



Computabilidade

Aula 1 – Introdução à Disciplina



Aulas

- Sextas:
 - Horário: 09h40 até 12h20

Calendário das Aulas (provisório)

Data	Conteúdo
24/03/2023	Apresentação da Disciplina; Revisão: Complexidade Computacional
31/03/2023	Revisão de Automatos e Máquina de Turing
07/04/2023	Feriado Paixão de Cristo
14/04/2023	Classes de Problemas e o Primeiro Problema NP-Completo;
21/04/2023	Feriado Tiradentes
28/04/2023	Introdução à Redução Polinomial e Os 21 Problemas de Karp
05/05/2023	Apresentação dos Trabalhos T1 : Problemas NP-Completo
...	...
26/07/2023	PF

Avaliações

- Provas escritas:
 - Pelo menos uma (regimento da UERJ)
- Trabalhos:
 - Provavelmente dois trabalhos:
 - T1: Problemas Intratáveis Computacionalmente
 - T2: Tema a ser definido
- Avaliação de Reposição:
 - Única para o período – conteúdo todo
 - Requisição segue as regras da UERJ
- Média Semestral:
 - Média simples entre todas as notas (provas e trabalhos)
- Prova final: PF

Regras

- A presença nas aulas será contabilizada (por listas de presença)
- Frequência mínima (regimento da UERJ): 75% das aulas
- Aprovação (alunos com presença $\geq 75\%$):
 - Média Semestral ≥ 7.0 (aprovação direta - sem prova final)
 - $(\text{Média Semestral} + \text{PF}) / 2 \geq 5.0$ (aprovado com prova final)
- Reprovação:
 - Presença $< 75\%$ (sem direito a prova final)
 - Média Semestral < 4.0 (sem direito a prova final)
 - $(\text{Média Semestral} + \text{PF}) / 2 < 5.0$

Material das Aulas e Atividades

- Todo o material das aulas, incluindo as atividades propostas, será postado no Google Classroom da disciplina.
- Link:
<https://classroom.google.com/c/NjAwOTIzMjM1NjA3?cjc=cltfict>
- Ou leia o QR-Code ao lado.



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  $i \leftarrow 1 \dots n$  faça
2    $min \leftarrow i$ 
3   para  $j \leftarrow i + 1 \dots n$  faça
4     se  $V[j] < V[min]$  então
5        $min \leftarrow j$ 
6    $aux \leftarrow V[i]$ 
7    $V[i] \leftarrow V[min]$ 
8    $V[min] \leftarrow aux$ 
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     $meio \leftarrow (ini + fim)/2$ 
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   $p_1 \leftarrow ini$        $p_2 \leftarrow fim$ 
10 para  $i \leftarrow 1 \dots x$  faça
11   se  $p_1 \leq meio \wedge p_2 \leq fim$  então
12     se  $V[p_1] < V[p_2]$  então  $A[i] \leftarrow V[p_1]$        $p_1 \leftarrow p_1 + 1;$ 
13     senão  $A[i] \leftarrow V[p_2]$        $p_2 \leftarrow p_2 + 1;$ 
14   senão
15     se  $p_1 \leq meio$  então  $A[i] \leftarrow V[p_1]$        $p_1 \leftarrow p_1 + 1;$ 
16     senão  $A[i] \leftarrow V[p_2]$        $p_2 \leftarrow p_2 + 1;$ 
17 para  $i \leftarrow 1 \dots x$  faça  $V[ini + i] \leftarrow A[i]$  ;
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

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

```
2  maior = vetor[0]
```

```
3  for x in vetor[1:]:
```

```
4      if x > maior:
```

```
5          maior = x
```

```
6  print(f"O maior é {maior}")
```

O algoritmo executa 6 instruções!



Como avaliar sem depender do computador?

- E a repetição? Os comandos dentro da repetição não são repetidos?

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

```
maior = vetor[0]  
for x in vetor[1:]:  
    if x > maior:  
        maior = x
```

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

```
print(f"O maior é {maior}")
```


Como avaliar sem depender do computador?

- E a repetição? Os comandos dentro da repetição não são repetidos?

`vetor = [...]` E se eu não conhecesse o vetor?
Vamos supor que ele tem n elementos

```
maior = vetor[0]
for x in vetor[1:]:
    if x > maior:
        maior = x
```

} Esses comandos são executados $n - 1$ vezes

```
print(f"O maior é {maior}")
```

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?

```
vetor = [ ... ]  
  
maior = vetor[0]  
for x in vetor[1:]:  
    if x > maior:  
        maior = x
```

```
print(f"O maior é {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?

```
vetor = [...]
```

```
maior = vetor[0]
```

```
menor = vetor[0]
```

```
for x in vetor[1:]:
```

```
    if x > maior:
```

```
        maior = x
```

```
    if x < menor:
```

```
        menor = x
```

$2(n - 1)$ comparações

Podemos melhorar esse número???

Como?

```
print(f"O maior é {maior}")
```

```
print(f"O menor é {menor}")
```


Como avaliar sem depender do computador?

- E agora?

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

```
maior = vetor[0]
menor = vetor[0]
for x in vetor[1:]:
    if x > maior:
        maior = x
    elif x < menor:
        menor = x
```

```
print(f"O maior é {maior}")
print(f"O menor é {menor}")
```

Complicou... ☹️

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

No pior caso, vamos ter que fazer todas as
duas perguntas...

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

Como avaliar sem depender do computador?

- E agora?

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

```
maior = vetor[0]  
menor = vetor[0]  
for x in vetor[1:]:  
    if x > maior:  
        maior = x  
    elif x < menor:  
        menor = x
```

```
print(f"O maior é {maior}")  
print(f"O menor é {menor}")
```

- Mas e no melhor caso?

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

- E da pra fazer uma média?

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

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

Para pensar:

```
vetor = [5,6,11,3,10,2,1,4,8,9,7,13,15,25,12]
fim = len(vetor) - 1;
for i in range(0,fim//2):
    aux = vetor[i]
    vetor[i] = vetor[fim - i]
    vetor[fim - i] = aux
print(vetor)
```

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

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

Seja t_i o número de passos efetuados 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]
x = int(input("Digite o número para buscar:"))
for i in range(0, len(vetor)-1):
    if vetor[i] == x:
        print(f"Encontrei na posição {i} do vetor")
        break
else:
    print("Não encontrei seu número no vetor")
```

- Qual é a complexidade de pior caso? **n comparações**
- Qual é a complexidade de melhor caso? **1 comparação**

Crescimento Assintótico de Funções

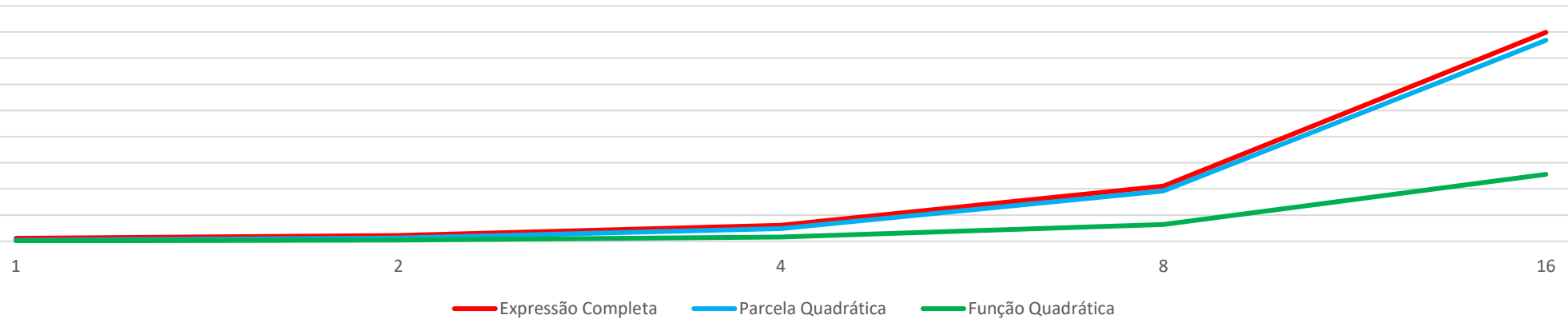
- Quando estudamos a complexidade de um algoritmo, estamos interessados em seu comportamento **assintótico**
- Isto é, o crescimento de sua complexidade para entradas suficientemente grandes.
- **Por que?**
- Simples: muitas vezes é difícil determinar o número exato de vezes que as instruções são executadas...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

Crescimento Assintótico de Funções

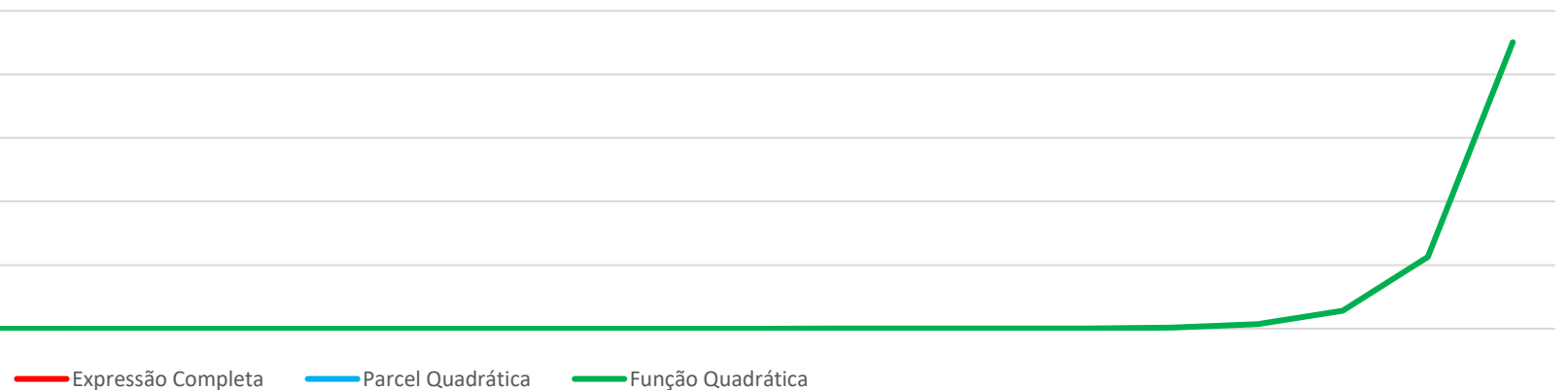
- Quando analisamos o comportamento assintótico de tais funções, podemos verificar que, à medida que o valor de n cresce, a parcela quadrática $3n^2$ torna-se dominante em relação às demais parcelas.
- Isto é: o valor de $3n^2$ se torna tão mais significativo do que $\frac{12}{8}n + 7$, que podemos considerar apenas a primeira parte sem nenhuma perda considerável.
- 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

Crescimento das Funções para Entradas Pequenas



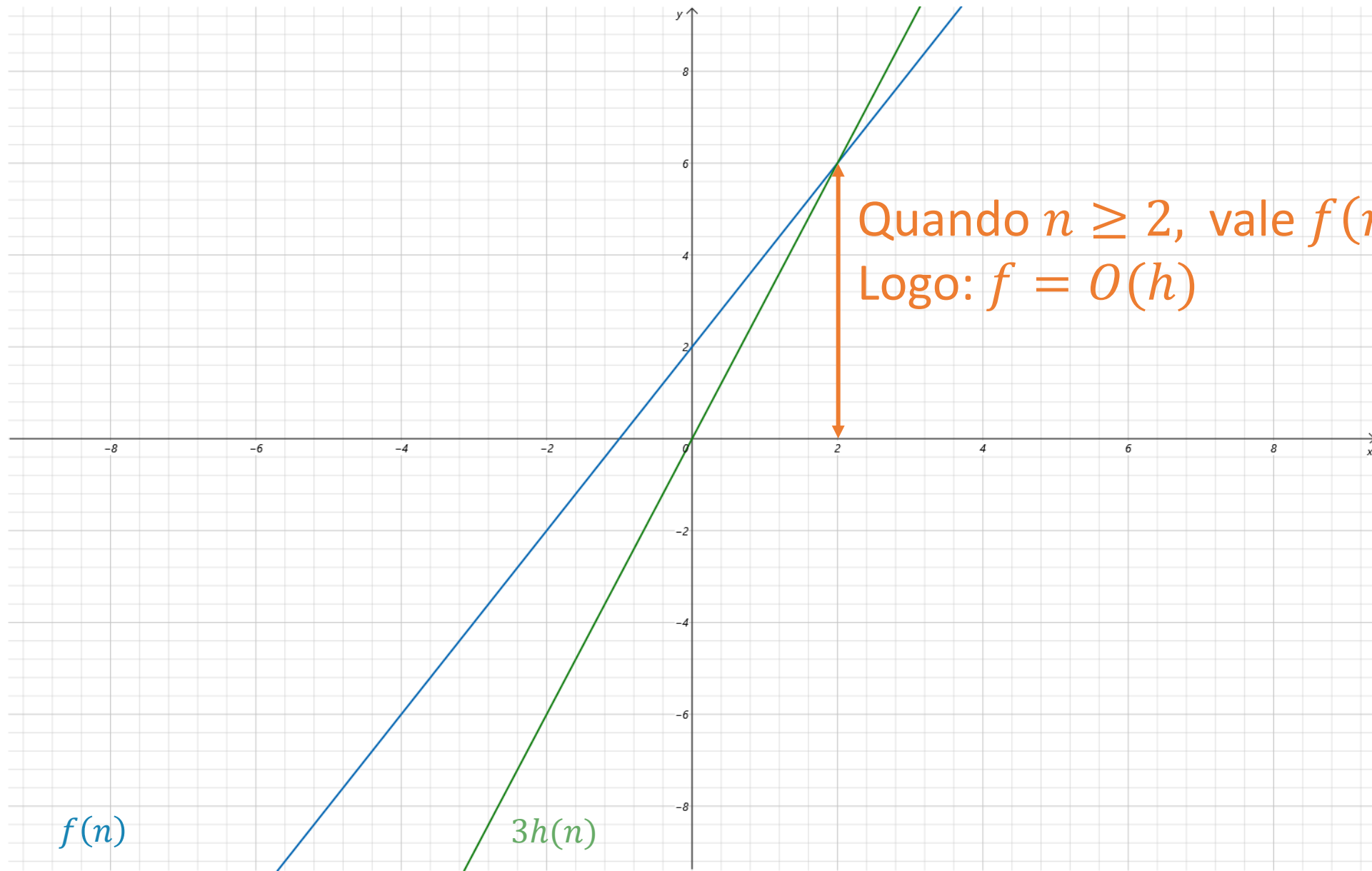
Crescimento Assintótico das Funções (Entradas Suficientemente Grandes)



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

Notação O



Quando $n \geq 2$, vale $f(n) \leq 3 \cdot h(n)$
Logo: $f = O(h)$

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

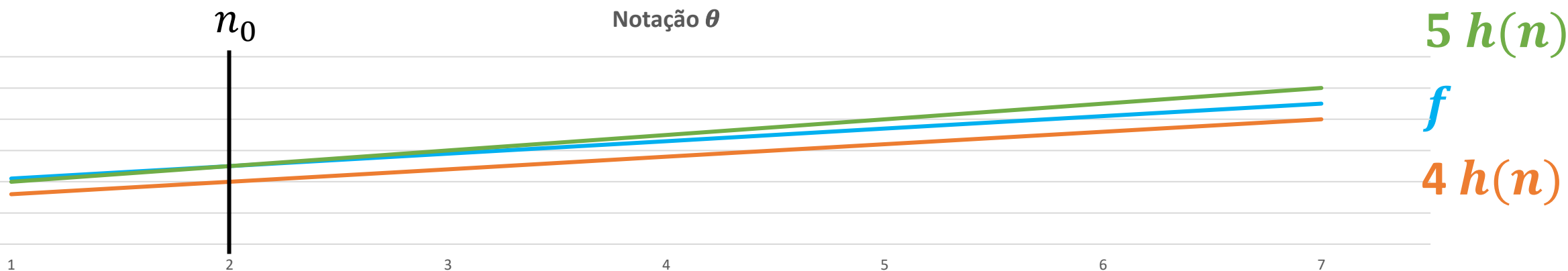
- Algoritmo para encontrar o máximo: $O(n)$
- Algoritmo para encontrar o máximo e o mínimo: $O(n)$
- Algoritmo para buscar um elemento no vetor: $O(n)$
- Algoritmo de ordenação MergeSort: $O(n \log n)$
- Algoritmo de ordenação BubbleSort: $O(n^2)$

Notação Ω

- Usada para definir limites assintoticamente inferiores
- Sejam f , h duas funções reais positivas
- Dizemos que f é $\Omega(h)$ ($f = \Omega(h)$) quando:
 - Existe uma constante $c > 0$
 - Existe um valor inteiro n_0
 - Tais que para todo $n > n_0$, vale $f(n) \geq c \cdot h(n)$
- Exemplo:
 - $f(n) = n^2 - 1 \Rightarrow f = \Omega(n^2)$
 - $f(n) = n^2 - 1 \Rightarrow f = \Omega(n)$
 - $f(n) = n^2 - 1 \Rightarrow f = \Omega(1)$
 - $f(n) = n^2 - 1 \not\Rightarrow f = \Omega(n^3)$

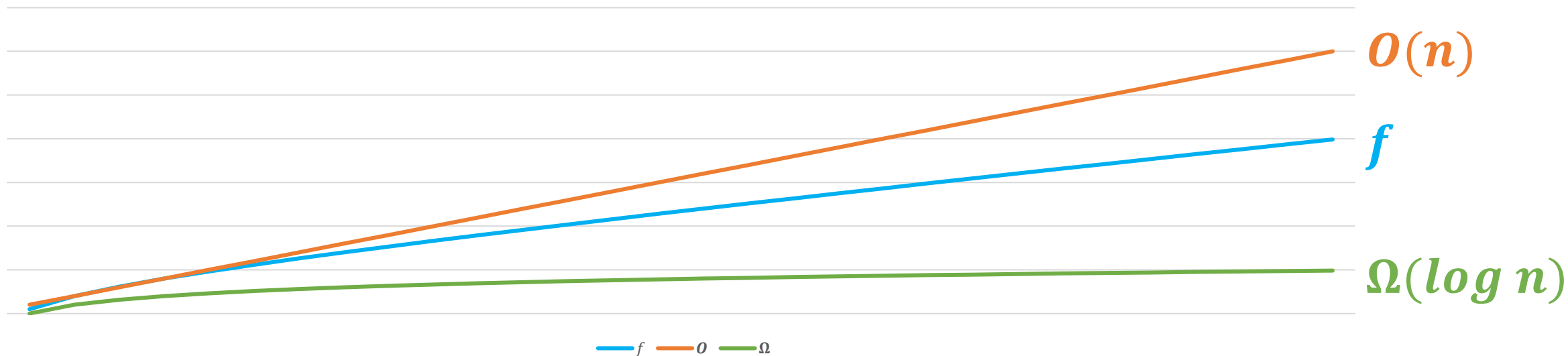
Notação θ

- 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 $\Theta(h)$ ($f = \Theta(h)$) quando:
 - Existem duas constantes $c_1 > 0$ 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

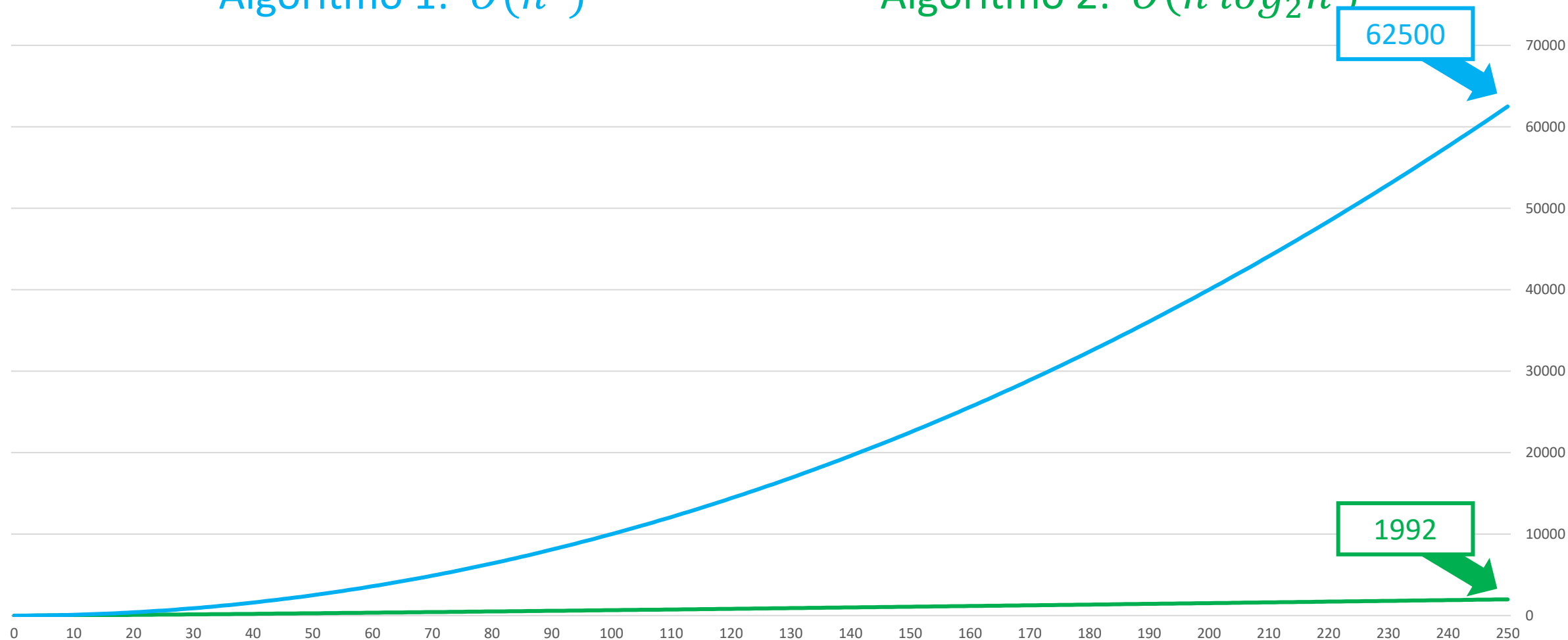
- Usamos para expressar respectivamente o pior e o melhor caso de um algoritmo. Por exemplo:
 - Um algoritmo $O(n)$ nunca vai executar **mais** do que n operações dominantes. É o limite superior ou a complexidade do **pior caso**.
 - Um algoritmo $\Omega(n^3)$ nunca vai executar **menos** do que n^3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

- $O(2^n)$

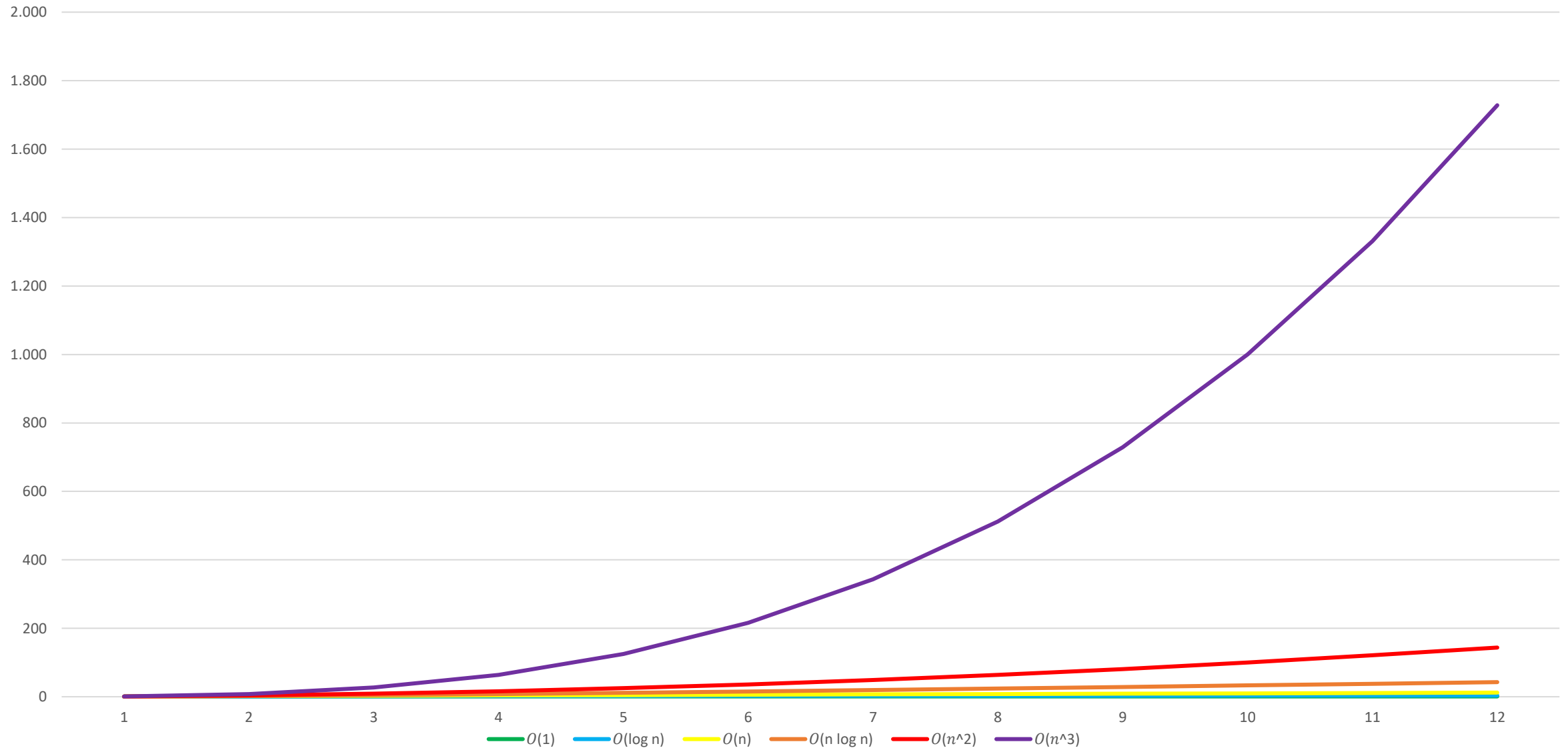
- $O(n!)$



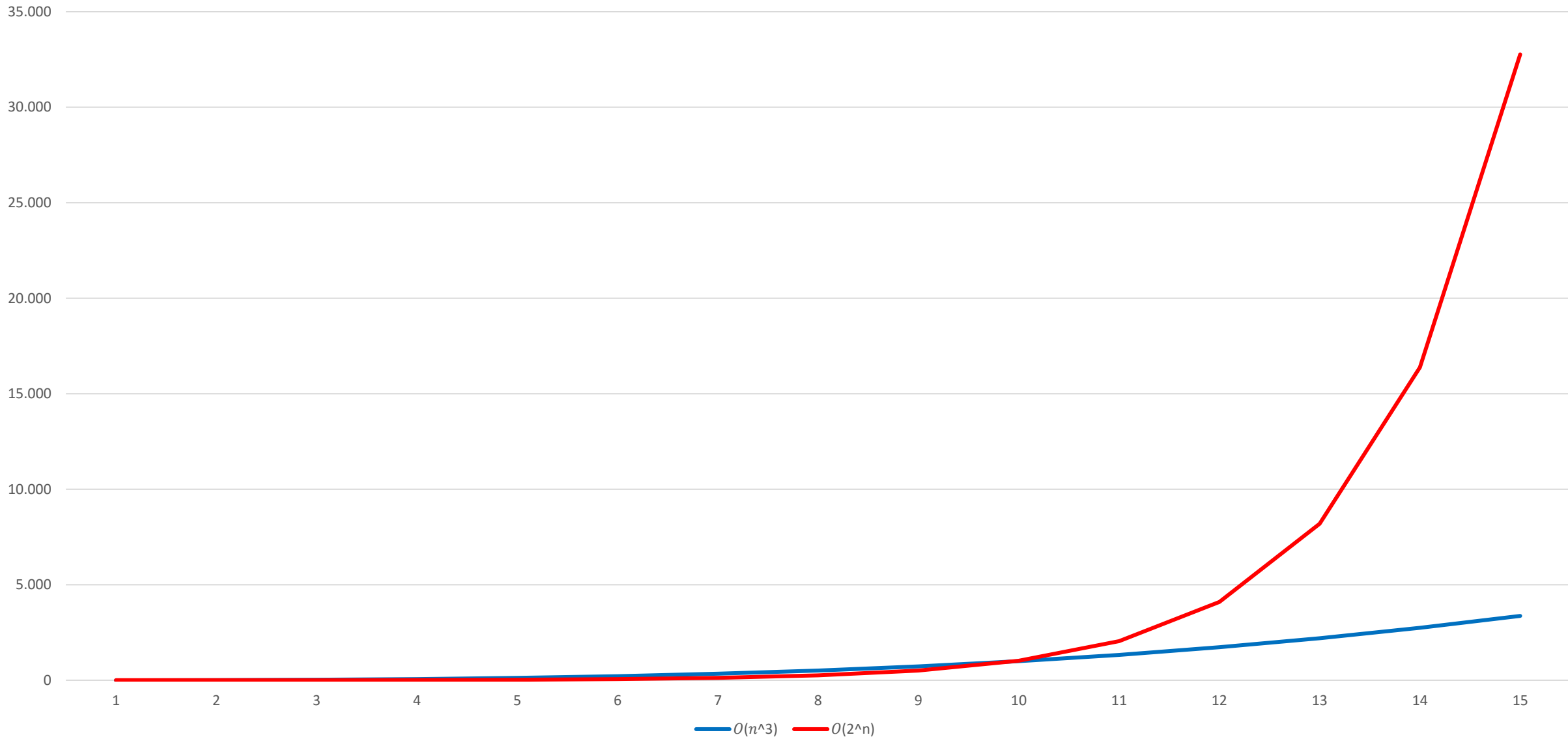
- $O(n^n)$

Para entradas grandes,
esse é o limite da computação

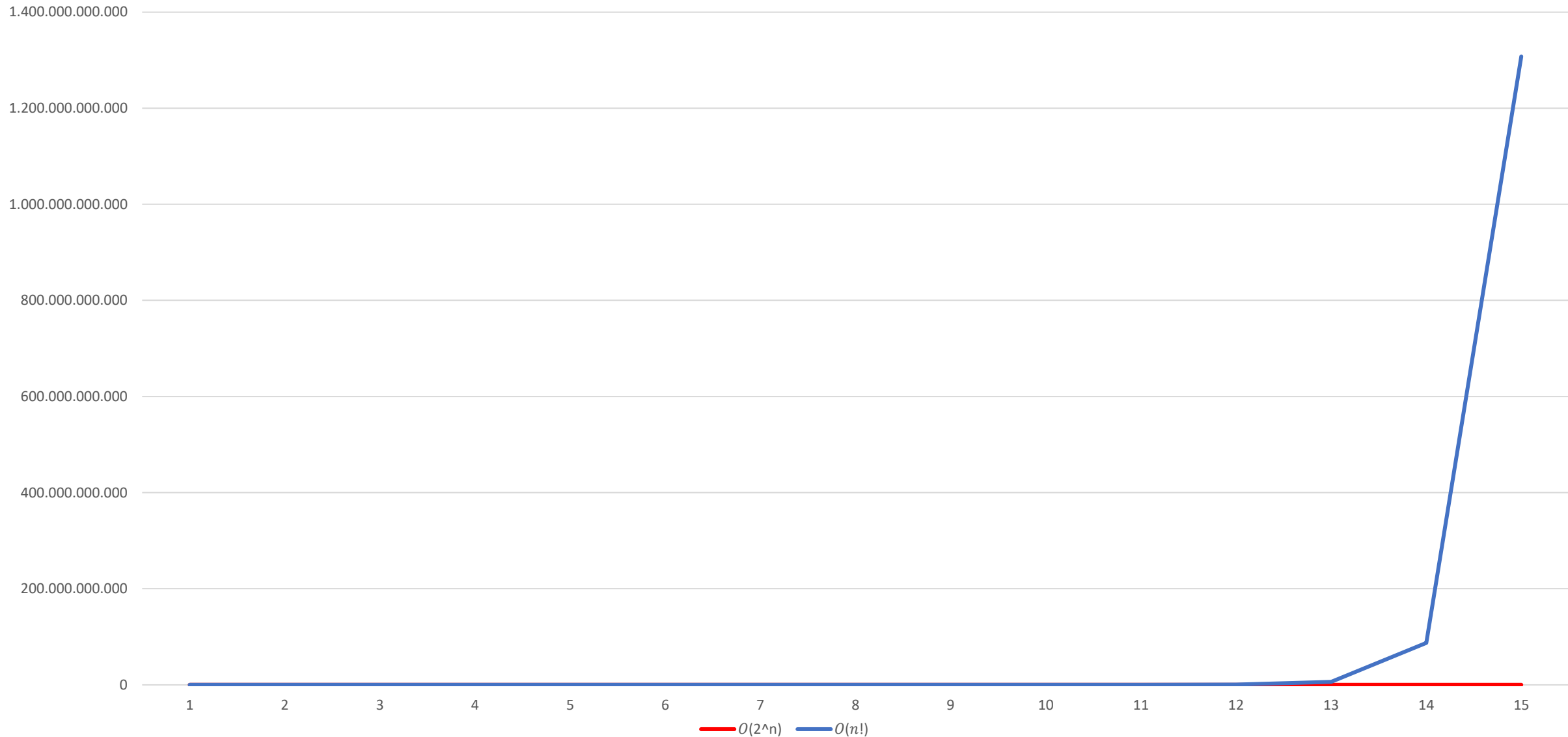
Comparação entre Algoritmos



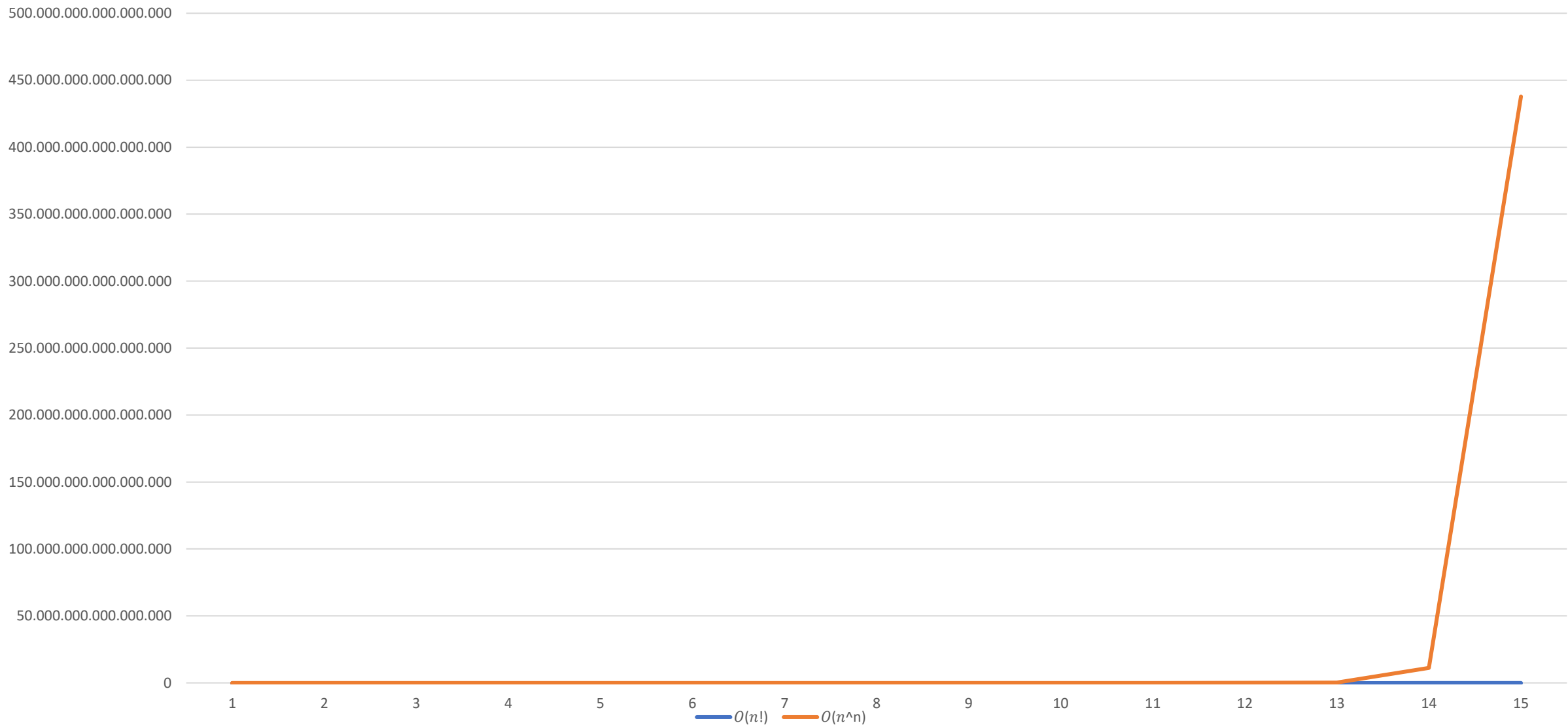
Comparação Entre Algoritmos



Comparação Entre Algoritmos



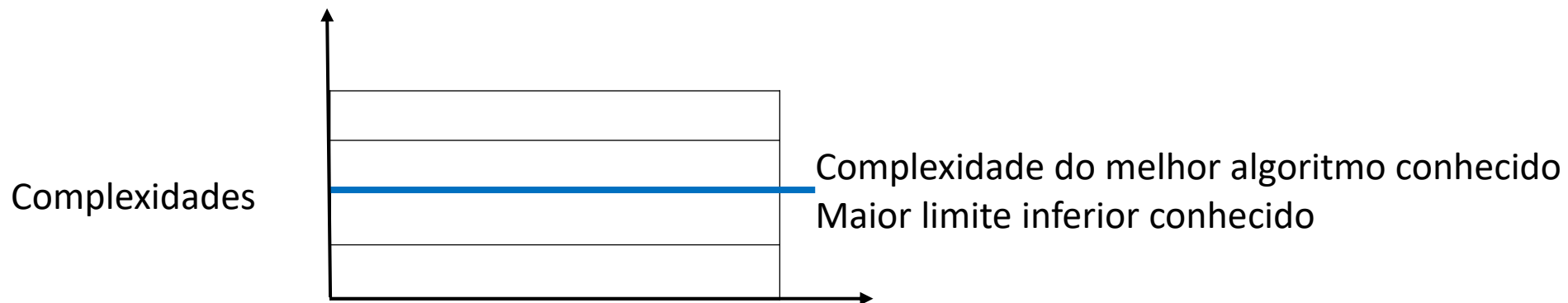
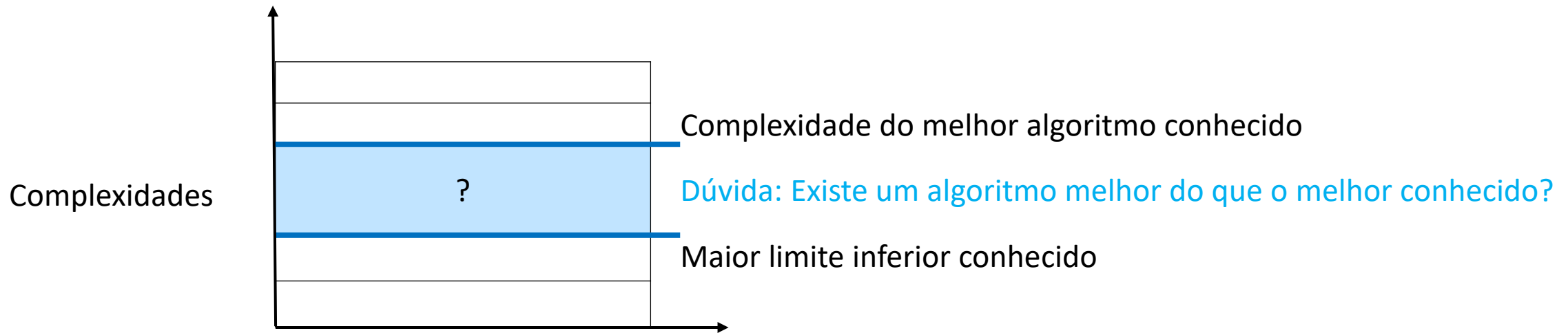
Comparação Entre Algoritmos



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

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: **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
-

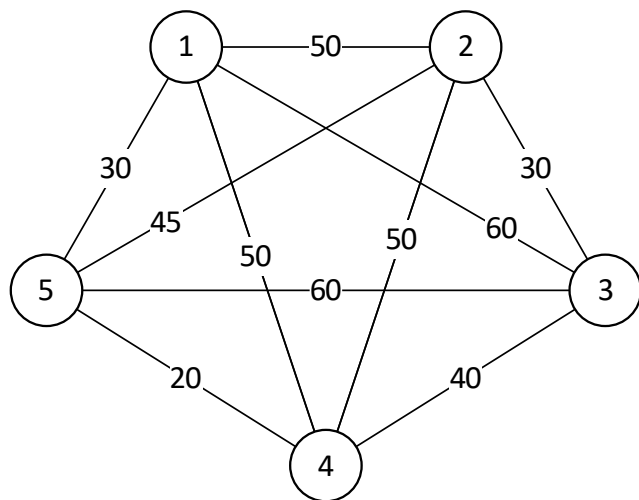
Para entradas grandes,
esse é o limite da computação

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



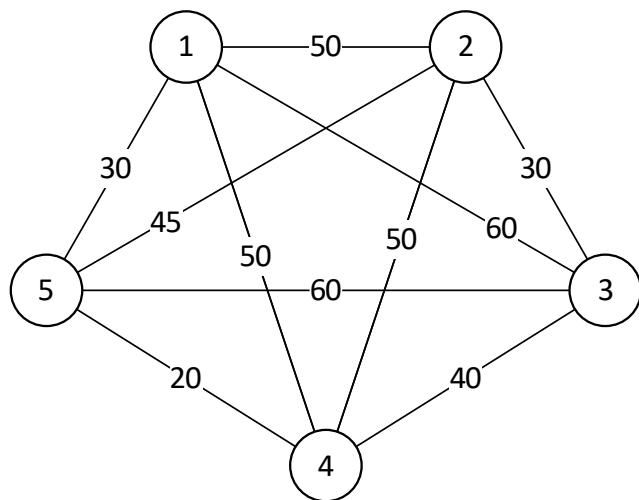
Custos do transporte

	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

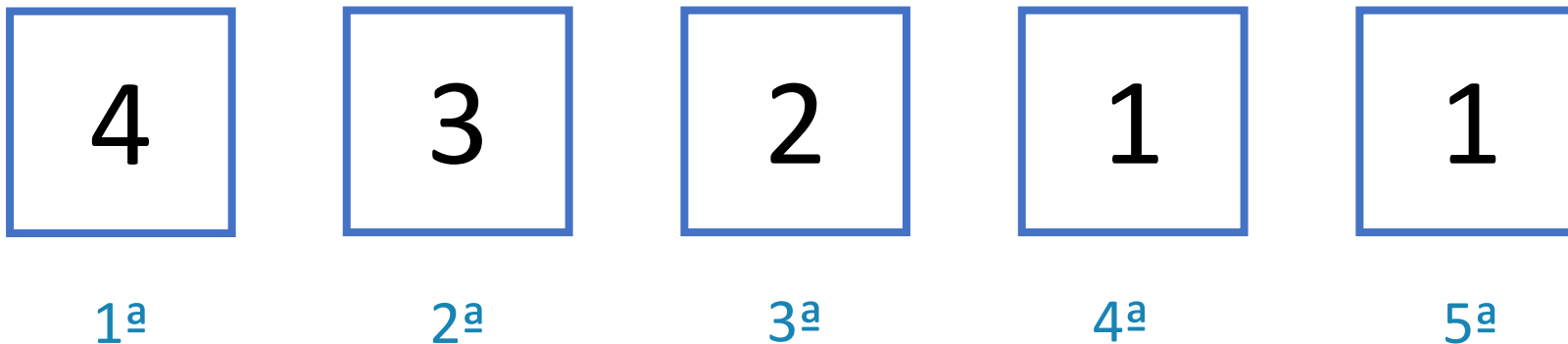
- Partindo da cidade: 3
- Solução gulosa: $3 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1 \rightarrow 3$ Custo: 210
- Melhor solução: $3 \rightarrow 2 \rightarrow 1 \rightarrow 5 \rightarrow 4 \rightarrow 3$ Custo: 170
- A ideia inicial não foi boa... E agora?

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:



- 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 $4 \times 3 \times 2 \times 1$ combinações: $4! = 24$ rotas possíveis.

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

[illegible]

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.