AED - Algoritmos e Estruturas de Dados

Aula 7 – Conceitos de Complexidade Computacional

Prof. Rodrigo Mafort

Prof. Rodrigo Mafort

Como avaliar um algoritmo?

- Ideia: Vamos usar o tempo de execução

vs

Pergunta: Essa é uma comparação justa?

Não!

Mas como podemos avaliar ou comparar um algoritmo sem depender da máquina?

Como avaliar um algoritmo?

Algoritmo 1:

Entrada: Vetor V com n elementos

```
1 para i ← 1 ... n faça
2     min ← i
3     para j ← i + 1 ... n faça
4         se V[j] < V[min] então
5             min ← j
6     aux ← V[i]
7     V[i] ← V[min]
8     V[min] ← aux
9 retorne V
```

Como avaliar um algoritmo?

Algoritmo 2:

Entrada: Vetor V com n elementos

Função `Ajustar`(V, ini, fim)

```
1  se ini < fim então
2      meio ← (ini + fim)/2
3      Ajustar(V, ini, meio)
4      Ajustar(V, meio + 1, fim)
5      Unir(V, ini, meio, fim)
6  retorne V
```

Função `Unir`$(V, ini, meio, fim)$

```
7  x ← fim − ini + 1
8  Criar vetor A com x posições
9  p1 ← ini        p2 ← fim
10 para i ← 1 ... x faça
11     se p1 ≤ meio ∧ p2 ≤ fim então
12         se V[p1] < V[p2] então A[i] ← V[p1]        p1 ← p1 + 1;
13         senão A[i] ← V[p2]        p2 ← p2 + 1;
14     senão
15         se p1] ≤ meio então A[i] ← V[p1]        p1 ← p1 + 1;
16         senão A[i] ← V[p2]        p2 ← p2 + 1;
17 para i ← 1 ... x faça  V[ini + i] ← A[i] ;
18 retorne V
```

Como avaliar um algoritmo

- O que os algoritmos fazem?
  - Ordenação de vetores
- Qual é mais eficiente em tempo de execução?
  - O segundo
- Por que?
  - Estudo da complexidade computacional.
- Curiosidade: O que acontece quando o computador tem a memória limitada praticamente ao tamanho do vetor?

Como avaliar sem depender do computador?

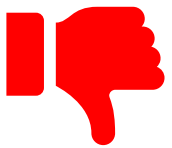
- Podemos estudar o algoritmo e ver quantas instruções ele executa

Algoritmo 1: Identificar o maior elemento

Entrada: Vetor V com n elementos
Saída: Maior elemento *maior*

```
1 maior ← v[0]
2 para i ← 1 ... n − 1 faça
3     se v[i] > maior então
4         maior ← v[i]
5 retorne maior
```

O algoritmo executa 5 instruções!



Não!

Como avaliar sem depender do computador?

- Podemos estudar o algoritmo e ver quantas instruções ele executa

Algoritmo 1: Identificar o maior elemento

Entrada: Vetor V com n elementos
Saída: Maior elemento *maior*

```
1 maior ← v[0]
2 para i ← 1 ... n − 1 faça
3     se v[i] > maior então
4         maior ← v[i]
5 retorne maior
```

Esses comandos são executados para todos os elementos da lista, exceto o primeiro

O algoritmo executa $n - 1$ instruções no total?

Como avaliar sem depender do computador?

- Mas vamos contar TODOS os comandos?

- Não! Somente a **operação dominante** do algoritmo

- Dentre todos os comandos do algoritmo, a operação dominante é aquela operação básica executada com a maior frequência no algoritmo.

- Em um algoritmo de ordenação, qual é a operação dominante?
  - Comparações! Mesmo as trocas, dependem das comparações...

Como avaliar sem depender do computador?

- Qual é a operação dominante desse algoritmo?

Algoritmo 1: Identificar o maior elemento

Entrada: Vetor V com n elementos
Saída: Maior elemento *maior*

```
1 maior ← v[0]
2 para i ← 1 ... n − 1 faça
3     se v[i] > maior então
4         maior ← v[i]
5 retorne maior
```

A comparação!

Precisamos compara todos os elementos do vetor para identificar o maior

Em um vetor com n elementos, realizamos $n - 1$ comparações

Como avaliar sem depender do computador?

- Quantas comparações esse algoritmo faz?

Algoritmo 2: Identificar o maior e o menor elemento

Entrada: Vetor V com n elementos
Saída: Maior elemento *maior*

```
1 maior ← v[0]
2 menor ← v[0]
3 para i ← 1 ... n − 1 faça
4 |   se v[i] > maior então maior ← v[i]
5 |   se v[i] < menor então menor ← v[i]
6 retorne maior, menor
```

$2\,(n-1)$ comparações

Podemos melhorar esse número???
Como?

Como avaliar sem depender do computador?

- E agora?

Algoritmo 3: Identificar o maior e o menor elemento

Entrada: Vetor V com n elementos
Saída: Maior elemento *maior*

```
1 maior ← v[0]
2 menor ← v[0]
3 para i ← 1 ... n − 1 faça
4     se v[i] > maior então
5         maior ← v[i]
6     senão
7         se v[i] < menor então
8             menor ← v[i]
9 retorne maior, menor
```

Complicou... ☹

Agora depende da entrada...
Mas e no pior caso?

Como avaliar sem depender do computador?

- E agora?

Algoritmo 3: Identificar o maior e o menor elemento

Entrada: Vetor V com n elementos
Saída: Maior elemento *maior*

```
1 maior ← v[0]
2 menor ← v[0]
3 para i ← 1 ... n − 1 faça
4     se v[i] > maior então
5         maior ← v[i]
6     senão
7         se v[i] < menor então
8             menor ← v[i]
9 retorne maior, menor
```

Complicou... ☹

Agora depende da entrada...
Mas e no pior caso?

Como avaliar sem depender do computador?

- E agora?

Algoritmo 3: Identificar o maior e o menor elemento

Entrada: Vetor V com n elementos
Saída: Maior elemento *maior*

```
1 maior ← v[0]
2 menor ← v[0]
3 para i ← 1 ... n − 1 faça
4     se v[i] > maior então
5         maior ← v[i]
6     senão
7         se v[i] < menor então
8             menor ← v[i]
9 retorne maior, menor
```

Pior caso:

Vamos ter que fazer as duas perguntas todas as vezes...

Logo, executamos $2\ (n-1)$ comparações

Exemplo de pior caso:

V = [1,2,3,4,5,6,7,8,9,10]

Como avaliar sem depender do computador?

- E agora?

Algoritmo 3: Identificar o maior e o menor elemento

Entrada: Vetor V com n elementos
Saída: Maior elemento *maior*

```
1 maior ← v[0]
2 menor ← v[0]
3 para i ← 1 ... n − 1 faça
4     se v[i] > maior então
5         maior ← v[i]
6     senão
7         se v[i] < menor então
8             menor ← v[i]
9 retorne maior, menor
```

Mas e no melhor caso?

No melhor caso, vamos executar apenas uma das comparações, isto é $(n-1)$ comparações.

Exemplo de melhor caso:

```
V = [10,9,8,7,6,5,4,3,2,1]
```

Como avaliar sem depender do computador?

- E agora?

Algoritmo 3: Identificar o maior e o menor elemento

Entrada: Vetor V com n elementos
Saída: Maior elemento *maior*

```
1 maior ← v[0]
2 menor ← v[0]
3 para i ← 1 ... n − 1 faça
4     se v[i] > maior então
5         maior ← v[i]
6     senão
7         se v[i] < menor então
8             menor ← v[i]
9 retorne maior, menor
```

Mas e na média?

Neste caso, precisamos estudar a probabilidade de que cada caso ocorra...

Em média, $\frac{3n-2}{2}$ comparações

Para pensar:

Algoritmo 4: "Ajustar" vetor

Entrada: Vetor V com n elementos
Saída: Vetor V

```
1 para i ← 0 ... ⌊n/2⌋ faça
2     aux ← V[i]
3     V[i] ← V[(n − 1) − i]
4     V[(n − 1) − i] ← aux
5 retorne V
```

- Qual é a operação dominante? **Trocas.**
- Qual operações dominantes são executadas? $\left\lfloor \frac{n}{2} \right\rfloor$

Para pensar:

Algoritmo 5: O que o algoritmo faz?

Entrada: Vetor V com n elementos
Saída: Vetor V

```
1  faça
2      troca ← Falso
3      para i ← 0 ... n − 2 faça
4          se V[i] > V[i + 1] então
5              aux ← V[i]
6              V[i] ← V[i + 1]
7              V[i + 1] ← aux
8              troca ← Verdadeiro
9  enquanto troca = Falso
10 retorne V
```

Para pensar:

Algoritmo 5: Qual é a complexidade de pior caso?

Entrada: Vetor V com n elementos
Saída: Vetor V

```
1  faça
2      troca ← Falso
3      para i ← 0 ... n − 2 faça
4          se V[i] > V[i + 1] então
5              aux ← V[i]
6              V[i] ← V[i + 1]
7              V[i + 1] ← aux
8              troca ← Verdadeiro
9  enquanto troca = Falso
10 retorne V
```

Para pensar:

Algoritmo 5: Qual é a complexidade de melhor caso?

Entrada: Vetor V com n elementos
Saída: Vetor V

```
1  faça
2      troca ← Falso
3      para i ← 0 ... n − 2 faça
4          se V[i] > V[i + 1] então
5              aux ← V[i]
6              V[i] ← V[i + 1]
7              V[i + 1] ← aux
8              troca ← Verdadeiro
9  enquanto troca = Falso
10 retorne V
```

Complexidade

- Essa contagem de instruções é a base para se obter a complexidade de um algoritmo.

- O objetivo aqui é avaliar como o número de instruções que são executadas cresce de acordo com o tamanho da entrada

- Existem três formas de medir:
  - **Complexidade de Pior Caso**
  - **Complexidade de Melhor Caso**
  - **Complexidade de Caso Médio**

- A complexidade de pior caso é a mais usada (ela é o limite máximo).

Complexidade

Seja A um algoritmo qualquer

Seja $E = \{E_1 \ \ldots \ E_m\}$ o conjunto de todos os tipos de entradas possíveis para A.

Seja t_i o número de operações dominantes efetuadas por A, quando a entrada for E_i

- Complexidade de pior caso: $\max_{E_i \in E} \{ t_i \}$
- Complexidade de melhor caso: $\min_{E_i \in E} \{ t_i \}$
- Complexidade de caso médio: $\sum_{E_i \in E} p_i \, t_i$, onde p_i é a probabilidade da entrada E_i ocorrer

- Podemos definir também a complexidade de espaço de um algoritmo, isto é, a quantidade de memória necessária para executar um algoritmo (memória principal, também conhecida como memória RAM)

Exemplo:

```
vetor = [5,6,11,3,10,2,1,4,8,9,7,13,15,25,12]
```

Algoritmo 6: Buscar posição

Entrada: Vetor V com n elementos, Elemento e
Saída: Posição do vetor

```
1 para i ← 0 ... n − 1 faça
2     se V[i] = e então
3         retorne i
4 retorne n
```

- Qual é a complexidade de pior caso?
- Qual é a complexidade de melhor caso?

n comparações

1 comparação

Complexidade de Algoritmos Recursivos

Algoritmo 7: Qual é a complexidade?

Entrada: Vetor V com n posições
Saída: Vetor V ordenado
Chamada Inicial: `CocktailSort` $(V, 0, n-1)$

```
1  Procedimento CocktailSort(V, ini, L)
2      se ini < fim então
3          menor ← ini
4          maior ← ini
5          para i ← ini + 1 ... fim faça
6              se V[i] < V[menor] então menor ← i
7              senão se V[i] > V[maior] então maior ← i
8          V[ini] ↔ V[menor]
9          V[fim] ↔ V[maior]
10         CocktailSort (V, ini + 1, fim − 1)
```

Complexidade de Algoritmos Recursivos

- Para determinar a complexidade de algoritmos recursivos precisamos muitas vezes retornar à fórmula da recorrência.
- A fórmula será usada para calcular o número de operações dominantes em cada chamada recursiva. Vamos chamar de $T(n)$ o trabalho necessário para uma entrada de tamanho n.

- $T(n) = \begin{cases} 2(n-1) + T(n-2), se\ n > 1 \\ 0, \qquad\quad caso\ contrário \end{cases}$

- Onde $2(n-1)$ é o número de comparações efetuadas no pior caso para uma lista de tamanho n.

Complexidade de Algoritmos Recursivos

- $T(n) = \begin{cases} 2(n-1) + T(n-2), & se\ n > 1 \\ 0, & caso\ contrário \end{cases}$

- Desenvolvendo a equação:
- $T(n) = 2(n-1) + T(n-2)$
- $T(n) = 2(n-1) + 2(n-3) + T(n-5)$
- ...
- $T(n) = 2(n-1) + 2(n-3) + 2(n-5) + \cdots + 0$

- Quantos termos tem essa sequência?

Complexidade de Algoritmos Recursivos

- $T(n) = 2(n-1) + 2(n-3) + 2(n-5) + \cdots + 0$

- Quantas subtrações são necessárias até atingir 0?
- $2(n - 2k - 1) = 0$
- $2n - 4k - 2 = 0$
- $-4k = 2 - 2n$
- $-2k = 1 - n$
- $2k = n - 1$
- $k = \frac{n-1}{2}$

Complexidade de Algoritmos Recursivos

- $T(n) = \begin{cases} 2(n-1) + T(n-2), & se\ n > 1 \\ 0, & caso\ contrário \end{cases}$

- $T(n) = 2(n-1) + 2(n-3) + 2(n-5) + \cdots + 0$
- $T(n) = 2(n-1+n-3+n-5+\cdots+0)$

- $n-1+n-3+n-5+\cdots+0$
- Soma de PA com razão -2 com $\frac{n-1}{2}$ termos e termo inicial $n-1$:
- $S = \frac{\frac{n-1}{2}(n-1+0)}{2} = \frac{n^2-2n+1}{4}$

Complexidade de Algoritmos Recursivos

- $T(n) = \begin{cases} 2(n-1) + T(n-2), & se\ n > 1 \\ 0, & caso\ contrário \end{cases}$

- $T(n) = 2(n-1) + 2(n-3) + 2(n-5) + \cdots + 0$
- $T(n) = 2(n-1+n-3+n-5+\cdots+0)$

- $T(n) = 2\left(\frac{n^2-2n+1}{4}\right)$
- $T(n) = \frac{n^2-2n+1}{2}$ comparações

Crescimento Assintótico de Funções

- Quando estudamos a complexidade de um algoritmo, estamos interessados em seu comportamento **assintótico**
- Isto é, o crescimento de sua complexidade para entradas suficientemente grandes.
- **Por que?**
- Simples: muitas vezes é difícil determinar o número exato de vezes que as instruções são executadas...
- Além disso, algumas vezes encontramos expressões grandes, com muitas informações, como: $3n^2 + \frac{12}{8}n + 7$

Crescimento Assintótico de Funções

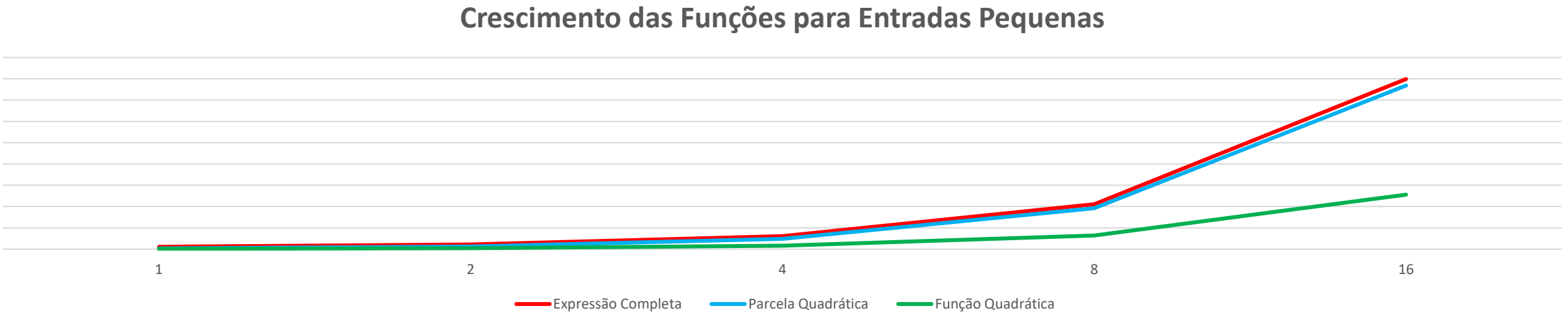
- Quando analisamos o comportamento assintótico de tais funções, podemos verificar que, à medida que o valor de $\boldsymbol{n}$ cresce, a parcela quadrática $3n^2$ torna-se dominante em relação às demais parcelas.

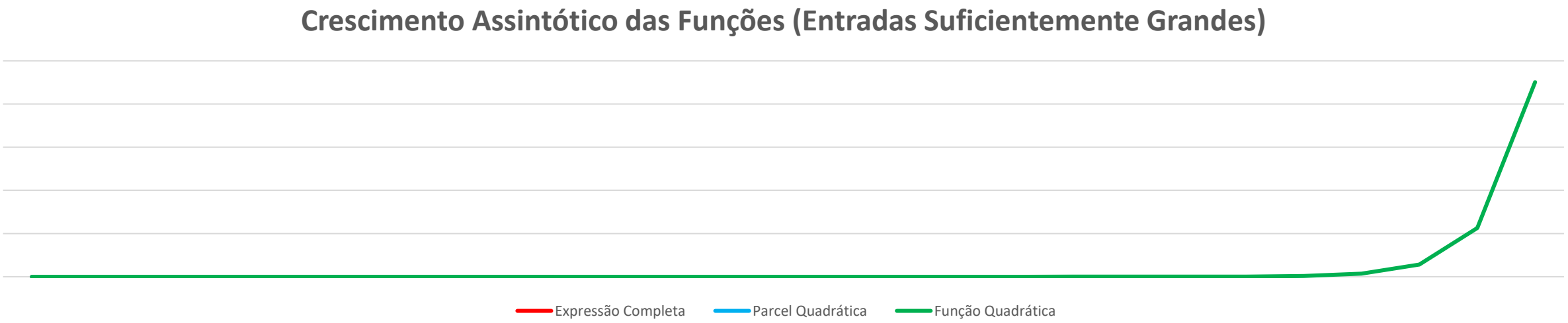
- Isto é: o valor de $3n^2$ se torna tão mais significativo do que $\frac{12}{8}n + 7$, que podemos considerar apenas a primeira parte sem nenhuma perda importante.

- Podemos aplicar a mesma lógica a $3n^2$ e n^2: Como o comportamento assintótico das duas funções é praticamente o mesmo, podemos desconsiderar a constante 3

- Isto é: Podemos considerar que $3n^2 + \frac{12}{8}n + 7$ é "assintoticamente equivalente" a n^2.

Crescimento Assintótico de Funções



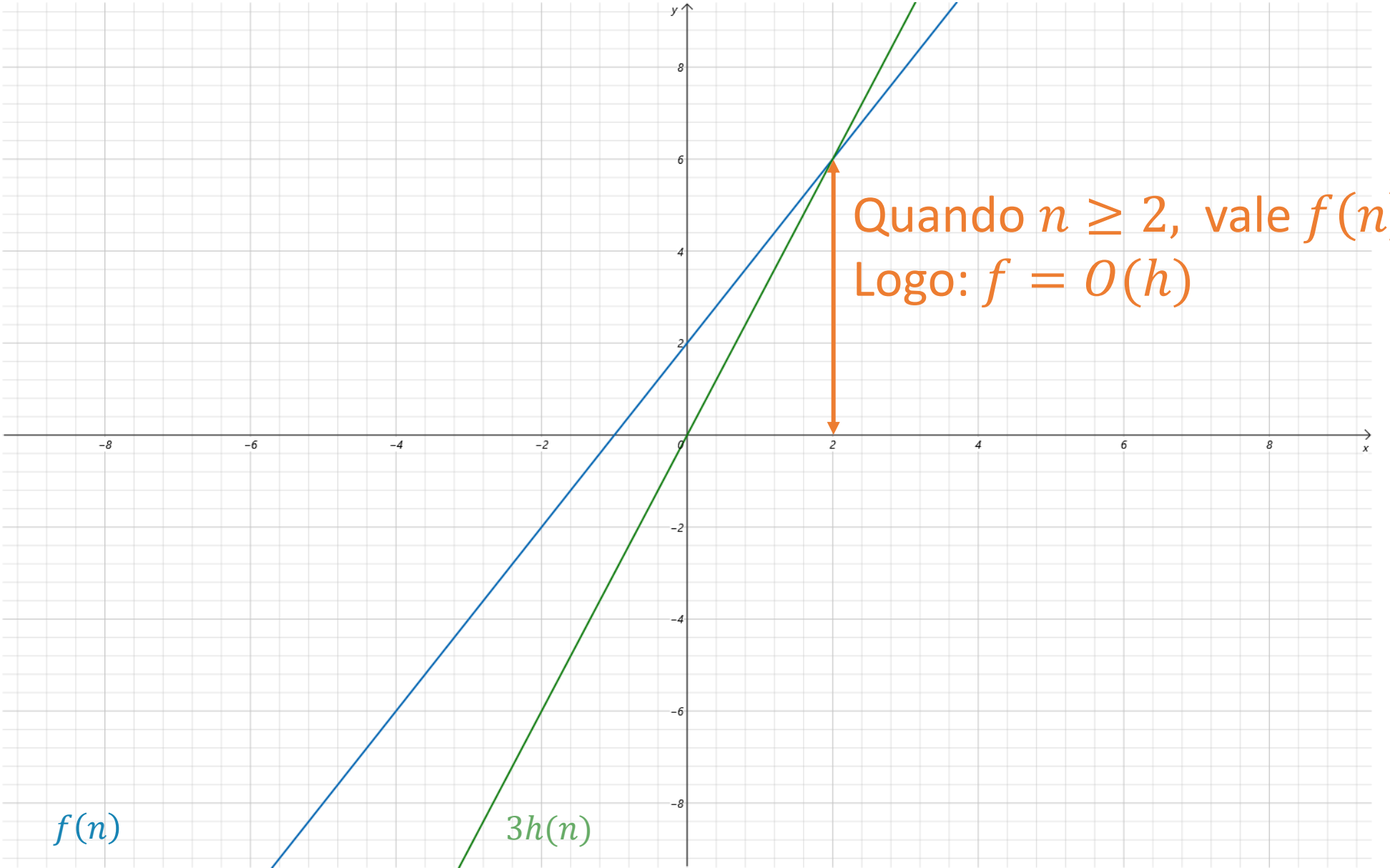


Notação O

- Sejam f e h duas funções reais positivas.
- Dizemos que f é $O(h)$ (escrevemos que $f = O(h)$), quando:
  - Existe uma constante $c > 0$
  - Existe um valor inteiro n_0
  - Tais que: para todo $n > n_0$, vale $f(n) \leq c \,.\, h(n)$

- Exemplo: $f(n) = 2n + 2$, $h(n) = n$, $n_0 = 2$ e $c = 3$
- A partir de $n_0 = 2$, podemos notar que $f(n) \leq c \,.\, h(n)$
- Ou seja: $f(n) \leq 3 \,.\, h(n)$ ou $2n + 2 \leq 3\, n$
- Logo: f é $O(h)$ ou f é $O(n)$

Notação O


y
x
8
6
4
2
-2
-4
-6
-8
-8
-6
-4
-2
0
2
4
6
8
Quando $n \geq 2$, vale $f(n) \leq 3 \,.\, h(n)$
Logo: $f = O(h)$
$f(n)$
$3h(n)$

Exemplos de Notação O

- $f(n) = n^2 - 1 \quad \Rightarrow \quad f = O(n^2)$
- $f(n) = n^2 - 1 \quad \Rightarrow \quad f = O(n^3)$
- $f(n) = 1058987125 \quad \Rightarrow \quad f = O(1)$
- $f(n) = 3n + 5\log n + 5 \quad \Rightarrow \quad f = O(n)$
- $f(n) = 2n * \log n \quad \Rightarrow \quad f = O(n\log n)$
- $f(n) = 2^n + n^2 \quad \Rightarrow \quad f = O(2^n)$
- $f(n) = \frac{n!}{6!(n-6)!} \quad \Rightarrow \quad f = O(n!)$

Exemplos de Notação O

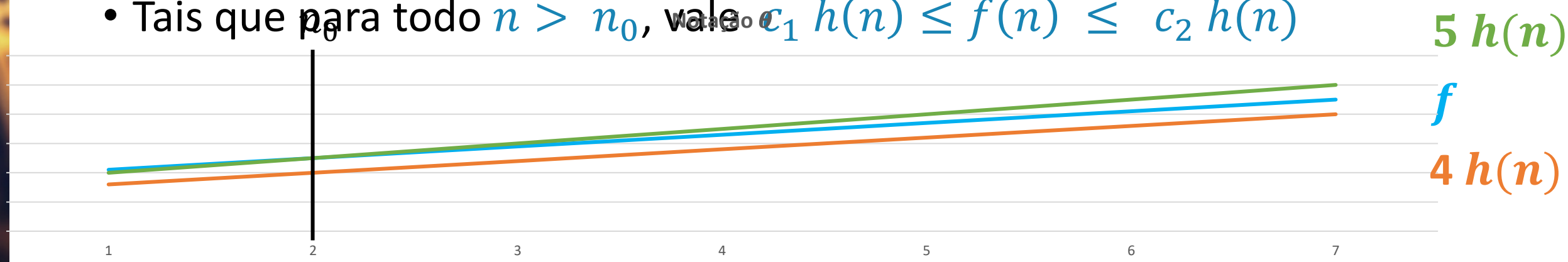
- Algoritmo para encontrar o máximo: $O(n)$
- Algoritmo para encontrar o máximo e o mínimo: $O(n)$
- Algoritmo para buscar um elemento no vetor: $O(n)$
- Algoritmo de ordenação MergeSort: $O(n \log n)$
- Algoritmo de ordenação BubbleSort: $O(n^2)$
- Algoritmo de ordenação BogoSort: $O(n!)$

Notação Ω

- Usada para definir limites assintoticamente inferiores
- Sejam f, h duas funções reais positivas
- Dizemos que f é $\boldsymbol{\Omega}\,(h)$ ($\mathrm{f} = \boldsymbol{\Omega}\,(h)$) quando:
  - Existe uma constante $c > 0$
  - Existe um valor inteiro n_0
  - Tais que para todo $n > n_0$, vale $f(n) \geq c\,.\,h(n)$
- Exemplo:
- $f(n) = n^2 - 1 \quad \Rightarrow \quad f = \boldsymbol{\Omega}\,(n^2)$
- $f(n) = n^2 - 1 \quad \Rightarrow \quad f = \boldsymbol{\Omega}\,(n)$
- $f(n) = n^2 - 1 \quad \Rightarrow \quad f = \boldsymbol{\Omega}\,(1)$
- $f(n) = n^2 - 1 \quad \nRightarrow \quad f = \boldsymbol{\Omega}\,(n^3)$

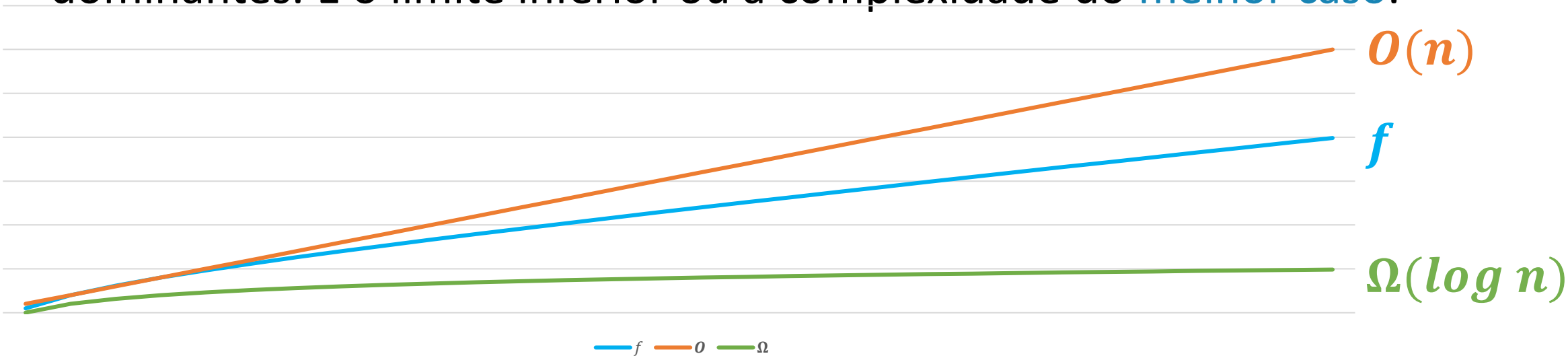
Notação θ

- Usada para definir um limite mais próximo (as duas funções crescem da mesma forma)
- Sejam f, h duas funções reais positivas: $f(n) = 4n + 5$ e $h(n) = n$
- Dizemos que f é $\Theta(h)$ ($\mathrm{f} = \Theta(h)$) quando:
  - Existem duas constante $c_1 > 0$ e $c_2 > 0$ tais que $c_1 \leq c_2$
  - Existe um valor inteiro n_0
  - Tais que para todo $n > n_0$, vale $c_1\, h(n) \leq f(n) \leq c_2\, h(n)$



Notação O e Ω

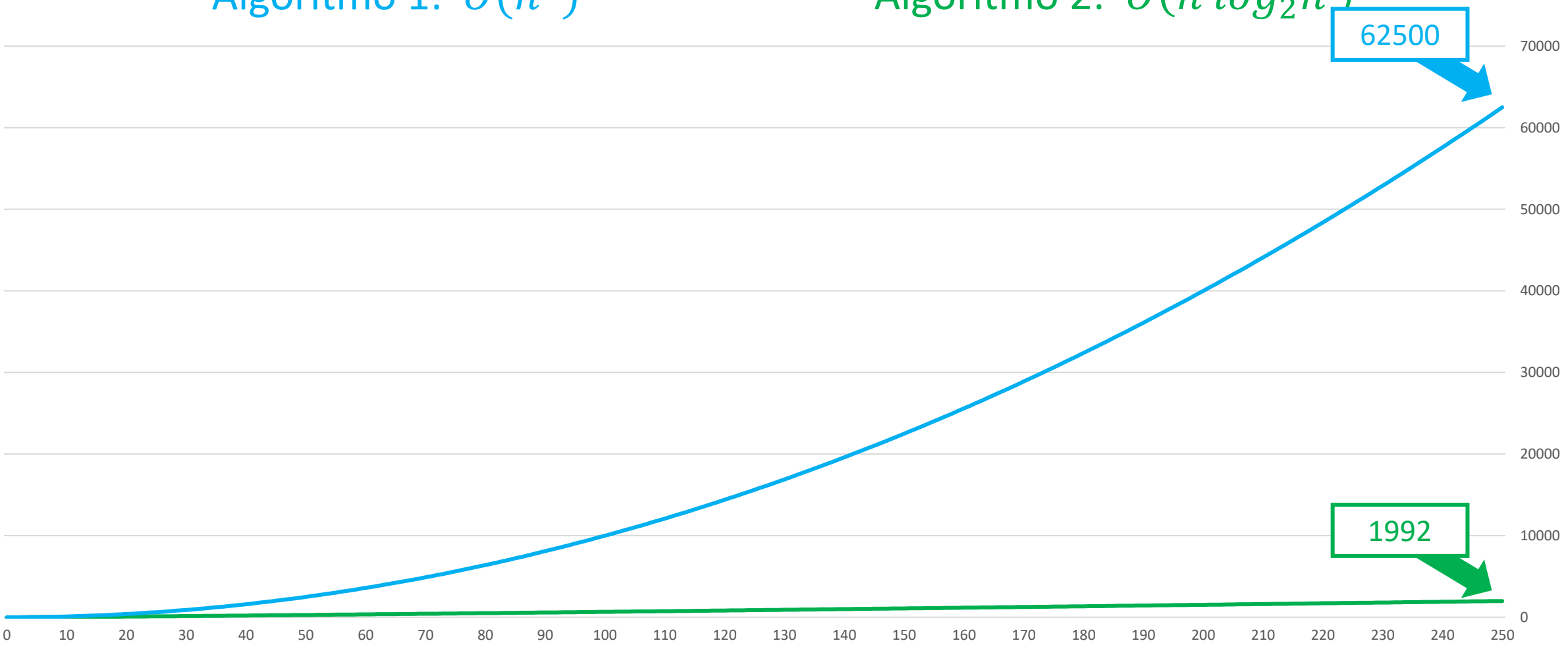
- Usamos para expressar respectivamente o pior e o melhor caso de um algoritmo. Por exemplo:
  - Um algoritmo $\boldsymbol{O(n)}$ nunca vai executar mais do que $c * n$ operações dominantes (onde c é uma constante). É o limite superior ou a complexidade do pior caso.
  - Um algoritmo $\boldsymbol{\Omega(n^3)}$ nunca vai executar menos do que n^3 operações dominantes. É o limite inferior ou a complexidade do melhor caso.



Comparação entre Algoritmos 1 e 2

Algoritmo 1: $O(n^2)$

Algoritmo 2: $O(n\ log_2 n)$



Quando $n = 1.000.000$: $n\ log_2 n = 19.931.568\ (20\ milhões)$ $n^2 = 1.000.000.000.000\ (1\ trilhão)$

Classes de Complexidade de Algoritmos

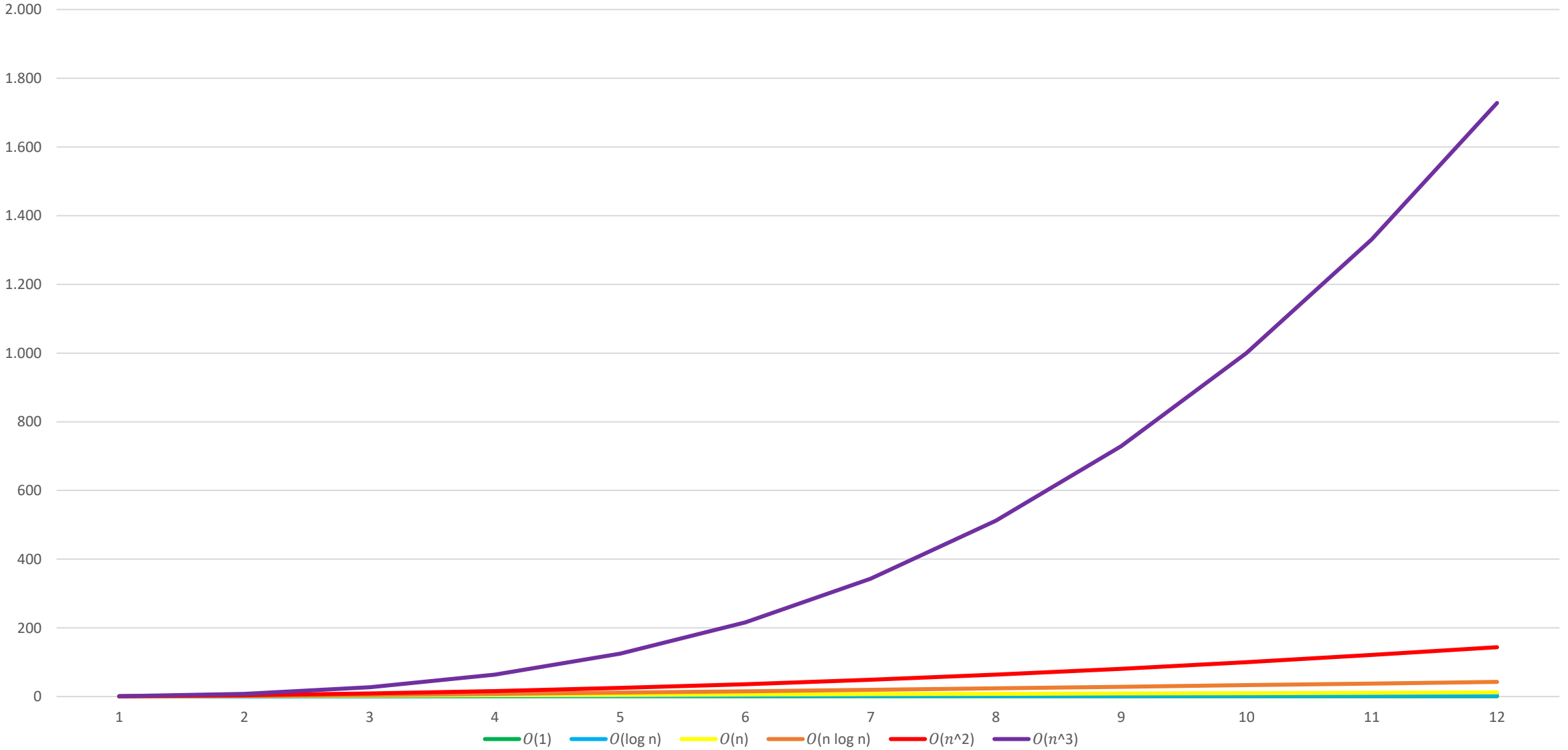
- $O(1)$
- $O(log_2\, n)$
- $O(n)$
- $O(n\, log_2\, n)$
- $O(n^2)$
- $O(n^3)$
- $O(n^c)$ (assumindo que c é uma constante)

Para entradas grandes,
esse é o limite da computação

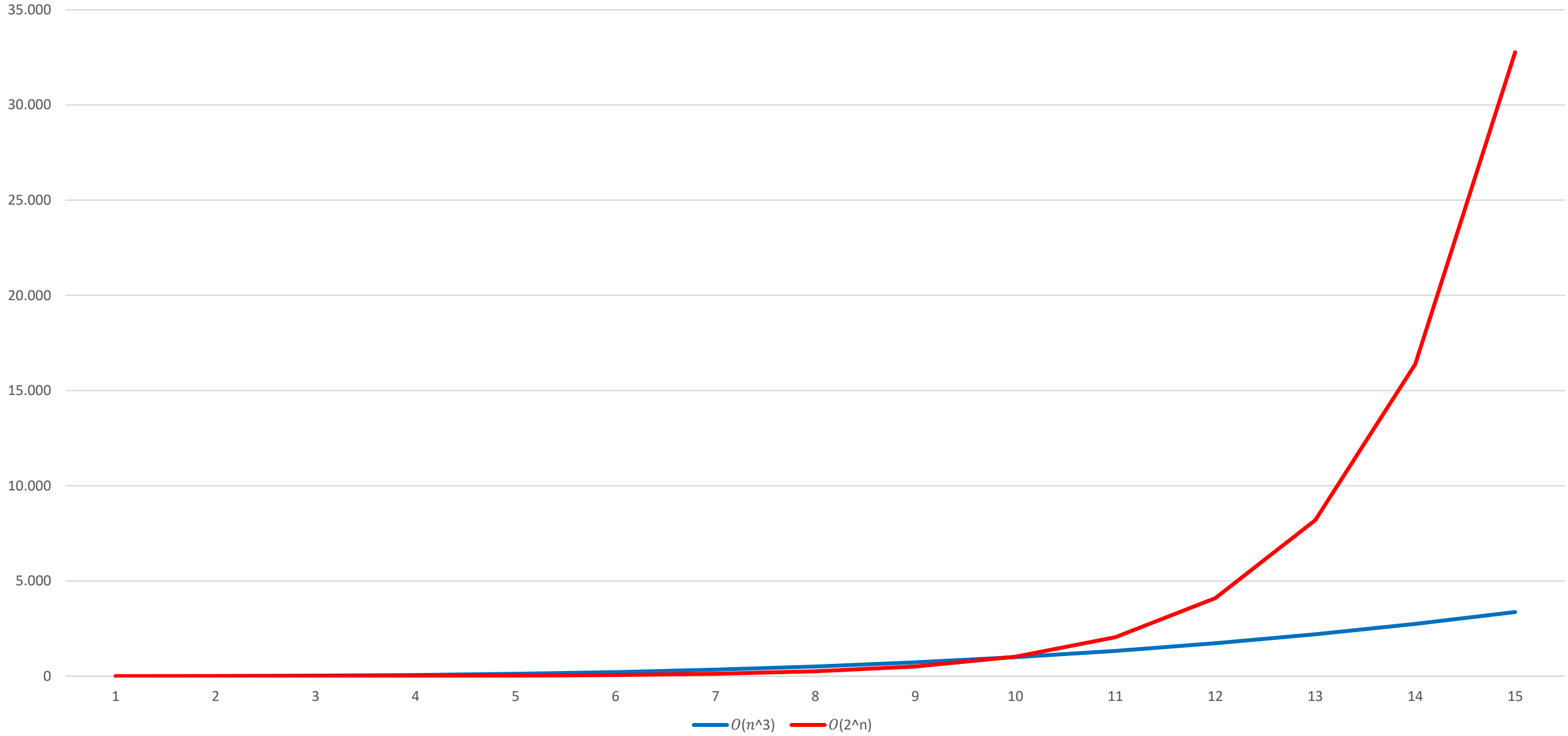
- $O(2^n)$
- $O(n!)$

- $O(n^n)$

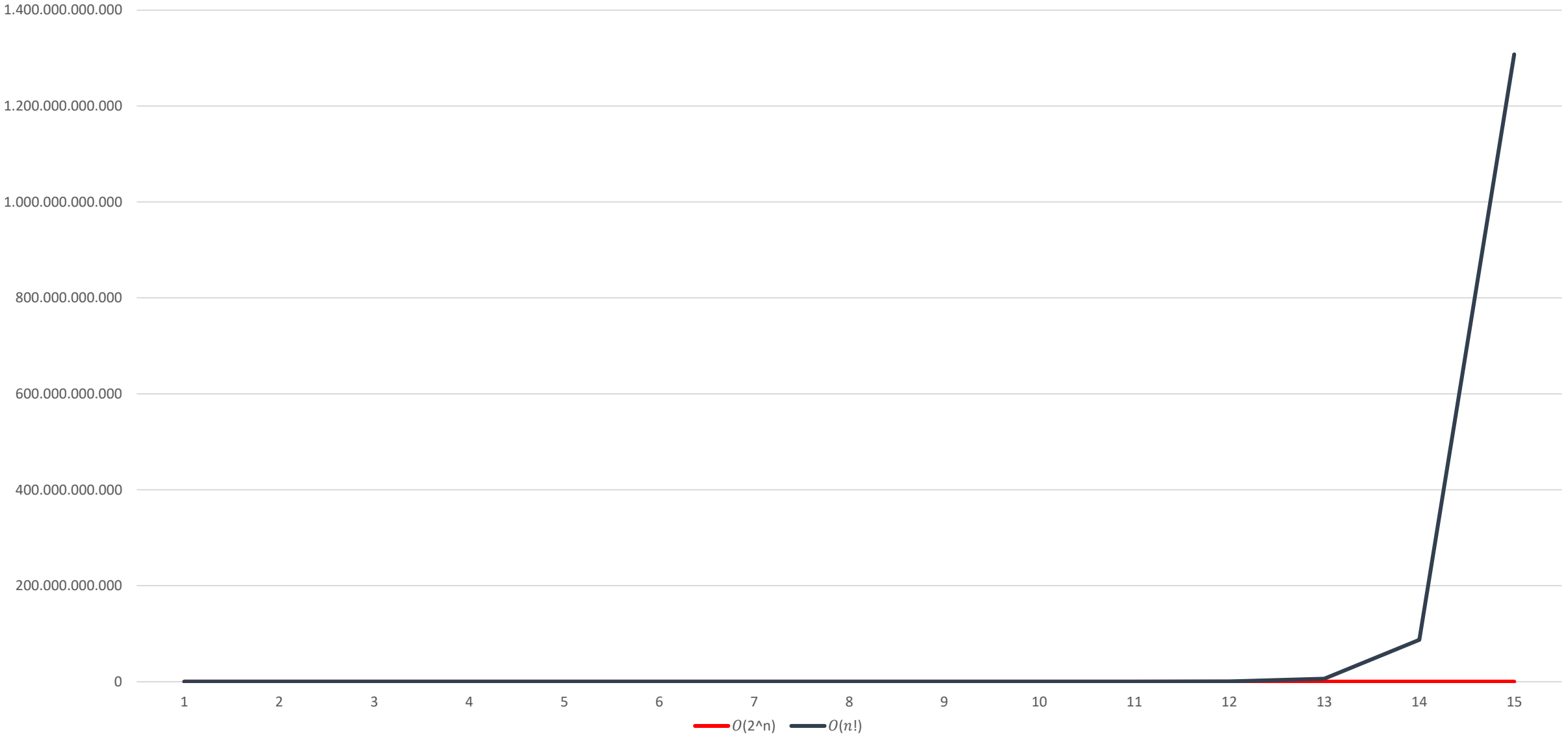
Comparação entre Algoritmos



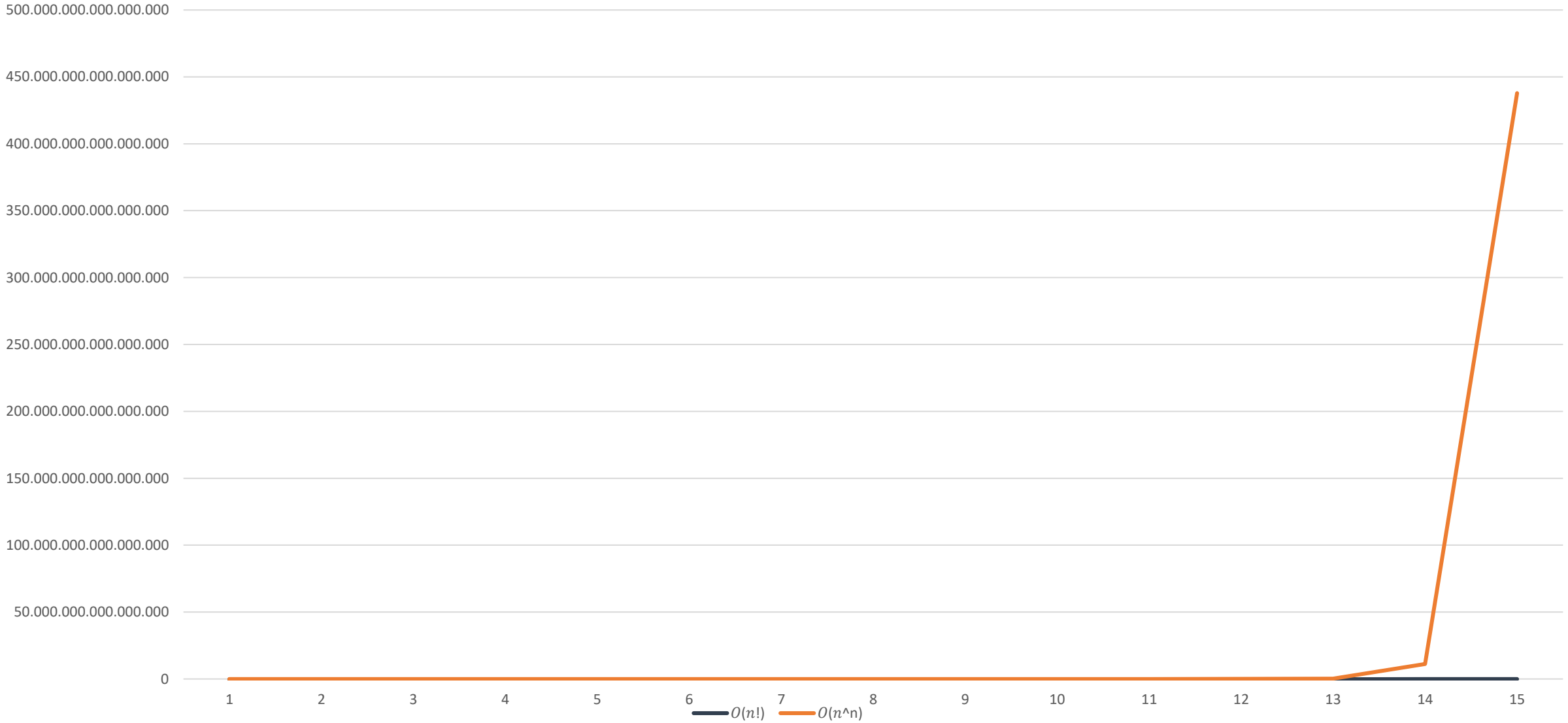
Comparação Entre Algoritmos



Comparação Entre Algoritmos



Comparação Entre Algoritmos



Comparação Entre Algoritmos

n	$O(1)$	$O(log_2 n)$	$O(n)$	$O(n\,log_2 n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$	$O(n!)$	$O(n^n)$
1	1	0,0	1,0	0,0	1	1	2	1	1
2	1	1,0	2,0	2,0	4	8	4	2	4
3	1	1,6	3,0	4,8	9	27	8	6	27
4	1	2,0	4,0	8,0	16	64	16	24	256
5	1	2,3	5,0	11,6	25	125	32	120	3125
6	1	2,6	6,0	15,5	36	216	64	720	46656
7	1	2,8	7,0	19,7	49	343	128	5040	823543
8	1	3,0	8,0	24,0	64	512	256	40320	16777216
9	1	3,2	9,0	28,5	81	729	512	362880	387420489
10	1	3,3	10,0	33,2	100	1000	1024	3628800	10000000000
11	1	3,5	11,0	38,1	121	1331	2048	39916800	285311670611
12	1	3,6	12,0	43,0	144	1728	4096	479001600	8916100448256
13	1	3,7	13,0	48,1	169	2197	8192	6227020800	302875106592253
14	1	3,8	14,0	53,3	196	2744	16384	87178291200	11112006825558000
15	1	3,9	15,0	58,6	225	3375	32768	1307674368000	437893890380859000

Complexidade de Algoritmos Iterativos

- Para calcular a complexidade basta identificar as repetições e a operação dominante:

Algoritmo 1:

```
Entrada: Vetor V com n elementos
1 para i ← 1 ... n faça
2     min ← i
3     para j ← i + 1 ... n faça
4         se V[j] < V[min] então
5             min ← j
6     aux ← V[i]
7     V[i] ← V[min]
8     V[min] ← aux
9 retorne V
```

Se a lista tem n elementos, vamos repetir n vezes.

Se a lista tem n elementos, vamos repetir no máximo n vezes.

Operação Dominante: Comparações

Logo: $\boldsymbol{n * n * 1 \Rightarrow O(n^2)}$

Complexidade de Algoritmos Recursivos

Algoritmo 7: Qual é a complexidade?

Entrada: Vetor V com n posições
Saída: Vetor V ordenado
Chamada Inicial: `CocktailSort` $(V, 0, n-1)$

```
1  Procedimento CocktailSort(V, ini, L)
2      se ini < fim então
3          menor ← ini
4          maior ← ini
5          para i ← ini + 1 ... fim faça
6              se V[i] < V[menor] então menor ← i
7              senão se V[i] > V[maior] então maior ← i
8          V[ini] ↔ V[menor]
9          V[fim] ↔ V[maior]
10         CocktailSort (V, ini + 1, fim − 1)
```

Complexidade de Algoritmos Recursivos

- $T(n) = \begin{cases} \boldsymbol{O(n)} + T(n-2), & se\ n > 1 \\ 0, & caso\ contrário \end{cases}$

- Desenvolvendo a equação:
- $T(n) = O(n) + T(n-2) + \ldots$
- $T(n) = O(n) + O(n-2) + T(n-4) + \cdots$
- $n - 2k = 0 \Rightarrow k < n$
- $T(n) = n * O(n)$
- Logo, a complexidade de pior caso do Algoritmo 7 é $O(n^2)$

E qual é a complexidade do Algoritmo 2?

Algoritmo 2:

Entrada: Vetor V com n elementos

Função `Ajustar`(V, ini, fim)

```
1  se ini < fim então
2      meio ← (ini + fim)/2
3      Ajustar(V, ini, meio)
4      Ajustar(V, meio + 1, fim)
5      Unir(V, ini, meio, fim)
6  retorne V
```

Função `Unir`$(V, ini, meio, fim)$

```
7  x ← fim − ini + 1
8  Criar vetor A com x posições
9  p1 ← ini        p2 ← fim
10 para i ← 1 ... x faça
11     se p1 ≤ meio ∧ p2 ≤ fim então
12         se V[p1] < V[p2] então A[i] ← V[p1]        p1 ← p1 + 1;
13         senão A[i] ← V[p2]        p2 ← p2 + 1;
14     senão
15         se p1] ≤ meio então A[i] ← V[p1]        p1 ← p1 + 1;
16         senão A[i] ← V[p2]        p2 ← p2 + 1;
17 para i ← 1 ... x faça  V[ini + i] ← A[i] ;
18 retorne V
```

Qual é a complexidade da Função Unir?

Algoritmo 2:

Entrada: Vetor V com n elementos
Função `Ajustar`(V, ini, fim)
1 **se** $ini < fim$ **então**
2 $meio \leftarrow (ini + fim)/2$
3 `Ajustar`$(V, ini, meio)$
4 `Ajustar`$(V, meio + 1, fim)$
5 `Unir`$(V, ini, meio, fim)$
6 **retorne** V

Função `Unir`$(V, ini, meio, fim)$
7 $x \leftarrow fim - ini + 1$
8 Criar vetor A com x posições
9 $p_1 \leftarrow ini \qquad p_2 \leftarrow fim$
10 **para** $i \leftarrow 1 \ \ldots \ x$ **faça**
11 **se** $p_1 \leq meio \wedge p_2 \leq fim$ **então**
12 **se** $V[p_1] < V[p_2]$ **então** $A[i] \leftarrow V[p_1] \qquad p_1 \leftarrow p_1 + 1;$
13 **senão** $A[i] \leftarrow V[p_2] \qquad p_2 \leftarrow p_2 + 1;$
14 **senão**
15 **se** $p_1] \leq meio$ **então** $A[i] \leftarrow V[p_1] \qquad p_1 \leftarrow p_1 + 1;$
16 **senão** $A[i] \leftarrow V[p_2] \qquad p_2 \leftarrow p_2 + 1;$
17 **para** $i \leftarrow 1 \ldots \ x$ **faça** $V[ini + i] \leftarrow A[i]$;
18 **retorne** V

Qual é a complexidade da Função Unir?

```
Função Unir(V, ini, meio, fim)
7     x ← fim − ini + 1
8     Criar vetor A com x posições
9     p1 ← ini        p2 ← fim
10    para i ← 1 ... x faça
11        se p1 ≤ meio ∧ p2 ≤ fim então
12            se V[p1] < V[p2] então A[i] ← V[p1]        p1 ← p1 + 1;
13            senão A[i] ← V[p2]        p2 ← p2 + 1;
14        senão
15            se p1] ≤ meio então A[i] ← V[p1]        p1 ← p1 + 1;
16            senão A[i] ← V[p2]        p2 ← p2 + 1;
17    para i ← 1 ... x faça V[ini + i] ← A[i] ;
18    retorne V
```

Se a lista tem n elementos, vamos repetir n vezes.

No pior caso:
3 comparações para cada item da lista

Não são comparações, mas observamos que impactam o algoritmo.

Qual é a complexidade da Função Unir?

- Pior caso: 3 comparações para cada item da lista
- São n elementos na lista
- Logo, no pior caso são $3n$ comparações

- Ou seja: A função unir demanda $O(n)$

E a função ajustar?

Algoritmo 2:

Entrada: Vetor V com n elementos — Para cada lista com n elementos:

Função `Ajustar`(V, ini, fim)

```
1  se ini < fim então
2      meio ← (ini + fim)/2
3      Ajustar(V, ini, meio)          — Chamada a Ajustar para uma lista com a primeira metade dos elementos
4      Ajustar(V, meio + 1, fim)      — Chamada a Ajustar para outra lista com a segunda metade dos elementos
5      Unir(V, ini, meio, fim)        — Chamada ao procedimento Unir para a lista com os n elementos
6  retorne V
```

Complexidade da Função Ajustar

- $T(n) = \begin{cases} \boldsymbol{O(n)} + \boldsymbol{2T(n/2)}, & se\ n > 1 \\ 0, & caso\ contrário \end{cases}$

- Onde:
  - $\boldsymbol{O(n)}$ é a complexidade da Função Unir.
  - $\boldsymbol{2T(n/2)}$ é a complexidade da chamada recursiva a função Ajustar, considerando duas listas com $n/2$ elementos cada.

- $T(n) = O(n) + 2T(n/2)$
- $T(n) = O(n) + 2O\left(\frac{n}{2}\right) + 2T(n/4)$

Complexidade da Função Ajustar

- $T(n) = O(n) + 2O\left(\frac{n}{2}\right) + 2T(n/4)$
- $T(n) = O(n) + 2O\left(\frac{n}{2}\right) + 2O\left(\frac{n}{4}\right) + 2T(n/8)$

- Vamos reescrever considerando potências de 2:
- $T(n) = O(n) + 2O\left(\frac{n}{2^1}\right) + 2O\left(\frac{n}{2^2}\right) + 2T(n/2^3)$

- Quantas vezes podemos continuar dividindo por 2 até atingir listas de tamanho unitário?

Complexidade da Função Ajustar

- $T(n) = O(n) + 2O\left(\frac{n}{2^1}\right) + 2O\left(\frac{n}{2^2}\right) + 2T(n/2^3)$

- Quantas vezes podemos continuar dividindo por 2 até atingir listas de tamanho unitário?
- $\frac{n}{2^k} = 1$
- $2^k = n$
- $k = log_2 n$

- Isso é, podemos dividir no máximo $log_2 n$ vezes

Complexidade da Função Ajustar

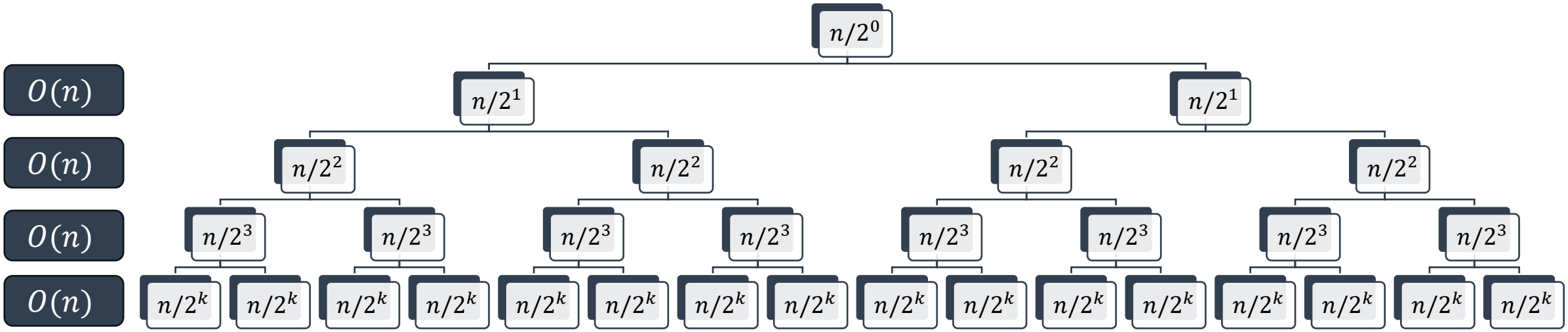
- $T(n) = O(n) + 2O\left(\frac{n}{2}\right) + 2O\left(\frac{n}{4}\right) + 2T(n/8)$
- $T(n) = \underbrace{O(n) + O(n) + \ldots +}_{log_2\, n \text{ vezes}} O(n)$

- $T(n) = log_2\, n * O(n)$
- $T(n) = O(n\, log_2\, n)$

- Logo: Algoritmo 3 é $O(n\, log_2\, n)$

Complexidade do Algoritmo 3

- Outra forma de visualizar:



- Cada "andar" gasta no total $O(n)$ para juntar as partes da lista com n elementos
- Existem no máximo $log_2\ n$ andares (até atingir listas de tamanho 1)
- Logo, no total, o algoritmo tem complexidade $O(n\ log_2\ n)$

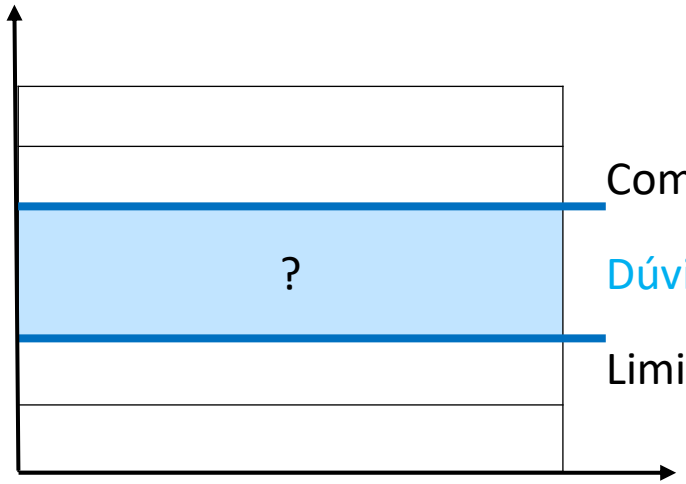
Algoritmos Ótimos

- Para o problema de ordenar uma lista com n elementos já observamos algoritmos de complexidade $O(n^2)$ e $O(n\ log_2\ n)$
- Qual é melhor?
  - Algoritmos de complexidade $O(n\ log_2\ n)$

- Mas dá para fazer ainda melhor?
- Esses algoritmos de complexidade $O(n\ log_2\ n)$ são ótimos?

Algoritmos Ótimos

Complexidades

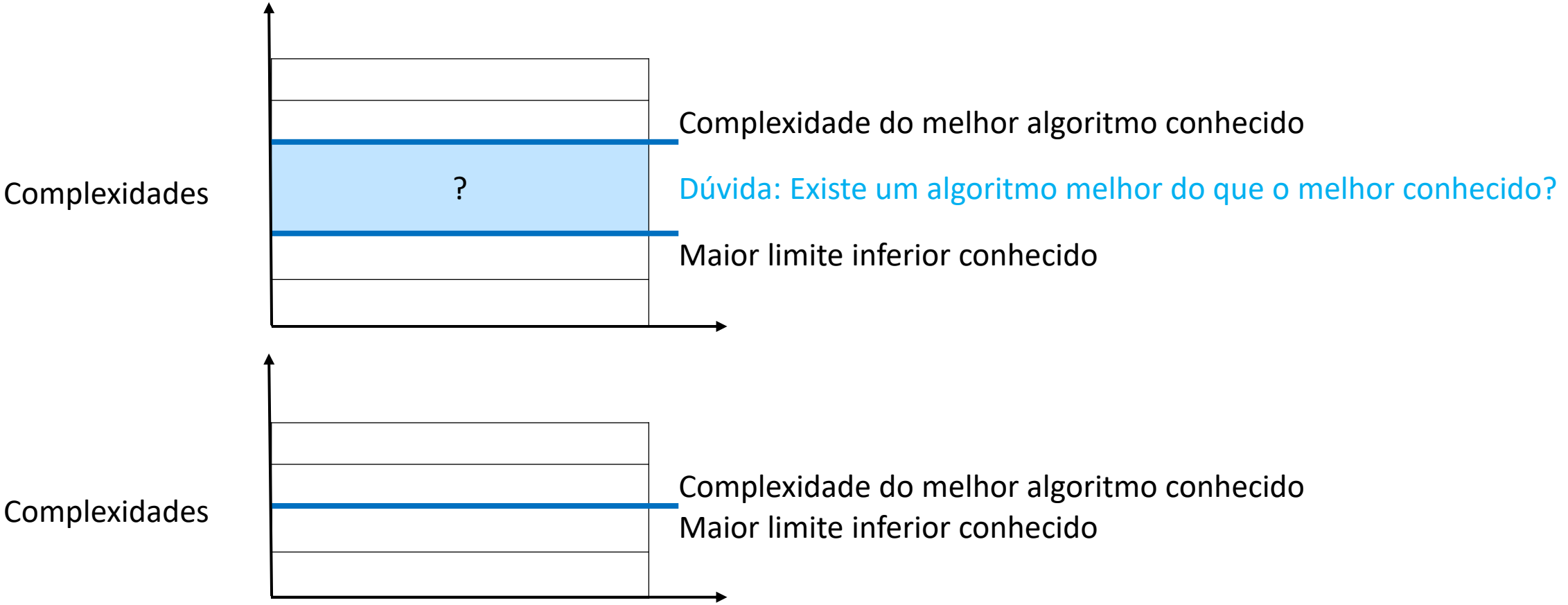


Complexidade do melhor algoritmo conhecido: $O(n\ log_2\ n)$

Dúvida: Existe um algoritmo melhor do que o melhor conhecido?

Limite mínimo para um algoritmo de ordenação

Algoritmos Ótimos



Quando o limite inferior foi provado matematicamente e conhecemos um algoritmo com essa complexidade, podemos dizer que esse algoritmo é ótimo!
Ótimo, pois não é possível que nenhum algoritmo seja ainda melhor.

Algoritmos Ótimos: Exemplos

- Ordenação de listas:
  - Complexidade do melhor algoritmo conhecido: $O(n\ log_2\ n)$
  - Limite inferior para o problema: $O(n\ log_2\ n)$
  - Logo, um algoritmo de complexidade $O(n\ log_2\ n)$ é **ótimo**

- Obter o produto de matrizes:
  - Complexidade do algoritmo "ingênuo": $O(n^3)$
  - Complexidade do melhor algoritmo conhecido: $O(n^{2,37})$
  - Limite inferior para o problema: $\boldsymbol{Desconhecido}$
  - Como não sabemos o limite inferior, não podemos afirmar se um algoritmo é ótimo ou não...

Classes de Algoritmos

- Algoritmos de tempo **constante**:
  - A complexidade não depende do tamanho da entrada.
  - $O(1)$

- Por exemplo:
  - Retornar o elemento central de uma lista
  - Verificar se o primeiro elemento de uma lista é maior do que um determinado valor

Classes de Algoritmos

- Algoritmos de tempo **linear**:
  - A complexidade é cresce linearmente com o tamanho da entrada
  - $O(log_2 n)$
  - $O(n)$

- Em geral, esses algoritmos são obtidos através da análise da estrutura.
- São algoritmos mais "inteligentes".

Classes de Algoritmos

- Algoritmos de tempo **polinomial**:
  - A complexidade é determinada por um polinômio da entrada
  - $O(n \log_2 n)$
  - $O(n^2)$
  - $O(n^3)$
  - $O(n^c)$ (assumindo que c é uma constante)

- Em geral, esses algoritmos são obtidos através da análise da estrutura. São algoritmos mais "inteligentes".

Classes de Algoritmos

- Algoritmos **exponenciais**:
  - A complexidade é determinada por uma função exponencial em relação a entrada
  - $O(2^n)$
  - $O(3^n)$
  - $O(n!)$
  - $O(n^n)$

- Geralmente são algoritmos baseados em força bruta: Testam todas as combinações possíveis de resposta em busca de uma solução viável para o problema (ou da melhor solução possível)
- Determinar se um algoritmo é polinomial ou exponencial é extremamente importante principalmente quando o tamanho da entrada é significativo (grande).

Classes de Algoritmos

- Algoritmos de tempo constante
- Algoritmos de tempo linear
- Algoritmos de tempo polinomial
- Algoritmos de tempo exponencial

Para entradas grandes,
esse é o limite da computação

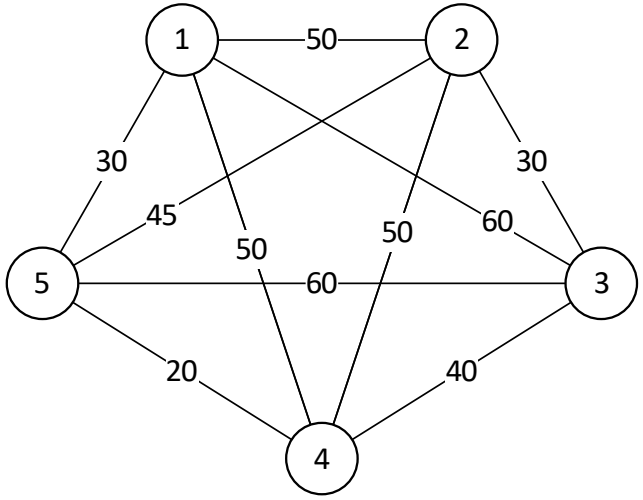
Exemplo de Algoritmo Exponencial

- Caixeiro viajante é uma profissão bastante antiga, onde o vendedor vende produtos de porta em porta. Antigamente, quando não havia facilidade do transporte entre cidades, os caixeiros-viajantes eram responsáveis por transportar produtos entre diferentes cidades.

- Em termos mais atuais, podemos considerar que transportadoras fazem o papel do caixeiro viajante. Uma das questões mais interessantes nesta área é definir a melhor rota entre as cidades. Até hoje esse problema é conhecido como o problema do caixeiro viajante.

Exemplo de Algoritmo Exponencial

Mapa das Cidades



Custos do transporte

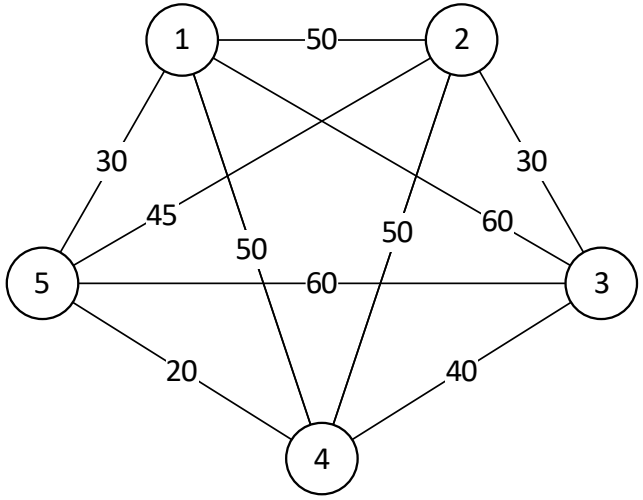
	1	2	3	4	5
1	0	50	60	50	30
2	50	0	30	50	45
3	60	30	0	40	60
4	50	50	40	0	20
5	30	45	60	20	0

- Como podemos encontrar a rota de menor custo que começa na cidade 3, passa por todas as cidades e retorna para a cidade 3?

- Ideia: Vamos escolher sempre o caminho de menor custo que sai da cidade atual e vai para uma cidade ainda não visitada.

Exemplo de Algoritmo Exponencial

Mapa das Cidades



Custos do transporte

	1	2	3	4	5
1	0	50	60	50	30
2	50	0	30	50	45
3	60	30	0	40	60
4	50	50	40	0	20
5	30	45	60	20	0

- Partindo da cidade: 3
- Solução gulosa: $3 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1 \rightarrow 3$ Custo: 210
- Melhor solução: $3 \rightarrow 2 \rightarrow 1 \rightarrow 5 \rightarrow 4 \rightarrow 3$ Custo: 170
- A ideia inicial não foi boa... E agora?

Exemplo de Algoritmo Exponencial

- Até hoje não se conhece um algoritmo polinomial capaz de resolver o problema do caixeiro viajante.

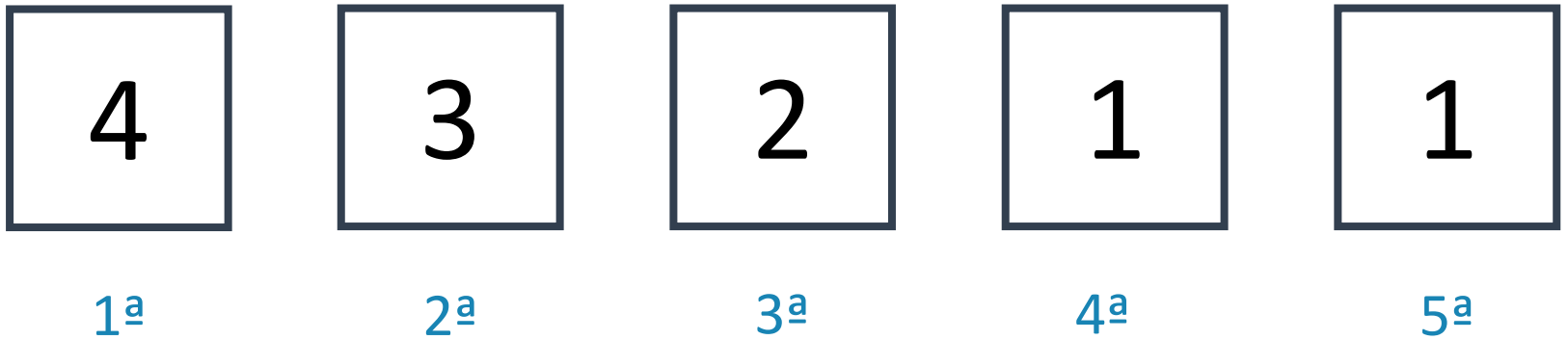
- Nesse caso temos duas possibilidades:
  - Usamos um algoritmo de força bruta, que vai testar todas as combinações possíveis de respostas... Esse tipo de algoritmo é geralmente exponencial.
  - No caso do Caixeiro Viajante: $O(n!)$

OU

  - Abrimos mão da solução ótima para conseguir uma boa solução em uma quantidade viável de tempo... Nesse caso vamos usar métodos aproximativos
  - No caso do Caixeiro Viajante: $O(n^2)$

Exemplo de Algoritmo Exponencial

- Considerando que são 5 cidades (além do ponto de partida):



- A primeira cidade (o ponto de partida) faz parte da entrada
- Partindo da primeira cidade, podemos ir para 4
- Na segunda, podemos ir para 3
- E assim, sucessivamente. Até que na última, precisamos retornar a primeira.
- No total, temos 4 x 3 x 2 x 1 combinações: 4! = 24 rotas possíveis.

Exemplo de Algoritmo Exponencial

- Se existem n cidades, então o algoritmo baseado em força bruta para o problema do caixeiro viajante tem complexidade $O(n!)$

Cidades	Rotas Possíveis (n!)
10	3628800
20	2432902008176640000
30	265252859812191000000000000000000
40	815915283247898000000000000000000000000000
50	30414093201713400000000000000000000000000000000000000000000000
100	Um número com 158 dígitos...
200	Um número com 375 dígitos...
1000	Um número com 2568 dígitos...

Estima-se que todo o Universo* tenha 10^{80} átomos (* apenas a parte visível a partir da Terra)
Considerando 60 cidades, existem $60! \cong 8{,}32 \times 10^{81}$ rotas

Logo existem menos átomos no Universo* do que rotas possíveis entre 60 cidades...

Avaliação dos Problemas

- Até agora avaliamos a complexidade dos **algoritmos**

- Mas como avaliar a dificuldade de um determinado **problema**?
- É possível resolver um problema computacionalmente em uma quantidade viável de tempo (antes do fim da Terra, por exemplo)?

- Problemas P, NP, NP-Completo, NP-Difícil, co-NP, etc...
- Tema das próximas aulas.

Qual é a complexidade de pior caso?

```
void AlgoritmoA(int V[], int n)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < (n -1) - i; j++)
            if (V[j] > V[j+1])
                Trocar(&V[j],&V[j+1]);
}

void Trocar(int *a, int* b)
{
    int aux = *a;
    *a = *b;
    *b = aux;
}
```

Qual é a complexidade de pior caso?

```
void AlgoritmoB(int V[], int tam)
{
    int trocou = 1;
    while (trocou == 1)
    {
        trocou = 0;
        for (int i = 0; i < tam - 1; i++)
            if (V[i] > V[i+1])
            {
                Trocar(&V[i],&V[i+1]);
                trocou = 1;
            }
    }
}
```

```
void Trocar(int *a, int* b)
{
    int aux = *a;
    *a = *b;
    *b = aux;
}
```

Qual é a complexidade de pior caso?

```
int AlgoritmoC(int lista[], int n, int chave)
{
    for (int i = 0; i < n; i++)
    {
        if (lista[i] == chave)
            return i;
    }
    return -1;
}
```

Qual é a complexidade de pior caso?

```
int AlgoritmoD(int listaOrd[], int n, int chave)
{
    int i = 0;
    while (i < n && listaOrd[i] < chave)
        i++;
    return i;
}
```

Qual é a complexidade de pior caso?

```
void AlgoritmoE(int n, int matA[n][n], int matB[n][n], int matR[n][n])
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            matR[i][j] = 0;
            for (int k = 0; k < n; k++)
            {
                matR[i][j] += matA[i][k] * matB[k][j];
            }
        }
    }
    return 0;
}
```

Qual é a complexidade de pior caso?

```
void AlgoritmoF(int vet[], int tam)
{
    if (tam > 0)
    {
        int pmax = 0;
        for (int i = 1; i < tam; i++)
            if (vet[i] > vet[pmax])
                pmax = i;

        int aux = vet[tam-1];
        vet[tam-1] = vet[pmax];
        vet[pmax] = aux;

        AlgoritmoF(vet, tam-1);
    }
}
```

Complexidade de algoritmos recursivos

- Sempre que temos um algoritmo recursivo, precisamos entender como ele divide a entrada.
- O AlgoritmoF pode ser dividido em duas partes:
  1. Encontrar o maior elemento do vetor
  2. Deslocar o maior elemento para o final do vetor
  3. Chamar o AlgoritmoF para um "subvetor" (da posição 0 até tam-1)
  4. Se o vetor se tornar vazio (tamanho = 0), o problema foi resolvido

Custo de encontrar o maior e mover para o final

Custo de resolver o problema para um vetor com um elemento a menos

$$T(n) = \begin{cases} O(n) + T(n-1), & se\ n > 0 \\ O(1), & se\ n = 0 \end{cases}$$

Se o vetor está vazio (n=0) então a solução é trivial.

Complexidade de algoritmos recursivos

- $T(n) = \begin{cases} O(n) + T(n-1), & se\ n > 0 \\ O(1), & se\ n = 0 \end{cases}$

- $T(n) = O(n) + T(n-1)$
- $T(n) = O(n) + O(n-1) + T(n-2)$
- $T(n) = O(n) + O(n-1) + O(n-2) + \cdots + O(1)$

- Como calcular T(n)?
- Soma dos termos de uma PA

Complexidade de algoritmos recursivos

- $T(n) = \begin{cases} O(n) + T(n-1), & se\ n > 0 \\ O(1), & se\ n = 0 \end{cases}$

- $T(n) = O(n) + O(n-1) + O(n-2) + \cdots + O(1)$
- $T(n) = O(1) + O(2) + O(3) + \cdots + O(n)$
- $2T(n) = O(n+1) + O(n-1+2) + \cdots + O(n+1)$
- $2T(n) = n * O(n+1)$

- $T(n) = \frac{O(n^2+n)}{2} = O(n^2)$