

# MC-202

## Noções de Eficiência de Algoritmos

Lehilton Pedrosa

Universidade Estadual de Campinas

Segundo semestre de 2021

## Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

# Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

# Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Depende...

# Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Depende...

- do computador onde ele for rodado

# Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Depende...

- do computador onde ele for rodado
  - computador rápido vs lento

# Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Depende...

- do computador onde ele for rodado
  - computador rápido vs lento
- da posição de  $x$  no vetor

# Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }
```

Depende...

- do computador onde ele for rodado
  - computador rápido vs lento
- da posição de  $x$  no vetor
  - no melhor caso, a linha 4 é executada 1 vez



# Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Depende...

- do computador onde ele for rodado
  - computador rápido vs lento
- da posição de  $x$  no vetor
  - no melhor caso, a linha 4 é executada 1 vez
  - no pior caso, a linha 4 é executada  $n$  vezes

# Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Depende...

- do computador onde ele for rodado
  - computador rápido vs lento
- da posição de  $x$  no vetor
  - no melhor caso, a linha 4 é executada 1 vez
  - no pior caso, a linha 4 é executada  $n$  vezes
- do valor de  $n$

# Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Depende...

- do computador onde ele for rodado
  - computador rápido vs lento
- da posição de  $x$  no vetor
  - no melhor caso, a linha 4 é executada 1 vez
  - no pior caso, a linha 4 é executada  $n$  vezes
- do valor de  $n$ 
  - $n = 10$  vs  $n = 10.000$

# Busca sequencial e consumo de tempo

Queremos analisar algoritmos:

# Busca sequencial e consumo de tempo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado

# Busca sequencial e consumo de tempo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de  $n$  (a quantidade de dados)

# Busca sequencial e consumo de tempo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de  $n$  (a quantidade de dados)

Em geral, queremos analisar o **pior** caso do algoritmo

# Busca sequencial e consumo de tempo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de  $n$  (a quantidade de dados)

Em geral, queremos analisar o **pio**r caso do algoritmo

- A análise do **melhor** caso pode ser interessante, mas é rara



# Busca sequencial e consumo de tempo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de  $n$  (a quantidade de dados)

Em geral, queremos analisar o **pior** caso do algoritmo

- A análise do **melhor** caso pode ser interessante, mas é rara
- A análise do caso **médio** é mais difícil

# Busca sequencial e consumo de tempo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de  $n$  (a quantidade de dados)

Em geral, queremos analisar o **pior** caso do algoritmo

- A análise do **melhor** caso pode ser interessante, mas é rara
- A análise do caso **médio** é mais difícil
  - Normalmente é uma análise probabilística

# Busca sequencial e consumo de tempo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de  $n$  (a quantidade de dados)

Em geral, queremos analisar o **pior** caso do algoritmo

- A análise do **melhor** caso pode ser interessante, mas é rara
- A análise do caso **médio** é mais difícil
  - Normalmente é uma análise probabilística
  - Precisamos fazer suposições sobre os dados de entrada

# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }
```

Consumo de tempo por linha no pior caso:

# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável)

# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável)
- Linha 3: tempo  $c_3$  (atribuições, acessos e comparação)

# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável)
- Linha 3: tempo  $c_3$  (atribuições, acessos e comparação)
  - No pior caso, essa linha é executada  $n + 1$  vezes



# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável)
- Linha 3: tempo  $c_3$  (atribuições, acessos e comparação)
  - No pior caso, essa linha é executada  $n + 1$  vezes
- Linha 4: tempo  $c_4$  (acessos, comparação e if)

# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {  
2     int i;  
3     for (i = 0; i < n; i++)  
4         if (v[i] == x)  
5             return i;  
6     return -1;  
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável)
- Linha 3: tempo  $c_3$  (atribuições, acessos e comparação)
  - No pior caso, essa linha é executada  $n + 1$  vezes
- Linha 4: tempo  $c_4$  (acessos, comparação e if)
  - No pior caso, essa linha é executada  $n$  vezes

# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável)
- Linha 3: tempo  $c_3$  (atribuições, acessos e comparação)
  - No pior caso, essa linha é executada  $n + 1$  vezes
- Linha 4: tempo  $c_4$  (acessos, comparação e if)
  - No pior caso, essa linha é executada  $n$  vezes
- Linha 5: tempo  $c_5$  (acesso e return)

# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável)
- Linha 3: tempo  $c_3$  (atribuições, acessos e comparação)
  - No pior caso, essa linha é executada  $n + 1$  vezes
- Linha 4: tempo  $c_4$  (acessos, comparação e if)
  - No pior caso, essa linha é executada  $n$  vezes
- Linha 5: tempo  $c_5$  (acesso e return)
- Linha 6: tempo  $c_6$  (return)

## Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável)
- Linha 3: tempo  $c_3$  (atribuições, acessos e comparação)
  - No pior caso, essa linha é executada  $n + 1$  vezes
- Linha 4: tempo  $c_4$  (acessos, comparação e if)
  - No pior caso, essa linha é executada  $n$  vezes
- Linha 5: tempo  $c_5$  (acesso e return)
- Linha 6: tempo  $c_6$  (return)

O tempo de execução é menor ou igual a

# Busca sequencial e consumo de tempo

```
1 int busca(int *v, int n, int x) {
2     int i;
3     for (i = 0; i < n; i++)
4         if (v[i] == x)
5             return i;
6     return -1;
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo  $c_2$  (declaração de variável)
- Linha 3: tempo  $c_3$  (atribuições, acessos e comparação)
  - No pior caso, essa linha é executada  $n + 1$  vezes
- Linha 4: tempo  $c_4$  (acessos, comparação e if)
  - No pior caso, essa linha é executada  $n$  vezes
- Linha 5: tempo  $c_5$  (acesso e return)
- Linha 6: tempo  $c_6$  (return)

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador



## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

Sejam  $a := c_2 + c_3 + c_5 + c_6$ ,  $b := c_3 + c_4$  e  $d := a + b$

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

Sejam  $a := c_2 + c_3 + c_5 + c_6$ ,  $b := c_3 + c_4$  e  $d := a + b$

Se  $n \geq 1$ , temos que o tempo de execução é menor ou igual a

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

Sejam  $a := c_2 + c_3 + c_5 + c_6$ ,  $b := c_3 + c_4$  e  $d := a + b$

Se  $n \geq 1$ , temos que o tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

Sejam  $a := c_2 + c_3 + c_5 + c_6$ ,  $b := c_3 + c_4$  e  $d := a + b$

Se  $n \geq 1$ , temos que o tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 = c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n$$

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

Sejam  $a := c_2 + c_3 + c_5 + c_6$ ,  $b := c_3 + c_4$  e  $d := a + b$

Se  $n \geq 1$ , temos que o tempo de execução é menor ou igual a

$$\begin{aligned}c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n\end{aligned}$$

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

Sejam  $a := c_2 + c_3 + c_5 + c_6$ ,  $b := c_3 + c_4$  e  $d := a + b$

Se  $n \geq 1$ , temos que o tempo de execução é menor ou igual a

$$\begin{aligned}c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \leq a \cdot n + b \cdot n\end{aligned}$$

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

Sejam  $a := c_2 + c_3 + c_5 + c_6$ ,  $b := c_3 + c_4$  e  $d := a + b$

Se  $n \geq 1$ , temos que o tempo de execução é menor ou igual a

$$\begin{aligned} c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \leq a \cdot n + b \cdot n = d \cdot n \end{aligned}$$



## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

Sejam  $a := c_2 + c_3 + c_5 + c_6$ ,  $b := c_3 + c_4$  e  $d := a + b$

Se  $n \geq 1$ , temos que o tempo de execução é menor ou igual a

$$\begin{aligned}c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \leq a \cdot n + b \cdot n = d \cdot n\end{aligned}$$

Isto é, o crescimento do tempo é linear em  $n$

## Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Cada  $c_i$  não depende de  $n$ , depende apenas do computador

- Leva um tempo constante

Sejam  $a := c_2 + c_3 + c_5 + c_6$ ,  $b := c_3 + c_4$  e  $d := a + b$

Se  $n \geq 1$ , temos que o tempo de execução é menor ou igual a

$$\begin{aligned} c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \leq a \cdot n + b \cdot n = d \cdot n \end{aligned}$$

Isto é, o crescimento do tempo é linear em  $n$

- Se  $n$  dobra, o tempo de execução praticamente dobra

## Notação Assintótica

Como vimos, existe uma constante  $d$  tal que, para  $n \geq 1$ ,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

## Notação Assintótica

Como vimos, existe uma constante  $d$  tal que, para  $n \geq 1$ ,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

$d$  não interessa tanto, depende apenas do computador...

## Notação Assintótica

Como vimos, existe uma constante  $d$  tal que, para  $n \geq 1$ ,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

$d$  não interessa tanto, depende apenas do computador...

- Estamos preocupados em estimar

## Notação Assintótica

Como vimos, existe uma constante  $d$  tal que, para  $n \geq 1$ ,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

$d$  não interessa tanto, depende apenas do computador...

- Estamos preocupados em estimar

O tempo do algoritmo é da **ordem de  $n$**

## Notação Assintótica

Como vimos, existe uma constante  $d$  tal que, para  $n \geq 1$ ,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

$d$  não interessa tanto, depende apenas do computador...

- Estamos preocupados em estimar

O tempo do algoritmo é da **ordem de  $n$**

- A **ordem de crescimento** do tempo é igual a de  $f(n) = n$

## Notação Assintótica

Como vimos, existe uma constante  $d$  tal que, para  $n \geq 1$ ,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

$d$  não interessa tanto, depende apenas do computador...

- Estamos preocupados em estimar

O tempo do algoritmo é da **ordem de  $n$**

- A **ordem de crescimento** do tempo é igual a de  $f(n) = n$

Dizemos que

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 = O(n)$$



## Notação Assintótica

Como vimos, existe uma constante  $d$  tal que, para  $n \geq 1$ ,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

$d$  não interessa tanto, depende apenas do computador...

- Estamos preocupados em estimar

O tempo do algoritmo é da **ordem de  $n$**

- A **ordem de crescimento** do tempo é igual a de  $f(n) = n$

Dizemos que

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 = O(n)$$

Veremos uma definição formal de  $O(\cdot)$  em breve...

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$ 
  - Ou seja,  $t \leq 1 + \lg n$

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$ 
  - Ou seja,  $t \leq 1 + \lg n$
- gastamos um tempo constante  $c$  em cada chamada



# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$ 
  - Ou seja,  $t \leq 1 + \lg n$
- gastamos um tempo constante  $c$  em cada chamada
  - operações aritméticas, comparações e return

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$ 
  - Ou seja,  $t \leq 1 + \lg n$
- gastamos um tempo constante  $c$  em cada chamada
  - operações aritméticas, comparações e return

Para  $n \geq 1$ , o consumo de tempo é no máximo:

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$ 
  - Ou seja,  $t \leq 1 + \lg n$
- gastamos um tempo constante  $c$  em cada chamada
  - operações aritméticas, comparações e return

Para  $n \geq 1$ , o consumo de tempo é no máximo:

- $ct$

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$ 
  - Ou seja,  $t \leq 1 + \lg n$
- gastamos um tempo constante  $c$  em cada chamada
  - operações aritméticas, comparações e return

Para  $n \geq 1$ , o consumo de tempo é no máximo:

- $ct$

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$ 
  - Ou seja,  $t \leq 1 + \lg n$
- gastamos um tempo constante  $c$  em cada chamada
  - operações aritméticas, comparações e return

Para  $n \geq 1$ , o consumo de tempo é no máximo:

- $ct \leq c + c \lg n$

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$ 
  - Ou seja,  $t \leq 1 + \lg n$
- gastamos um tempo constante  $c$  em cada chamada
  - operações aritméticas, comparações e return

Para  $n \geq 1$ , o consumo de tempo é no máximo:

- $ct \leq c + c \lg n \leq 2c \lg n$

# Busca Binária

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r) / 2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

Se realizarmos  $t$  chamadas, quanto vale  $t$ ?

- primeiro chamamos para  $n$
- depois para  $n/2, n/4, n/8, \dots, n/2^{t-1}$
- no pior caso, só paramos quando  $n/2^t < 1 \leq n/2^{t-1}$ 
  - Ou seja,  $t \leq 1 + \lg n$
- gastamos um tempo constante  $c$  em cada chamada
  - operações aritméticas, comparações e return

Para  $n \geq 1$ , o consumo de tempo é no máximo:

- $ct \leq c + c \lg n \leq 2c \lg n = O(\lg n)$

# Objetivos

Temos dois objetivos para analisar algoritmo



# Objetivos

Temos dois objetivos para analisar algoritmo

- Entender o tempo de execução de um algoritmo

# Objetivos

Temos dois objetivos para analisar algoritmo

- Entender o tempo de execução de um algoritmo
  - Exemplo: busca linear é  $O(n)$

# Objetivos

Temos dois objetivos para analisar algoritmo

- Entender o tempo de execução de um algoritmo
  - Exemplo: busca linear é  $O(n)$
  - Vamos dizer que o algoritmo é  $O(f(n))$

# Objetivos

Temos dois objetivos para analisar algoritmo

- Entender o tempo de execução de um algoritmo
  - Exemplo: busca linear é  $O(n)$
  - Vamos dizer que o algoritmo é  $O(f(n))$
- Comparar dois algoritmos

# Objetivos

Temos dois objetivos para analisar algoritmo

- Entender o tempo de execução de um algoritmo
  - Exemplo: busca linear é  $O(n)$
  - Vamos dizer que o algoritmo é  $O(f(n))$
- Comparar dois algoritmos
  - Busca linear é  $O(n)$  e busca binária é  $O(\lg n)$

# Objetivos

Temos dois objetivos para analisar algoritmo

- Entender o tempo de execução de um algoritmo
  - Exemplo: busca linear é  $O(n)$
  - Vamos dizer que o algoritmo é  $O(f(n))$
- Comparar dois algoritmos
  - Busca linear é  $O(n)$  e busca binária é  $O(\lg n)$
  - Veremos que um algoritmo  $O(\lg n)$  é melhor que um  $O(n)$

# Objetivos

Temos dois objetivos para analisar algoritmo

- Entender o tempo de execução de um algoritmo
  - Exemplo: busca linear é  $O(n)$
  - Vamos dizer que o algoritmo é  $O(f(n))$
- Comparar dois algoritmos
  - Busca linear é  $O(n)$  e busca binária é  $O(\lg n)$
  - Veremos que um algoritmo  $O(\lg n)$  é melhor que um  $O(n)$
  - Prova formal que um algoritmo é melhor que o outro

# Comparando funções

Queremos comparar duas funções  $f$  e  $g$



# Comparando funções

Queremos comparar duas funções  $f$  e  $g$

- Queremos entender a velocidade de crescimento de  $f$

# Comparando funções

Queremos comparar duas funções  $f$  e  $g$

- Queremos entender a velocidade de crescimento de  $f$
- Queremos dizer que  $f$  cresce mais lentamente ou igual a  $g$

# Comparando funções

Queremos comparar duas funções  $f$  e  $g$

- Queremos entender a velocidade de crescimento de  $f$
- Queremos dizer que  $f$  cresce mais lentamente ou igual a  $g$

$f$  pode ser o tempo de execução do algoritmo e  $g$  uma função mais simples de entender

# Comparando funções

Queremos comparar duas funções  $f$  e  $g$

- Queremos entender a velocidade de crescimento de  $f$
- Queremos dizer que  $f$  cresce mais lentamente ou igual a  $g$

$f$  pode ser o tempo de execução do algoritmo e  $g$  uma função mais simples de entender

- $f(n) = c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$  e  $g(n) = n$

# Comparando funções

Queremos comparar duas funções  $f$  e  $g$

- Queremos entender a velocidade de crescimento de  $f$
- Queremos dizer que  $f$  cresce mais lentamente ou igual a  $g$

$f$  pode ser o tempo de execução do algoritmo e  $g$  uma função mais simples de entender

- $f(n) = c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$  e  $g(n) = n$
- $f(n) = 3n^2 + 10 \lg n$  e  $g(n) = n^2$

# Comparando funções

Queremos comparar duas funções  $f$  e  $g$

- Queremos entender a velocidade de crescimento de  $f$
- Queremos dizer que  $f$  cresce mais lentamente ou igual a  $g$

$f$  pode ser o tempo de execução do algoritmo e  $g$  uma função mais simples de entender

- $f(n) = c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$  e  $g(n) = n$
- $f(n) = 3n^2 + 10 \lg n$  e  $g(n) = n^2$

$f$  e  $g$  podem ser os tempos de execução de dois algoritmos

# Comparando funções

Queremos comparar duas funções  $f$  e  $g$

- Queremos entender a velocidade de crescimento de  $f$
- Queremos dizer que  $f$  cresce mais lentamente ou igual a  $g$

$f$  pode ser o tempo de execução do algoritmo e  $g$  uma função mais simples de entender

- $f(n) = c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$  e  $g(n) = n$
- $f(n) = 3n^2 + 10 \lg n$  e  $g(n) = n^2$

$f$  e  $g$  podem ser os tempos de execução de dois algoritmos

- $f(n) = dn$  e  $g(n) = c + c \lg n$

## Primeira Ideia

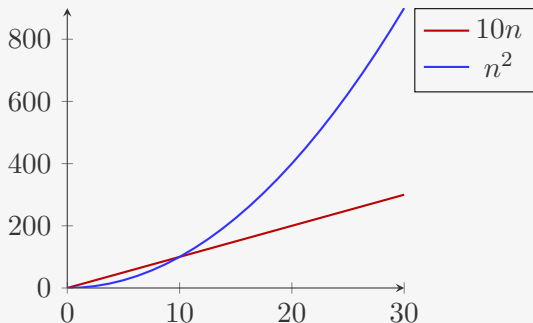
Comparar funções verificando se  $f(n) \leq g(n)$  para todo  $n$



## Primeira Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para todo  $n$

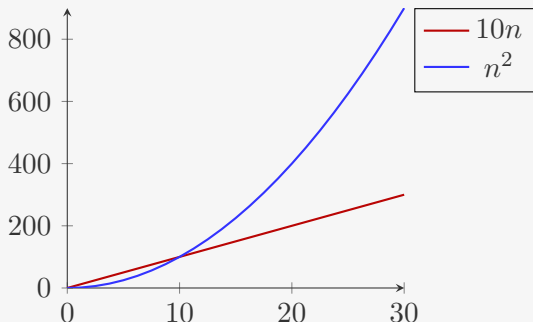
Problema:  $10n > n^2$  para  $n < 10$



## Primeira Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para todo  $n$

Problema:  $10n > n^2$  para  $n < 10$

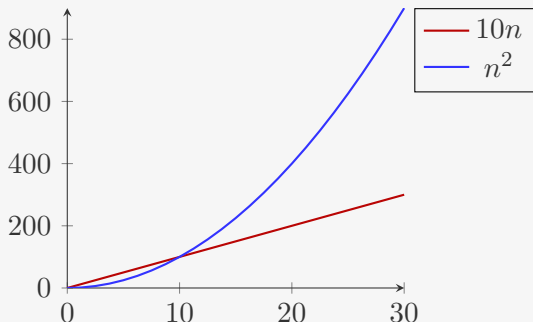


Solução: Ao invés de comparar todo  $n$ , comparar apenas  $n$  suficientemente grande

## Primeira Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para todo  $n$

Problema:  $10n > n^2$  para  $n < 10$



Solução: Ao invés de comparar todo  $n$ , comparar apenas  $n$  suficientemente grande

- Para todo  $n \geq n_0$  para algum  $n_0$

## Segunda Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para  $n \geq n_0$

## Segunda Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para  $n \geq n_0$

Problema:  $n + 5 > n$  para todo  $n$

## Segunda Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para  $n \geq n_0$

Problema:  $n + 5 > n$  para todo  $n$

- Mas a velocidade de crescimento das funções é o mesmo

## Segunda Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para  $n \geq n_0$

Problema:  $n + 5 > n$  para todo  $n$

- Mas a velocidade de crescimento das funções é o mesmo
- Constantes dependem da máquina onde executamos

## Segunda Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para  $n \geq n_0$

Problema:  $n + 5 > n$  para todo  $n$

- Mas a velocidade de crescimento das funções é o mesmo
- Constantes dependem da máquina onde executamos
- Vamos ignorar constantes e termos menos importantes

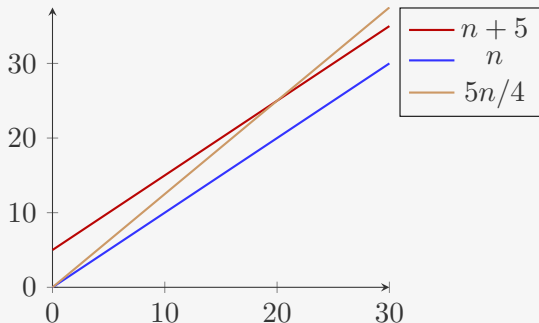


## Segunda Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para  $n \geq n_0$

Problema:  $n + 5 > n$  para todo  $n$

- Mas a velocidade de crescimento das funções é o mesmo
- Constantes dependem da máquina onde executamos
- Vamos ignorar constantes e termos menos importantes

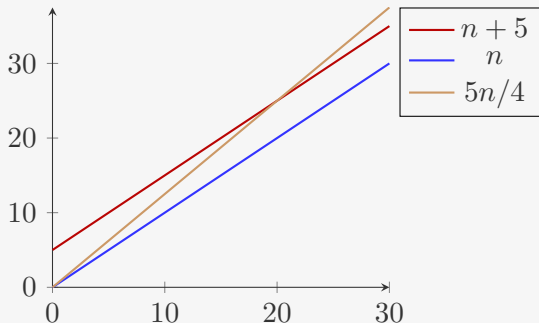


## Segunda Ideia

Comparar funções verificando se  $f(n) \leq g(n)$  para  $n \geq n_0$

Problema:  $n + 5 > n$  para todo  $n$

- Mas a velocidade de crescimento das funções é o mesmo
- Constantes dependem da máquina onde executamos
- Vamos ignorar constantes e termos menos importantes



Solução: Ao invés de comparar  $f$  com  $g$ , comparar com  $c \cdot g$ , onde  $c$  é uma constante

# Notação Assintótica

Dizemos que uma função  $f(n) = O(g(n))$  se

# Notação Assintótica

Dizemos que uma função  $f(n) = O(g(n))$  se

- existe uma constante  $c$

# Notação Assintótica

Dizemos que uma função  $f(n) = O(g(n))$  se

- existe uma constante  $c$
- existe uma constante  $n_0$

# Notação Assintótica

Dizemos que uma função  $f(n) = O(g(n))$  se

- existe uma constante  $c$
- existe uma constante  $n_0$

tal que

# Notação Assintótica

Dizemos que uma função  $f(n) = O(g(n))$  se

- existe uma constante  $c$
- existe uma constante  $n_0$

tal que

$$f(n) \leq c \cdot g(n), \quad \text{para todo } n \geq n_0$$

# Notação Assintótica

Dizemos que uma função  $f(n) = O(g(n))$  se

- existe uma constante  $c$
- existe uma constante  $n_0$

tal que

$$f(n) \leq c \cdot g(n), \quad \text{para todo } n \geq n_0$$

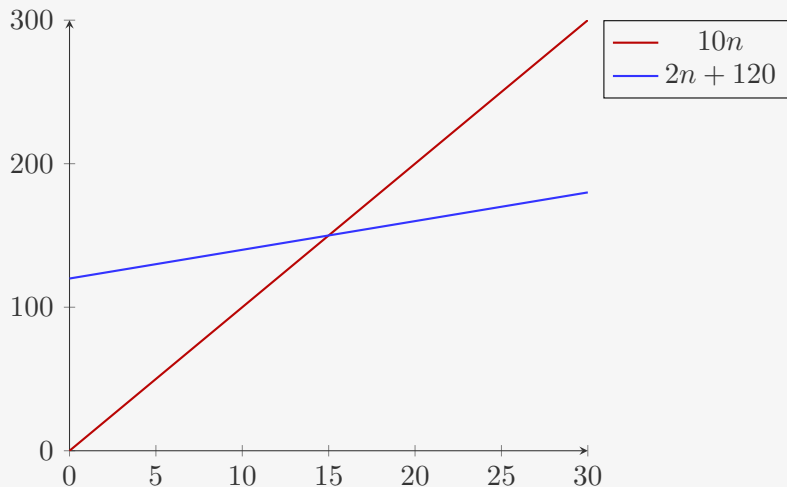
$f(n) = O(g(n))$  se, para todo  $n$  suficientemente grande,  $f(n)$  é menor ou igual a um múltiplo de  $g(n)$



Exemplo:  $2n + 120 = O(n)$

Exemplo:  $2n + 120 = O(n)$

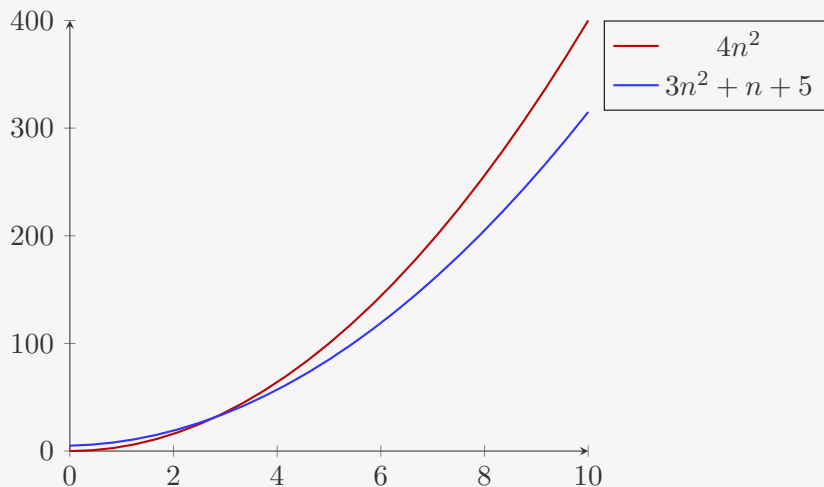
Basta escolher, por exemplo,  $c = 10$  e  $n_0 = 15$



Exemplo:  $3n^2 + n + 5 = O(n^2)$

Exemplo:  $3n^2 + n + 5 = O(n^2)$

Basta escolher, por exemplo,  $c = 4$  e  $n_0 = 4$



## Outros exemplos

$$1 = O(1)$$

## Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

## Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

$$5n + 2 = O(n)$$

## Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

$$5n + 2 = O(n)$$

$$5n^2 + 5n + 2 = O(n^2)$$



## Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

$$5n + 2 = O(n)$$

$$5n^2 + 5n + 2 = O(n^2)$$

$$\log_2 n = O(\log_{10} n)$$

## Outros exemplos

$$1 = O(1)$$

$$1.000.000 = O(1)$$

$$5n + 2 = O(n)$$

$$5n^2 + 5n + 2 = O(n^2)$$

$$\log_2 n = O(\log_{10} n)$$

$$\log_{10} n = O(\log_2 n)$$

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: +, -, \*, /

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: +, -, \*, /
  - Ex: comparações (<, <=, ==, >=, >, !=)

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: +, -, \*, /
  - Ex: comparações (<, <=, ==, >=, >, !=)
  - Ex: operadores booleanos (&&, &, ||, |, !)



# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: +, -, \*, /
  - Ex: comparações (<, <=, ==, >=, >, !=)
  - Ex: operadores booleanos (&&, &, ||, |, !)
  - Ex: acesso a uma posição de um vetor

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: +, -, \*, /
  - Ex: comparações (<, <=, ==, >=, >, !=)
  - Ex: operadores booleanos (&&, &, ||, |, !)
  - Ex: acesso a uma posição de um vetor
  
- $O(\lg n)$ : logarítmico

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: +, -, \*, /
  - Ex: comparações (<, <=, ==, >=, >, !=)
  - Ex: operadores booleanos (&&, &, ||, |, !)
  - Ex: acesso a uma posição de um vetor
- $O(\lg n)$ : logarítmico
  - $\lg$  indica  $\log_2$

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: +, -, \*, /
  - Ex: comparações (<, <=, ==, >=, >, !=)
  - Ex: operadores booleanos (&&, &, ||, |, !)
  - Ex: acesso a uma posição de um vetor
- $O(\lg n)$ : logarítmico
  - $\lg$  indica  $\log_2$
  - quando  $n$  dobra, o tempo aumenta em uma constante

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: +, -, \*, /
  - Ex: comparações (<, <=, ==, >=, >, !=)
  - Ex: operadores booleanos (&&, &, ||, |, !)
  - Ex: acesso a uma posição de um vetor
- $O(\lg n)$ : logarítmico
  - $\lg$  indica  $\log_2$
  - quando  $n$  dobra, o tempo aumenta em uma constante
  - Ex: Busca binária

# Nomenclatura e consumo de tempo

- $O(1)$ : tempo constante
  - não depende de  $n$
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: +, -, \*, /
  - Ex: comparações (<, <=, ==, >=, >, !=)
  - Ex: operadores booleanos (&&, &, ||, |, !)
  - Ex: acesso a uma posição de um vetor
- $O(\lg n)$ : logarítmico
  - $\lg$  indica  $\log_2$
  - quando  $n$  dobra, o tempo aumenta em uma constante
  - Ex: Busca binária
  - Outros exemplos durante o curso

# Nomenclatura e consumo de tempo

- $O(n)$ : linear

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra



# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores
  
- $O(n \lg n)$ :

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores
- $O(n \lg n)$ :
  - quando  $n$  dobra, o tempo um pouco mais que dobra

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores
- $O(n \lg n)$ :
  - quando  $n$  dobra, o tempo um pouco mais que dobra
  - Ex: algoritmos de ordenação que veremos

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores
- $O(n \lg n)$ :
  - quando  $n$  dobra, o tempo um pouco mais que dobra
  - Ex: algoritmos de ordenação que veremos
- $O(n^2)$ : quadrático

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores
- $O(n \lg n)$ :
  - quando  $n$  dobra, o tempo um pouco mais que dobra
  - Ex: algoritmos de ordenação que veremos
- $O(n^2)$ : quadrático
  - quando  $n$  dobra, o tempo quadriplica



# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores
- $O(n \lg n)$ :
  - quando  $n$  dobra, o tempo um pouco mais que dobra
  - Ex: algoritmos de ordenação que veremos
- $O(n^2)$ : quadrático
  - quando  $n$  dobra, o tempo quadriplica
  - Ex: BubbleSort, SelectionSort e InsertionSort

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores
- $O(n \lg n)$ :
  - quando  $n$  dobra, o tempo um pouco mais que dobra
  - Ex: algoritmos de ordenação que veremos
- $O(n^2)$ : quadrático
  - quando  $n$  dobra, o tempo quadriplica
  - Ex: BubbleSort, SelectionSort e InsertionSort
- $O(n^3)$ : cúbico

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores
- $O(n \lg n)$ :
  - quando  $n$  dobra, o tempo um pouco mais que dobra
  - Ex: algoritmos de ordenação que veremos
- $O(n^2)$ : quadrático
  - quando  $n$  dobra, o tempo quadriplica
  - Ex: BubbleSort, SelectionSort e InsertionSort
- $O(n^3)$ : cúbico
  - quando  $n$  dobra, o tempo octuplica

# Nomenclatura e consumo de tempo

- $O(n)$ : linear
  - quando  $n$  dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
  - Ex: Produto interno de dois vetores
- $O(n \lg n)$ :
  - quando  $n$  dobra, o tempo um pouco mais que dobra
  - Ex: algoritmos de ordenação que veremos
- $O(n^2)$ : quadrático
  - quando  $n$  dobra, o tempo quadriplica
  - Ex: BubbleSort, SelectionSort e InsertionSort
- $O(n^3)$ : cúbico
  - quando  $n$  dobra, o tempo octuplica
  - Ex: multiplicação de matrizes  $n \times n$

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )
- O tempo é **menor ou igual** a um múltiplo de  $n^3$

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )
- O tempo é **menor ou igual** a um múltiplo de  $n^3$

Pode ser que o tempo do algoritmo seja  $2n^2...$



## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )
- O tempo é **menor ou igual** a um múltiplo de  $n^3$

Pode ser que o tempo do algoritmo seja  $2n^2$ ...

- $2n^2 = O(n^3)$ , mas...

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )
- O tempo é **menor ou igual** a um múltiplo de  $n^3$

Pode ser que o tempo do algoritmo seja  $2n^2$ ...

- $2n^2 = O(n^3)$ , mas...
- $2n^2 = O(n^2)$

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )
- O tempo é **menor ou igual** a um múltiplo de  $n^3$

Pode ser que o tempo do algoritmo seja  $2n^2$ ...

- $2n^2 = O(n^3)$ , mas...
- $2n^2 = O(n^2)$

Ou seja, podemos ter feito uma análise “folgada”

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )
- O tempo é **menor ou igual** a um múltiplo de  $n^3$

Pode ser que o tempo do algoritmo seja  $2n^2$ ...

- $2n^2 = O(n^3)$ , mas...
- $2n^2 = O(n^2)$

Ou seja, podemos ter feito uma análise “folgada”

- achamos que o algoritmo é muito pior do que é realmente

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )
- O tempo é **menor ou igual** a um múltiplo de  $n^3$

Pode ser que o tempo do algoritmo seja  $2n^2$ ...

- $2n^2 = O(n^3)$ , mas...
- $2n^2 = O(n^2)$

Ou seja, podemos ter feito uma análise “folgada”

- achamos que o algoritmo é muito pior do que é realmente

No curso, não faremos análises “folgadas”

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )
- O tempo é **menor ou igual** a um múltiplo de  $n^3$

Pode ser que o tempo do algoritmo seja  $2n^2$ ...

- $2n^2 = O(n^3)$ , mas...
- $2n^2 = O(n^2)$

Ou seja, podemos ter feito uma análise “folgada”

- achamos que o algoritmo é muito pior do que é realmente

No curso, não faremos análises “folgadas”

- existe uma maneira formal de lidar com isso (notação  $\Theta$ )

## Um cuidado

O que significa dizer que o tempo de um algoritmo é  $O(n^3)$ ?

- Para instâncias grandes ( $n \geq n_0$ )
- O tempo é **menor ou igual** a um múltiplo de  $n^3$

Pode ser que o tempo do algoritmo seja  $2n^2$ ...

- $2n^2 = O(n^3)$ , mas...
- $2n^2 = O(n^2)$

Ou seja, podemos ter feito uma análise “folgada”

- achamos que o algoritmo é muito pior do que é realmente

No curso, não faremos análises “folgadas”

- existe uma maneira formal de lidar com isso (notação  $\Theta$ )
- mas não precisamos desse formalismo em MC202

## Exercício

1. Mostre que  $n + \lg n = O(n)$



## Exercício

1. Mostre que  $n + \lg n = O(n)$
2. Mostre que  $15n = O(n \lg n)$  mas que  $n \lg n \neq O(n)$

## Exercício

1. Mostre que  $n + \lg n = O(n)$
2. Mostre que  $15n = O(n \lg n)$  mas que  $n \lg n \neq O(n)$ 
  - Essa análise é folgada, já que  $15n = O(n)$

## Exercício

1. Mostre que  $n + \lg n = O(n)$
2. Mostre que  $15n = O(n \lg n)$  mas que  $n \lg n \neq O(n)$ 
  - Essa análise é folgada, já que  $15n = O(n)$
3. Mostre que  $42n = O(n^2)$  mas que  $n^2 \neq O(42n)$

## Exercício

1. Mostre que  $n + \lg n = O(n)$
2. Mostre que  $15n = O(n \lg n)$  mas que  $n \lg n \neq O(n)$ 
  - Essa análise é folgada, já que  $15n = O(n)$
3. Mostre que  $42n = O(n^2)$  mas que  $n^2 \neq O(42n)$ 
  - Essa análise é folgada, já que  $42n = O(n)$