho de n aumenta
$O(n)$	Aumenta linearmente e proporcional ao tamanho de n
$O(n^2)$	Proporcional ao quadrado do tamanho de n
$O(2^n)$	Dobra a cada elemento da entrada (exponencial)

Lembre-se: qualquer problema é trivial desde que o tamanho da entrada seja pequeno. Mas cuidado, pois sempre estará a sua disposição algoritmos $O(2^n)$

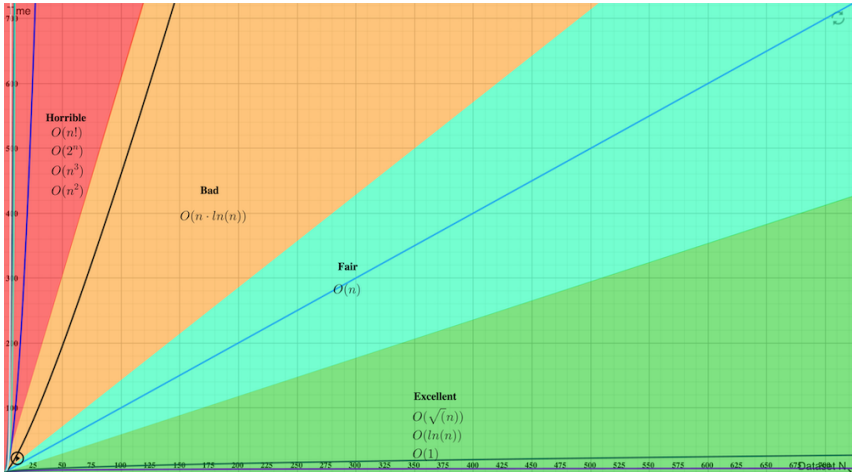
Desempenho de cada ordem de complexidade

Número de computações ou tempo de execução

n	Constante $O(1)$	Logarítmica $O(\log n)$	Linear $O(n)$	Linear Logarítmica $O(n \log n)$	Quadrática $O(n^2)$	Cúbica $O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4.096
1024	1	10	1.024	10.240	1.048.576	1.073.741.824

Notação Big O

Ordem de crescimento

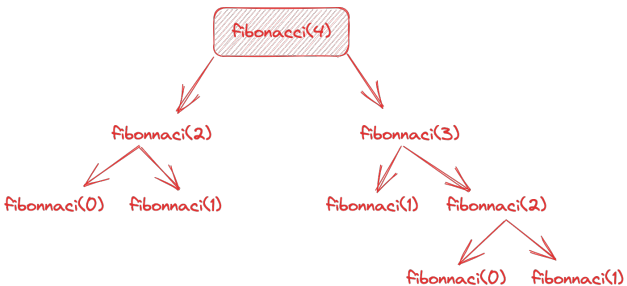


Fonte: Nicknick Mario - https://www.geogebra.org/m/HzqGHtBt

Complexidade do algoritmo: sequência de Fibonacci

Algoritmo recursivo

```
long fib_rec(int n){
    if ((n == 0) || (n == 1)){
        return n;
    }
    return (fib_rec(n-1) + fib_rec(n-2));
}
```



- Exceto a chamada recursiva, a comparação tem complexidade constante $O(1)$
- A função invoca 2 outras vezes ela mesma, até atingir o caso base, assim tem complexidade exponencial $O(2^n)$

Complexidade do algoritmo: sequência de Fibonacci

Algoritmo iterativo

```
long fib_ite(int n){
    if (n == 0){
        return 0;
    }
    long a = 0;
    long b = 1;

    for (int i = 1; i < n; ++i) {
        int k = a+b;
        a = b;
        b = k;
    }
    return b;
}
```

- É necessário conhecer apenas os dois últimos valores da sequência para calcular o próximo valor
- Todas instruções possuem complexidade constante, exceto o laço, que repete as instruções do bloco $n - 1$
- Algoritmo com complexidade linear $O(n)$

Complexidade do algoritmo: calcular fatorial

- As implementações recursivas e iterativas do cálculo do fatorial apresentam a mesma **complexidade em tempo**, linear $O(n)$
- Contudo, o algoritmo recursivo tem uma **complexidade de espaço** maior, uma vez que a cada chamada recursiva o parâmetro da função é empilhado até atingir o caso base
- O algoritmo iterativo usa a memória de forma constante

Análise de algoritmos, custos das operações

1. **Instrução simples**
   - Tempo de execução constante $O(1)$
2. **Laços**
   - Tempo de execução é igual ao tempo de execução das instruções dentro do bloco vezes o número de iterações
   - Se o total de repetições for em função de n, então é $O(n)$, se a repetição for fixo e não em função de n, então $O(1)$
3. **Laços aninhados**
   - Tempo de execução de uma instrução dentro do laço mais interno é igual o tempo de execução dessa instrução vezes o produto do número de iterações de todos os laços. No exemplo abaixo, $O(n^2)$

```
for (int i = 1; i < n; ++i) {
    for (int j = 1; j < n; ++i) {
        // instrução qualquer com O(1)
    }
}
```

Análise de algoritmos, custos das operações

4 Laços com iteração multiplicativa

- A cada iteração o contador é atualizado por meio de uma multiplicação ou divisão.
- Com o contador dobrando a cada iteração tem-se $O(\log n)$. Independente da base do logaritmo[2]

```
// por multiplicação
int contador = 1;
int n = 100;
while(contador < n){
    // instrução qualquer com O(1)
    contador = contador * 2; // indica O(log n)
}

//por divisão
for(int i = n; contador > 0; contador/=2){ // indica O(log n)
    // instrução qualquer com O(1)
}
```

[2] $\log_a n = \log_a b \times \log_b n = c \times \log_b n$, que por simplificação $O(\log n)$

Laços com iteração multiplicativa

```
// laço de fora em função de n,
int contador = n;

// controle é dividido pela metade a cada iteração, assim O (log n)
while(contador > 0){

    // laço interno em função de n, O(n)
    for(int i = 0; i < n; i++){
        // instrução qualquer com O(1)
    }
    contador /= 2;
}
```

- $O(n) + O(\log n) = O(n \log n)$

Notação Ômega – Ω

Cota assintótica inferior

Usada para definir limite inferiores para problemas. Na comparação entre classes de algoritmos pode ser útil para indicar que nenhum algoritmo terá uma ordem inferior àquela defina por Ω

- A expressão f é $O(g)$ pode ser lida como $f \leq g$
- A expressão f é $\Omega(g)$ pode ser lida como $f \geq g$
- f é $\Omega(g) \Leftrightarrow g$ é $O(f)$
- **Exemplo**
  - A função cúbica $g(n) = 7n^3 + 5$ cresce mais devagar do que a exponencial $f(n) = 2^n$ (a partir de certo ponto)

Exemplo 1 – Adição de matrizes

$A_{n \times n} + B_{n \times n} = C_{n \times n}$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \\ a_{31} + b_{31} & a_{32} + b_{32} \end{bmatrix}$$

Exemplo 1 – Adição de matrizes

$A_{m \times n} + B_{m \times n} = C_{m \times n}$

Algoritmo 3: Adição de matrizes

Entrada: a[1..m][1..n], b[1..m][1..n]
Saída: c[1..m][1..n]

```
1 para i de 1 até n faça
2     para j de 1 até n faça
3         c_ij ← a_ij + b_ij
4     fim
5 fim
6 retorna (c)
```

- Operação fundamental: $c_{ij} \leftarrow a_{ij} + b_{ij}$
- É executada n^2 vezes
- Complexidade: $O(n^2)$

Exemplo 2 – Produto de matrizes

$A_{m \times n} \times B_{n \times p} = C_{m \times p}$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \\ b_{51} & b_{42} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix}$$

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ldots + a_{in} b_{nj}$$

Exemplo 2 – Produto de matrizes

$A_{m \times n} \times B_{n \times p} = C_{m \times p}$

Algoritmo 4: Produto de matrizes

Entrada: a[1..m][1..n], b[1..n][1..p]
Saída: c[1..m][1..p]

```
1 para i de 1 até m faça
2     para j de 1 até p faça
3         c_ij ← 0
4         para k de 1 até n faça
5             c_ij ← c_ij + a_ik × b_kj
6         fim
7     fim
8 fim
9 retorna (c)
```

- Operação fundamental: $c_{ij} \leftarrow c_{ij} + a_{ij} \times b_{ij}$
- É executada n^3 vezes
- A operação $c_{ij} \leftarrow 0$ é executada n^2, por ser menor grau, é ignorada
- Complexidade: $O(n^3)$

Exemplo

$A_{3 \times 5} \times B_{5 \times 2} = C_{3 \times 2}$

Exemplo 3 – Encontrar elemento que falta no vetor

- Dado um vetor de tamanho $n - 1$, que armazena números inteiros distintos no intervalo de 1 até n (ex: $n = 5$ e $vetor = \{2, 5, 4, 1\}$), determine qual é o elemento que está faltando.
  - Neste exemplo, o elemento que está faltando é o número 3
- Uma das possíveis soluções tem complexidade de tempo $O(n)$

Notação Theta – Θ

Limite assintótico exato

Para expressar uma estimativa precisa do custo de um algoritmo

- A expressão f é $\Theta(g)$ pode ser lida como $f = g$
  - Para indicar que duas funções são da mesma ordem
- f é $\Theta(g) \Leftrightarrow g$ é $\Theta(f)$
- **Exemplo**
  - As funções quadráticas $f(n) = 7n^2 + 13$ e $g(n) = n^2 + 3$ crescem com a mesma rapidez (a partir de certo ponto)

Aplicação do assunto dessa aula no restante da disciplina

- Para cada estrutura de dados é possível executar um conjunto de operações (i.e. inserção, busca, remoção)
- Para uma mesma operação, cada estrutura de dados pode apresentar um custo computacional diferente
- Na literatura são propostos diferentes algoritmos de busca e ordenação adequados para diferentes estruturas de dados e com diferentes custos computacionais
- Algoritmos e estruturas na literatura já possuem sua complexidade calculada, assim use a notação *O* (pior caso) para escolher a estrutura e algoritmo mais adequado para o problema que pretende resolver
- Um algoritmo é considerado mais eficiente que outro se seu tempo de execução do pior caso apresentar uma ordem de crescimento mais baixa

Curiosidade

Donald Knuth em 2005

- *The Art of Computer Programming* volume I (1968), II (1969) e III (1973) de Knuth são pioneiros no estudo de algoritmos e estruturas de dados
- O volume I foi impresso usando a tecnologia do século 19 que possuía uma boa qualidade tipográfica
- Para o volume II houve uma evolução da indústria gráfica e migração para técnicas fotográficas cuja qualidade desagrou Knuth
- Em 1978 Knuth cria o TEX e exigiu a invenção de novas técnicas de programação[a]

[a] Esses *slides* foram escritos usando LATEX, um conjunto de macros para o TEX

Referências

Aula baseada em

- CORMEN, Thomas H. et al. **Algoritmos: teoria e prática**. LTC, 2012. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/9788595158092>.
- SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. **Estruturas de dados e seus algoritmos**. LTC, 2010. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/978-85-216-2995-5>.
- TOSCANI, Laira Vieira; VELOSO, Paulo A. S. **Complexidade de algoritmos**. Bookman, 2012. ISBN 9788540701397. Disponível em: <https://app.minhabiblioteca.com.br/#/books/9788540701397>.