

# Complexidade de Algoritmos (parte 2)

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados II (AE23CP)  
Engenharia de Computação  
Departamento Acadêmico de Informática (Dainf)  
Universidade Tecnológica Federal do Paraná (UTFPR)  
Campus Pato Branco

- Análise de Algoritmos
- Cálculo do Tempo de Execução

- Aula anterior: análise assintótica
  - $O$
  - $\Omega$
  - $\Theta$
- Taxas de crescimento

## Análise de Algoritmos

# Análise de Algoritmos

- Para proceder a uma análise de algoritmos e determinar as taxas de crescimento, necessitamos de um modelo de computador e das operações que executa
- Assume-se o uso de um computador tradicional, em que as instruções de um algoritmo são executadas sequencialmente
  - Com memória infinita, por simplicidade

- Repertório de instruções simples: soma, multiplicação, comparação, atribuição, etc
  - Por simplicidade e viabilidade da análise, assume-se que cada instrução demora exatamente uma unidade de tempo para ser executada, ou seja,  $O(1)$
  - Operações complexas, como inversão de matrizes e ordenação de valores, não são realizadas em uma única unidade de tempo
  - Operações complexas devem ser analisadas em partes

- Considera-se somente o algoritmo e suas entradas (de tamanho  $n$ )
- Para uma entrada de tamanho  $n$ , pode-se calcular o tempo de execução para os seguintes casos
  - Melhor caso ( $T_{melhor}(n)$ )
  - Caso médio ( $T_{media}(n)$ )
  - Pior caso ( $T_{pior}(n)$ )
  - $T_{melhor}(n) \leq T_{media}(n) \leq T_{pior}(n)$
- Atenção: para mais de uma entrada, essas funções teriam mais de um argumento

- Geralmente, utiliza-se somente a análise do pior caso ( $T_{pior}(n)$ )
  - Análise do melhor caso é de pouco interesse prático
  - O tempo médio pode ser útil, principalmente em sistemas executados rotineiramente
  - Dá mais trabalho calcular o tempo médio
  - Na análise do pior caso é verificado o máximo de esforço computacional que deve ser utilizado para o processamento da entrada



- Qual a complexidade do algoritmo abaixo?

```
int busca(int x, int v[], int n){  
1.   int i;  
2.   for (i = 0; i < n; i++)  
3.       if (x == v[i])  
4.           return i;  
5.   return -1;  
}
```

- Qual a complexidade do algoritmo abaixo?

```
int busca(int x, int v[], int n){  
1.   int i;  
2.   for (i = 0; i < n; i++)  
3.       if (x == v[i])  
4.           return i;  
5.   return -1;  
}
```

- Como a linha 1 é apenas uma declaração de variável, a mesma não entra na contagem
- Na linha 2, desconsiderando as linhas 3 e 4, podem ser realizadas até  $2n + 2$  operações (pior caso)
- Na linha 3 são realizadas um total de  $n$  comparações (o comando está dentro do laço for)

- Qual a complexidade do algoritmo abaixo?

```
int busca(int x, int v[], int n){  
1.   int i;  
2.   for (i = 0; i < n; i++)  
3.       if (x == v[i])  
4.           return i;  
5.   return -1;  
}
```

- Por fim, a linha 4 (item localizado) ou 5 (caso o item não seja localizado) será executada
  - Melhor caso e caso médio: a instrução da linha 4 será executada, que é o retorno da posição do item encontrado, ou seja, uma unidade de tempo
  - Pior caso: a instrução da linha 5 será executada

- Qual a complexidade do algoritmo abaixo?

```
int busca(int x, int v[], int n){  
1.   int i;  
2.   for (i = 0; i < n; i++)  
3.       if (x == v[i])  
4.           return i;  
5.   return -1;  
}
```

- Análise de complexidade para os três casos (melhor, médio e pior):
  - $T_{melhor}(n) = 4$  (iniciar a variável  $i$ , comparar  $i$  com  $n$ , comparar  $x$  com  $v[i]$  e retornar  $i$ )
  - $T_{media}(n) = \frac{3n}{2} + 2$  (estimativa "aproximada", ou seja, supondo que geralmente, a chave procurada esteja ao redor do meio do arranjo)
  - $T_{pior}(n) = 3n + 3$  (quando a chave não é encontrada)

- Qual a complexidade do algoritmo abaixo?

```
int busca(int x, int v[], int n){  
1.   int i;  
2.   for (i = 0; i < n; i++)  
3.       if (x == v[i])  
4.           return i;  
5.   return -1;  
}
```

- Análise de complexidade para os três casos (melhor, médio e pior):
  - Melhor caso:  $\Omega(1)$  ou  $O_{\text{melhor}}(1)$
  - Caso médio:  $O_{\text{médio}}(n)$
  - Pior caso:  $O_{\text{pior}}(n)$  ou  $O(n)$
  - Dizemos que a complexidade do algoritmo acima é de  $O(n)$  (geralmente, o que nos interessa é o pior caso)

- Idealmente, para um algoritmo qualquer de ordenação de vetores com  $n$  elementos:
  - Qual a configuração do vetor que você imagina que provavelmente resultaria no melhor tempo de execução?
  - E qual resultaria no pior tempo?
- Outro exemplo de problema: soma da subsequência máxima
  - Dada uma sequência de inteiros (possivelmente negativos)  $a_1, a_2, \dots, a_n$  encontre o valor da máxima soma de quaisquer números de elementos consecutivos
  - Se todos os inteiros forem negativos, o algoritmo deve retornar 0 como resultado da maior soma
  - Por exemplo, para a entrada -2, 11, -4, 13, -5 e -2, a resposta é 20 ( $a_2 + a_3 + a_4$ )

- Assim como o problema de ordenação, há diversos algoritmos propostos para encontrar a subsequência máxima:
  - Alguns são mostrados abaixo juntamente com seus tempos de execução ( $n$  é o tamanho da entrada):

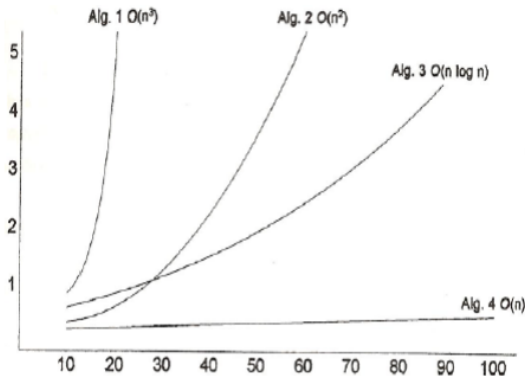
Algoritmo	1	2	3	4
tempo	$O(n^3)$	$O(n^2)$	$O(n \log n)$	$O(n)$
$n = 10$	0,00103	0,00045	0,00066	0,00034
$n = 100$	0,47015	0,01112	0,00486	0,00063
$n = 1.000$	448,77	1,1233	0,05843	0,00333
$n = 10.000$	-	111,13	0,68631	0,03042
$n = 100.000$	-	-	8,0113	0,29832

- Deve-se notar que:
  - Para entradas pequenas, todas as implementações rodam num piscar de olhos
  - Para entradas grandes, o melhor algoritmo é o 4
  - Os tempos não incluem o tempo requerido para leitura dos dados de entrada

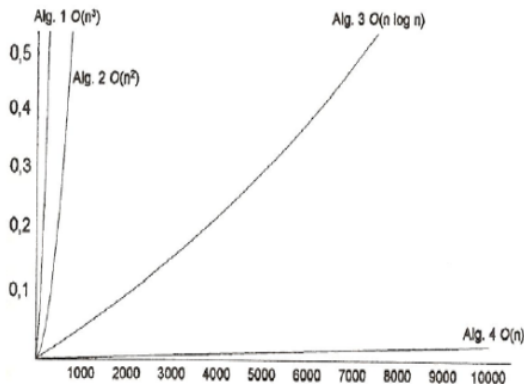


# Análise de Algoritmos

- Gráfico ( $n$  vs. milissegundos) das taxas de crescimentos dos quatro algoritmos com entradas entre 10 e 100



- Gráfico ( $n$  vs. segundos) das taxas de crescimentos dos quatro algoritmos para entradas maiores crescimento



## Cálculo do Tempo de Execução

# Cálculo do Tempo de Execução

- Existem basicamente 2 formas de estimar o tempo de execução de programas e decidir quais são os melhores:
  - Empiricamente
  - Teoricamente
- É desejável e possível estimar qual o melhor algoritmo sem ter que executá-los: função da análise de algoritmos

# Cálculo do Tempo de Execução

- Supondo que as operações simples demoram uma unidade de tempo para executar e a linguagem de programação utilizada é a C, considere o programa abaixo para calcular o resultado de

$$\sum_{i=1}^n i^3$$

- 1 Início
- 2 declare soma\_parcial numérico;
- 3 soma\_parcial  $\leftarrow$  0;
- 4 para  $i \leftarrow 0$  até  $n$  faça
- 5     soma\_parcial  $\leftarrow$  soma\_parcial +  $i*i*i$ ;
- 6     escreva(soma\_parcial);
- 7 Fim

$$\sum_{i=1}^n i^3$$

- 3 1 unidade de tempo
  - 4 1 unidade para iniciação de  $i$ ,  $n + 1$  unidades para testar se  $i = n$  e  $n$  unidades para incrementar  $i = 2n + 2$
  - 5 4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição) executada  $n$  vezes (pelo comando "para") =  $4n$  unidades
  - 6 1 unidade para escrita
- **Custo total:**  $6n + 4$ , ou seja, a função é  $O(n)$ !
    - Também podemos dizer que a complexidade do algoritmo acima é  $\Theta(n)$ , já que não há distinção entre o pior e o melhor caso

# Cálculo do Tempo de Execução

- Ter que realizar todos esses passos para cada algoritmo (principalmente algoritmos grandes) pode se tornar uma tarefa cansativa
- Em geral, como se dá a resposta em termos do *big-oh*, costuma-se desconsiderar as constantes e elementos menores dos cálculos
- No exemplo anterior:
  - A linha 3  $soma\_parcial \leftarrow 0$  é insignificante em termos de tempo
  - É desnecessário ficar contando 2, 3 ou 4 unidades de tempo na linha 5  $soma\_parcial \leftarrow soma\_parcial + i * i * i$
  - O que realmente dá a grandeza de tempo desejada é a repetição na linha 4 "para  $i \leftarrow 1$  até  $n$  faça"

- Regras para o cálculo de execução
  - Repetições: tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada

```
para  $i \leftarrow 0$  até  $n$  faça  
   $x \mathrel{+}= 1$ ;
```



- Regras para o cálculo de execução
  - Repetições: tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada
    - No exemplo abaixo são realizadas  $3n + 2$  operações (uma unidade para iniciar  $i + n * (\text{incremento na variável } i + \text{uma comparação} + \text{atribuição na variável } x) + \text{uma última comparação, que é o momento em que a variável } i \text{ atinge o valor de } n)$ , ou seja, o seu custo é  $O(n)$
    - Por mais que o operador  $+=$  seja equivalente a duas operações (uma atribuição e uma soma), o mesmo é contado como uma unidade

```
para  $i \leftarrow 0$  até  $n$  faça  
     $x += 1$ ;
```

- Regras para o cálculo de execução
  - Repetições: tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada
  - No exemplo abaixo também são realizadas  $3n + 2$  operações

```
 $i \leftarrow 0$   
enquanto  $i < n$  faça  
     $x \ += \ 1;$   
     $i \ += \ 1;$ 
```

# Cálculo do Tempo de Execução

- Regras para o cálculo de execução
  - Repetições aninhadas
    - A análise é feita de dentro para fora
    - Tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições
    - Exemplo de fragmento de código com custo de  $O(n^2)$

```
para  $i \leftarrow 0$  até  $n$  faça  
  para  $j \leftarrow 0$  até  $n$  faça  
     $k \leftarrow k + 1$ ;
```

- Regras para o cálculo de execução
  - Repetições aninhadas

```
para  $i \leftarrow 0$  até  $n$  faça  
  para  $j \leftarrow 0$  até  $n$  faça  
     $k \leftarrow k + 1$ ;
```

- Nas duas últimas linhas do código acima (laço interno), são realizadas  $4n + 2$  operações: uma atribuição para a variável  $j + n *$  (uma atualização de  $j +$  mais uma comparação entre  $i$  e  $n +$  uma atribuição na variável  $k +$  uma soma na variável  $k$ )
- A operação acima é realizada  $n$  vezes, ou seja, o total de operações no fragmento de código acima é  $n * (4n + 2 + 2) + 2 = 4n^2 + 4n + 2$

- Regras para o cálculo de execução
  - Comandos consecutivos
    - É a soma dos tempos de cada um, o que pode significar o máximo entre eles
    - O exemplo abaixo é  $O(n^2)$  apesar da primeira repetição ser  $O(n)$

```
para  $i \leftarrow 0$  até  $n$  faça  
     $k \leftarrow 0$ ;  
    para  $j \leftarrow 0$  até  $n$  faça  
         $k \leftarrow k + 1$ ;
```

- Regras para o cálculo de execução
  - Se... então... senão:
    - Para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste (então) mais o tempo do senão
    - Em outras palavras, a complexidade é comando que leva o maior tempo
    - O exemplo abaixo é  $O(n)$  no pior caso e  $\Omega(1)$  no melhor caso:  
se  $i < j$   
então  $i \leftarrow i + 1$   
senão para  $k \leftarrow 0$  até  $n$  faça  
     $i \leftarrow i * k;$
  - Chamadas de sub-rotinas: uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou

- **Exercício 1:** Estime quantas unidades de tempo são necessárias para rodar o algoritmo abaixo:

```
1 void função1(int v[], int n){
2     int i, j, auxA, auxB;
3     i = 1;
4     while (i <= n){
5         v[i - 1] += 0;
6         i = i + 1;
7     }
8     for (i = 0; i < n; i++)
9         for (j = 0; j < n; j++)
10             v[i] += v[j] + i + j;
11 }
```

## • Resolução do exercício 1:

- 1 Linha 3: inicialização de  $i$ : 1 operação
- 2 Linhas 4-7:  $5n + 1$ 
  - $n * (1 \text{ comparação} + 1 \text{ subtração } (v[i - 1]) + 1 \text{ atribuição com soma } (+) \text{ em } v + 1 \text{ atribuição e 1 soma em } i)$ :  $5n$  operações
  - Última comparação (quando  $i > n$ ): 1 operação
- 3 Linhas 9-10 (*loop interno*):  $5n + 1$ 
  - Inicialização de  $j$ : 1 operação
  - $n * (1 \text{ comparação} + 3 \text{ operações em } A \text{ (1 atribuição e duas somas)} + 1 \text{ incremento em } j)$ :  $5n$  operações
  - Última comparação (quando  $j > n$ ): 1 operação
- 4 Linhas 8-10:  $5n^2 + 4n + 2$ 
  - Inicialização de  $i$ : 1 operação
  - $n * (1 \text{ comparação} + 5n + 2 \text{ (for interno)} + 1 \text{ atribuição em } i)$ :  $5n^2 + 4n$  operações
  - Última comparação (quando  $i > n$ ): 1 operação
- 5 Soma dos itens 1, 2 e 3:  $1 + 5n + 1 + 5n^2 + 4n + 2 = 5n^2 + 9n + 4$ 
  - Complexidade:  $O(n^2)$



- **Exercício 2:** Estime quantas unidades de tempo são necessárias para rodar o algoritmo abaixo:

```
1 void função2(int n){  
2     int i, j, auxA, auxB;  
3     i = 0;  
4     auxA = 0;  
5     auxB = 0;  
6     while (i <= n){  
7         auxA += 1;  
8         i = i + 1;  
9     }  
10    for (i = 0; i <= n; i++)  
11        for (j = 0; j <= n; j++)  
12            auxB += (auxA - i + j);  
13 }
```

## • Resolução do exercício 2:

① Linhas 3–9:  $4n + 8$

- Inicialização de  $i$ , auxA, AuxB: 3 operações
- $(n + 1) * (1 \text{ comparação} + 1 \text{ atribuição em auxA} + 1 \text{ atribuição e 1 soma em } i)$ :  $4n + 4$  operações
- Última comparação (quando  $i > n$ ): 1 operação

② Linhas 11–12 (*loop* interno):  $5n + 7$

- Inicialização de  $j$ : 1 operação
- $(n + 1) * (1 \text{ comparação} + 3 \text{ operações em auxB (1 atribuição e duas somas)} + 1 \text{ incremento em } j)$ :  $5n + 5$  operações
- Última comparação (quando  $j > n$ ): 1 operação

③ Linhas 10–12:  $5n^2 + 14n + 11$

- Inicialização de  $i$ : 1 operação
- $(n + 1) * (1 \text{ comparação} + 5n + 7 \text{ (for interno)} + 1 \text{ atribuição em } i)$ :  $5n^2 + 14n + 9$  operações
- Última comparação (quando  $i > n$ ): 1 operação

④ Soma dos itens 1 e 3:  $5n^2 + 18n + 19$

- Complexidade:  $O(n^2)$



Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S.  
*Algoritmos: teoria e prática.*  
Elsevier, 2012.



Horowitz, E., Sahni, S. Rajasekaran, S.  
*Computer Algorithms.*  
Computer Science Press, 1998.



Rosa, J. L. G.  
Análise de Algoritmos - parte 1. SCC-201 – Introdução à  
Ciência da Computação II.  
*Slides.* Ciência de Computação. ICMC/USP, 2016.



Ziviani, N.  
*Projeto de Algoritmos - com implementações em Java e C++.*  
Thomson, 2007.