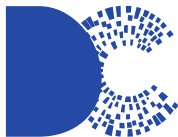




UNIVERSIDADE  
FEDERAL DE  
SERGIPE



DEPARTAMENTO  
DE COMPUTAÇÃO

# Eficiência computacional e notação $O$

## Projeto e Análise de Algoritmos

Bruno Prado

Departamento de Computação / UFS

# Introdução

- ▶ O que é eficiência?
  - ▶ Tempo ou esforço empregado para realizar algo
  - ▶ Otimização do uso dos recursos

$\uparrow \textit{Eficiência} \longleftrightarrow \textit{Tempo} \downarrow$

$\uparrow \textit{Eficiência} \longleftrightarrow \textit{Recursos} \downarrow$

# Introdução

- ▶ Qual a história e por que era importante?
  - ▶ Os recursos computacionais eram muito limitados
  - ▶ Grande consumo de potência e uso compartilhado



# Introdução

- ▶ Por que hoje é importante?
  - ▶ Restrições de custo
  - ▶ Baixo consumo de potência

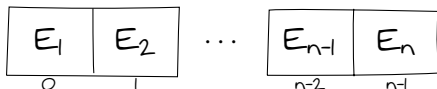


# Introdução

- ▶ Quais os tipos de eficiência ou de complexidade computacional?
  - ▶ Tempo
    - ▶ Número de passos executados
  - ▶ Espaço
    - ▶ Tamanho da alocação em memória

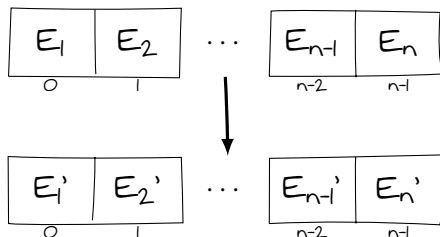
# Complexidade de tempo

- ▶ Indicador de eficiência de execução do algoritmo
  - ▶ Métrica: número de passos executados
  - ▶ Problema: ordenação de sequência com  $n$  números  $E_1, E_2, \dots, E_{n-1}, E_n$  para gerar uma sequência ordenada  $E'_1, E'_2, \dots, E'_{n-1}, E'_n$



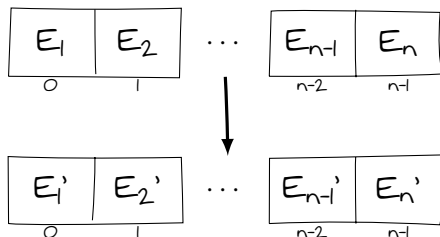
# Complexidade de tempo

- ▶ Indicador de eficiência de execução do algoritmo
  - ▶ Métrica: número de passos executados
  - ▶ Problema: ordenação de sequência com  $n$  números  $E_1, E_2, \dots, E_{n-1}, E_n$  para gerar uma sequência ordenada  $E'_1, E'_2, \dots, E'_{n-1}, E'_n$



# Complexidade de tempo

- ▶ Indicador de eficiência de execução do algoritmo
  - ▶ Métrica: número de passos executados
  - ▶ Problema: ordenação de sequência com  $n$  números  $E_1, E_2, \dots, E_{n-1}, E_n$  para gerar uma sequência ordenada  $E'_1, E'_2, \dots, E'_{n-1}, E'_n$



Quantos passos são realizados?



# Complexidade de tempo

## ► Ordenação por seleção

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de ordenação por seleção
4 void selection_sort(uint32_t* V, uint32_t n) {
5     for(uint32_t i = 0; i < n - 1; i++) {
6         uint32_t min = i;
7         for(uint32_t j = i + 1; j < n; j++)
8             if(V[j] < V[min]) min = j;
9         if(i != min) trocar(&V[i], &V[min]);
10    }
11 }
```

# Complexidade de tempo

## ► Ordenação por seleção

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de ordenação por seleção
4 void selection_sort(uint32_t* V, uint32_t n) {
5     for(uint32_t i = 0; i < n - 1; i++) {
6         uint32_t min = i;
7         for(uint32_t j = i + 1; j < n; j++)
8             if(V[j] < V[min]) min = j;
9         if(i != min) trocar(&V[i], &V[min]);
10    }
11 }
```

$$\begin{aligned}\text{Número de passos} &= (n-1) + (n-2) + \dots + 2 + 1 \\ &= \frac{(n-1)[1 + (n-1)]}{2} \approx n^2\end{aligned}$$

# Complexidade de tempo

## ► Ordenação por inserção

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de ordenação por inserção
4 void insertion_sort(uint32_t* V, uint32_t n) {
5     for(uint32_t i = 1; i < n; i++)
6         for(uint32_t j = i; j > 0 && V[j - 1] > V[j];
7             j--)
8             trocar(&V[j], &V[j - 1]);
9 }
```

# Complexidade de tempo

## ► Ordenação por inserção

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de ordenação por inserção
4 void insertion_sort(uint32_t* V, uint32_t n) {
5     for(uint32_t i = 1; i < n; i++)
6         for(uint32_t j = i; j > 0 && V[j - 1] > V[j];
7             j--)
8             trocar(&V[j], &V[j - 1]);
9 }
```

$$n < \text{Número de passos} < n^2$$

# Complexidade de tempo

- ▶ Como calcular a quantidade de passos?
  - ▶ Análise depende somente do tamanho da entrada  $n$
  - ▶ Demais trechos do código são constantes

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de exemplo
4 void exemplo(uint32_t n) {
5     c1();
6     for(uint32_t i = 0; i < n; i++)
7         c2();
8     for(uint32_t j = 0; j < n; j++)
9         c3();
10    for(uint32_t k = 0; k < n; k++)
11        c4();
12 }
```

# Complexidade de tempo

- ▶ Como calcular a quantidade de passos?
  - ▶ Expressão em função do valor de  $n$
  - ▶ Sub-rotinas  $c_1$ ,  $c_2$ ,  $c_3$  e  $c_4$  não dependem de  $n$

$$\begin{aligned} exemplo(n) &= c_1 + n \times \{c_2 + n \times [c_3 + (c_4 \times n)]\} \\ &= c_1 + c_2 \times n + c_3 \times n^2 + c_4 \times n^3 \end{aligned}$$

# Complexidade de tempo

- ▶ Como obter o tempo consumido?

- ▶ Entrada de tamanho 1.000

- ▶ Valores de

$c1 = 200\text{ ns}$ ,  $c2 = 150\text{ ns}$ ,  $c3 = 250\text{ ns}$  e  $c4 = 100\text{ ns}$

$$\begin{aligned} \text{exemplo}(1000) &= 200\text{ ns} + 150\text{ ns} \times 10^3 + 250\text{ ns} \times 10^6 + 100\text{ ns} \times 10^9 \\ &= (0,0000002 + 0,00015 + 0,25 + 100) \times 10^9\text{ ns} \\ &= 100,2501502 \times 10^9\text{ ns} \\ &\approx 100\text{ s} \end{aligned}$$

# Complexidade de tempo

- ▶ Como obter o tempo consumido?
  - ▶ Quanto maior o valor do tamanho da entrada  $n$ , maior é o domínio do fator de maior grau da função
  - ▶ Para um valor de  $n$  suficientemente grande  $n > n_0$

$$\text{exemplo}(n) \leq g(n)$$

$$g(n) = c \times n^3$$



# Complexidade de tempo

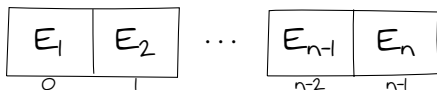
## ► Análise assintótica

- Valores das constantes dependem da máquina
- Com  $n \rightarrow \infty$  se analisa a ordem das funções

$$\lim_{n \rightarrow \infty} \frac{\text{exemplo}(n)}{g(n)} = \begin{cases} 0 & \text{exemplo}(n) < g(n) \\ k & \text{exemplo}(n) = g(n) \\ \infty & \text{exemplo}(n) > g(n) \end{cases}$$

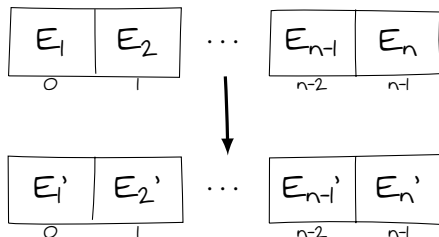
# Complexidade de espaço

- ▶ Indicador de eficiência de memória do algoritmo
  - ▶ Métrica: tamanho da alocação em memória
  - ▶ Problema: ordenação de sequência com  $n$  números  $E_1, E_2, \dots, E_{n-1}, E_n$  para gerar uma sequência ordenada  $E'_1, E'_2, \dots, E'_{n-1}, E'_n$



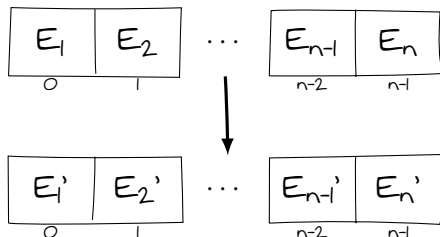
# Complexidade de espaço

- ▶ Indicador de eficiência de memória do algoritmo
  - ▶ Métrica: tamanho da alocação em memória
  - ▶ Problema: ordenação de sequência com  $n$  números  $E_1, E_2, \dots, E_{n-1}, E_n$  para gerar uma sequência ordenada  $E'_1, E'_2, \dots, E'_{n-1}, E'_n$



# Complexidade de espaço

- ▶ Indicador de eficiência de memória do algoritmo
  - ▶ Métrica: tamanho da alocação em memória
  - ▶ Problema: ordenação de sequência com  $n$  números  $E_1, E_2, \dots, E_{n-1}, E_n$  para gerar uma sequência ordenada  $E'_1, E'_2, \dots, E'_{n-1}, E'_n$



Quantas posições de memória são utilizadas?

# Complexidade de espaço

- ▶ Como calcular a memória alocada?
  - ▶ Expressão em função do valor de entrada  $n$
  - ▶ Constantes dependem do tamanho do dado

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de ordenação por inserção
4 void insertion_sort(uint32_t* V, uint32_t n) {
5     for(uint32_t i = 1; i < n; i++)
6         for(uint32_t j = i; j > 0 && V[j - 1] > V[j];
7             j--)
8             trocar(&V[j], &V[j - 1]);
9 }
```

# Complexidade de espaço

- ▶ Como calcular a memória alocada?
  - ▶ Expressão em função do valor de entrada  $n$
  - ▶ Constantes dependem do tamanho do dado

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de ordenação por inserção
4 void insertion_sort(uint32_t* V, uint32_t n) {
5     for(uint32_t i = 1; i < n; i++)
6         for(uint32_t j = i; j > 0 && V[j - 1] > V[j];
7             j--)
8             trocar(&V[j], &V[j - 1]);
9 }
```

$$\text{insertion\_sort}(n) = c_{\text{uint32\_t}} \times n + c_{\text{uint32\_t}} \times 3$$

# Complexidade de espaço

- ▶ Como calcular a memória alocada?
  - ▶ Expressão em função do valor de entrada  $n$
  - ▶ Constantes dependem do tamanho do dado

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de exemplo
4 void exemplo(uint32_t* V, uint32_t n) {
5     V = (uint32_t*)(malloc(n * sizeof(uint32_t)));
6     for(uint32_t i = 0; i < n; i++)
7         V[i] = rand();
8 }
```

# Complexidade de espaço

- ▶ Como calcular a memória alocada?
  - ▶ Expressão em função do valor de entrada  $n$
  - ▶ Constantes dependem do tamanho do dado

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de exemplo
4 void exemplo(uint32_t* V, uint32_t n) {
5     V = (uint32_t*)(malloc(n * sizeof(uint32_t)));
6     for(uint32_t i = 0; i < n; i++)
7         V[i] = rand();
8 }
```

$$\text{exemplo}(n) = c_{\text{uint32\_t}} \times n + c_{\text{uint32\_t}} \times 2$$



# Complexidade de espaço

- ▶ Como calcular a memória alocada?
  - ▶ Quanto maior o valor do tamanho da entrada  $n$ , maior é o domínio do fator de maior grau da função
  - ▶ Para um valor de  $n$  suficientemente grande  $n > n_0$

$$\text{exemplo}(n) \leq g(n)$$

$$g(n) = c \times n$$

# Complexidade de espaço

- ▶ Análise assintótica

- ▶ Valores das constantes dependem da máquina
- ▶ Com  $n \rightarrow \infty$  se analisa a ordem das funções

$$\lim_{n \rightarrow \infty} \frac{\text{exemplo}(n)}{g(n)} = \begin{cases} 0 & \text{exemplo}(n) < g(n) \\ k & \text{exemplo}(n) = g(n) \\ \infty & \text{exemplo}(n) > g(n) \end{cases}$$

# Ordem de crescimento

- Classes de complexidade para entrada  $n$

$n$	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$	$n!$
$10^1$	3,3	$10^1$	$3,3 \times 10^1$	$10^2$	$10^3$	$10^3$	$3,6 \times 10^6$
$10^2$	6,6	$10^2$	$6,6 \times 10^2$	$10^4$	$10^6$	$1,3 \times 10^{30}$	$9,3 \times 10^{157}$
$10^3$	10	$10^3$	$1,0 \times 10^4$	$10^6$	$10^9$	-	-
$10^4$	13	$10^4$	$1,3 \times 10^5$	$10^8$	$10^{12}$	-	-
$10^5$	17	$10^5$	$1,7 \times 10^6$	$10^{10}$	$10^{15}$	-	-
$10^6$	20	$10^6$	$2,0 \times 10^7$	$10^{12}$	$10^{18}$	-	-

# Exemplo

- ▶ Calcular a complexidade de tempo e de espaço do algoritmo fatorial
  - ▶ Descrever sua implementação iterativa
  - ▶ Tudo deve ser claramente justificado

$$Fatorial(n) = \begin{cases} 1 & n = 0 \\ n \times Fatorial(n - 1) & n > 0 \end{cases}$$

# Notação $O$

- ▶ Necessidade de formalização da complexidade dos algoritmos
  - ▶ Notação matemática
  - ▶ Análise assintótica
- ▶ Notações para melhor caso ( $\Omega$ ), pior caso ( $O$ ) e caso médio ( $\Theta$ )
  - ▶ Definições e aplicações

# Notação O

- ▶ Função de busca sequencial
  - ▶ Descrita pela equação  $busca(n) = c_A + c_B \times n$

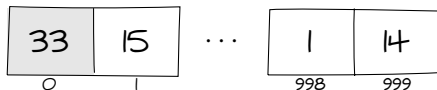
```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de busca
4 int32_t busca(int32_t elem, int32_t V[], uint32_t n) {
5     int32_t r = -1;
6     for(uint32_t i = 0; r == -1 && i < n; i++)
7         if(V[i] == elem)
8             r = i;
9     return r;
10 }
```

# Melhor caso

- ▶ O que é a análise de melhor caso de um algoritmo?
  - ▶ Situação com menor número de passos realizados
  - ▶ Estabelece um limitante inferior ou melhor caso

# Melhor caso

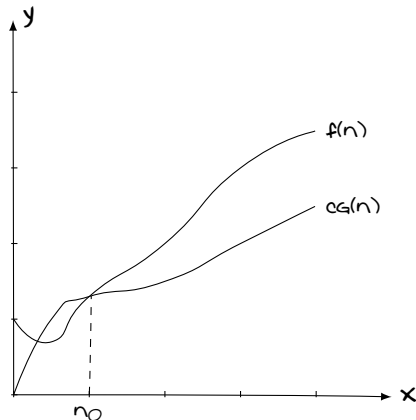
- ▶ O que é a análise de melhor caso de um algoritmo?
  - ▶ Situação com menor número de passos realizados
  - ▶ Estabelece um limitante inferior ou melhor caso
- ▶ Busca sequencial pelo elemento 33
  - ▶ Primeira ocorrência
  - ▶ Vetor possui 1.000 elementos sem repetições





# Melhor caso

- ▶ Análise de melhor caso da busca sequencial
  - ▶ Existem constantes positivas  $c$  e  $n_0$  tal que  $0 \leq cg(n) \leq busca(n)$ , para todo  $n \geq n_0$ , logo  $\Omega(busca(n)) = \Omega(g(n)) = \Omega(c_A + c_B) = c_{MC}$



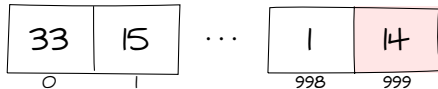
$$f(n) = \Omega(g(n))$$

# Pior caso

- ▶ O que é a análise de pior caso de um algoritmo?
  - ▶ Descreve a situação com maior número de passos
  - ▶ Estabelece um limitante superior

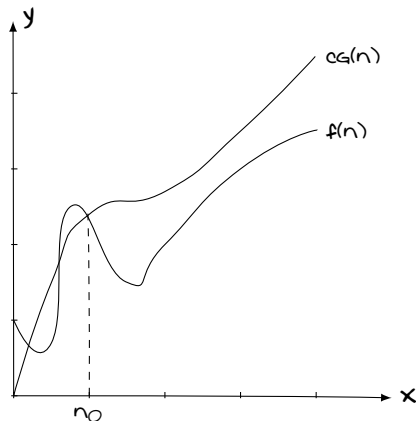
## Pior caso

- ▶ O que é a análise de pior caso de um algoritmo?
  - ▶ Descreve a situação com maior número de passos
  - ▶ Estabelece um limitante superior
- ▶ Busca sequencial pelo elemento 14
  - ▶ Última ocorrência
  - ▶ Vetor possui 1.000 elementos sem repetições



## Pior caso

- ▶ Análise de pior caso da busca sequencial
  - ▶ Existem constantes positivas  $c$  e  $n_0$  tal que  $0 \leq busca(n) \leq cg(n)$ , para todo  $n \geq n_0$ , logo  $O(busca(n)) = O(cg(n)) = O(c_A + c_B \times n) = c_{PC} \times n$



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

# Notação $O$

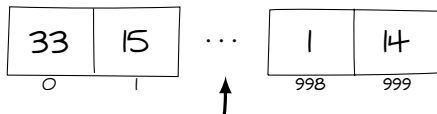
- ▶ Propriedades da notação  $O$ 
  - ▶ Termos constantes:  $O(c) = O(1)$
  - ▶ Multiplicação por constantes:  $O(c \times f(n)) = O(f(n))$
  - ▶ Adição:  $O(f_1(n)) + O(f_2(n)) = O(|f_1(n)| + |f_2(n)|)$
  - ▶ Produto:  $O(f_1(n)) \times O(f_2(n)) = O(f_1(n) \times f_2(n))$

# Caso médio

- ▶ Não confundir com caso prático ou real
  - ▶ Observa o comportamento real do algoritmo
  - ▶ Utiliza dados estatísticos

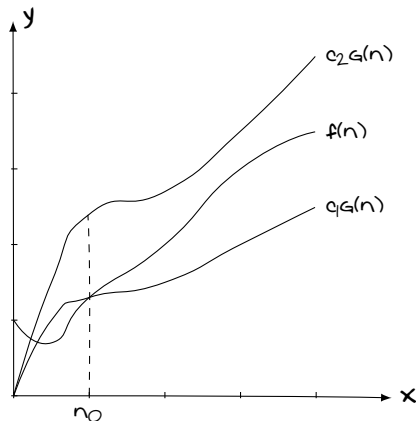
# Caso médio

- ▶ Não confundir com caso prático ou real
  - ▶ Observa o comportamento real do algoritmo
  - ▶ Utiliza dados estatísticos
- ▶ Busca sequencial por um elemento qualquer
  - ▶ São executadas na busca entre 1 e  $n$  iterações
  - ▶ Vetor possui 1.000 elementos sem repetições



# Caso médio

- ▶ Análise de caso médio da busca sequencial
  - ▶ Existem constantes positivas  $c$  e  $n_0$  tal que  $0 \leq c_1 g(n) \leq busca(n) \leq c_2 g(n)$ , para todo  $n \geq n_0$ , logo  $\Omega(c_{MC}) \leq busca(n) \leq O(n)$



$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$



# Caso médio

- Ordem exata de execução de um algoritmo  $f(n)$

$$f(n) = \Omega(c_1 g(n)) \text{ e } f(n) = O(c_2 g(n))$$

↓

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

## Exemplo

- Calcule a complexidade de tempo e espaço do código abaixo, utilizando as 3 notações vistas

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Procedimento de exemplo
4 void exemplo(uint32_t n) {
5     int a[] = (int*)(malloc((n*n+10) * sizeof(int)));
6     for(int i = 0; i < 10; i++)
7         a[i] = 1;
8     for(int i = 0; i < n; i++) {
9         int b = 3;
10        for(int j = 0; j < n; j++) {
11            a[i][j] = b * a[i][j];
12            for(int k = 0; k < 10; k++)
13                a[i][j] = a[i][j] * a[k];
14        }
15    }
16    for(int i = n; i < n * n; i++)
17        a[i] = a[i] + 2;
18 }
```

# Exercícios

- ▶ Explique porque é utilizada a análise assintótica e qual a importância de utilização da notação  $O$
- ▶ Descreva com suas palavras o que você entende por pior caso, melhor caso e caso médio
- ▶ Verifique como calcular a complexidade de algoritmos recursivos