

Algoritmos e Estruturas de Dados I
Prof. Tiago Eugenio de Melo
tmelo@uea.edu.br

www.tiagodemelo.info

Observações

Observações

- O conteúdo dessa aula é parcialmente proveniente do Capítulo 11 do livro *“Fundamentals of Python – From First Programs Through Data Structures”*.

Observações

- O conteúdo dessa aula é parcialmente proveniente do Capítulo 11 do livro *“Fundamentals of Python – From First Programs Through Data Structures”*.
- As palavras com a fonte `Courier` indicam uma palavra-reservada da linguagem de programação.

Observações

- O conteúdo dessa aula é parcialmente proveniente do Capítulo 11 do livro “*Fundamentals of Python – From First Programs Through Data Structures*”.
- As palavras com a fonte `Courier` indicam uma palavra-reservada da linguagem de programação.
- Basta clicar na imagem para acessar as fontes das animações e figuras usadas nesse material.

Observações

- O conteúdo dessa aula é parcialmente proveniente do Capítulo 11 do livro “*Fundamentals of Python – From First Programs Through Data Structures*”.
- As palavras com a fonte `Courier` indicam uma palavra-reservada da linguagem de programação.
- Basta clicar na imagem para acessar as fontes das animações e figuras usadas nesse material.
- Os slides com o símbolo CO (COLAB) abaixo indicam que o código estará disponível on-line:

Observações

- O conteúdo dessa aula é parcialmente proveniente do Capítulo 11 do livro “*Fundamentals of Python – From First Programs Through Data Structures*”.
- As palavras com a fonte `Courier` indicam uma palavra-reservada da linguagem de programação.
- Basta clicar na imagem para acessar as fontes das animações e figuras usadas nesse material.
- Os slides com o símbolo CO (COLAB) abaixo indicam que o código estará disponível on-line:



Objetivos

Objetivos

- Após o término dessa aula você deverá ser capaz de:

Objetivos

- Após o término dessa aula você deverá ser capaz de:
 - Medir a performance de um algoritmo através do tempo e do número de instruções executadas com diferentes tamanhos de dados.

Objetivos

- Após o término dessa aula você deverá ser capaz de:
 - Medir a performance de um algoritmo através do tempo e do número de instruções executadas com diferentes tamanhos de dados.
 - Analisar a performance de um algoritmo e determinar a sua ordem de complexidade utilizando a notação big-O.

Objetivos

- Após o término dessa aula você deverá ser capaz de:
 - Medir a performance de um algoritmo através do tempo e do número de instruções executadas com diferentes tamanhos de dados.
 - Analisar a performance de um algoritmo e determinar a sua ordem de complexidade utilizando a notação big-O.
 - Diferenciar as ordens de complexidade de algoritmos.

Medindo a Eficiência de Algoritmos



Medindo a Eficiência de Algoritmos

- Na escolha de algoritmos, nós frequentemente temos que optar entre **espaço** e **tempo** (*tradeoff*).



Medindo a Eficiência de Algoritmos

- Na escolha de algoritmos, nós frequentemente temos que optar entre **espaço e tempo** (*tradeoff*).
 - Um algoritmo pode ser projetado para ter ganhos de velocidade ao custo de usar espaço extra de memória.



Medindo a Eficiência de Algoritmos

- Na escolha de algoritmos, nós frequentemente temos que optar entre **espaço e tempo** (*tradeoff*).
 - Um algoritmo pode ser projetado para ter ganhos de velocidade ao custo de usar espaço extra de memória.
- Memória tem, atualmente, um preço aceitável para computadores, mas isso ainda não é o mesmo para dispositivos men



Medindo a Eficiência de Algoritmos

- Na escolha de algoritmos, nós frequentemente temos que optar entre **espaço** e **tempo** (*tradeoff*).
 - Um algoritmo pode ser projetado para ter ganhos de velocidade ao custo de usar espaço extra de memória.
- Memória tem, atualmente, um preço aceitável para computadores, mas isso ainda não é o mesmo para dispositivos menores.
 - Exemplo: memória de celular.



Medindo o tempo de execução de um algoritmo

Medindo o tempo de execução de um algoritmo

- Uma maneira de medir o custo de tempo de um algoritmo é usar o clock do computador para obter o tempo real de execução.

Medindo o tempo de execução de um algoritmo

- Uma maneira de medir o custo de tempo de um algoritmo é usar o clock do computador para obter o tempo real de execução.
 - *Benchmarking* ou *profiling*.

Medindo o tempo de execução de um algoritmo

- Uma maneira de medir o custo de tempo de um algoritmo é usar o clock do computador para obter o tempo real de execução.
 - *Benchmarking* ou *profiling*.
- Nós podemos usar a função `time ()` do módulo `time` de Python.

Medindo o tempo de execução de um algoritmo

- Uma maneira de medir o custo de tempo de um algoritmo é usar o clock do computador para obter o tempo real de execução.
 - *Benchmarking* ou *profiling*.
- Nós podemos usar a função `time ()` do módulo `time` de Python.
 - Essa função retorna o número de segundos transcorridos entre a hora do computador e a data de 1º de janeiro de 1970.

Medindo o tempo de execução de um algoritmo (cont.)

```
1 import time
2
3 tamanho = 100000000
4 print ("%12s%16s" % ("Tamanho", "Segundos"))
5 for count in range(5):
6
7     start = time.time()
8     # início do algoritmo
9     work = 1
10    for x in range(tamanho):
11        work+=1
12        work-=1
13
14    # término do algoritmo
15    elapsed = time.time() - start
16
17    print ("%12d%16.3f" % (tamanho, elapsed))
18    tamanho *= 2
```



Medindo o tempo de execução de um algoritmo (cont.)

- Saída do programa anterior:

Tamanho	Segundos
100000000	1.252
200000000	2.511
400000000	4.928
800000000	9.814
1600000000	19.581

Medindo o tempo de execução de um algoritmo (cont.)

- Versão modificada do programa anterior:

```
1 import time
2
3 tamanho = 1000
4 print ("%12s%16s" % ("Tamanho", "Segundos"))
5 for count in range(5):
6
7     start = time.time()
8     # início do algoritmo
9     work = 1
10    for x in range(tamanho):
11        for y in range(tamanho):
12            work+=1
13            work-=1
14
15    # término do algoritmo
16    elapsed = time.time() - start
17
18    print ("%12d%16.3f" % (tamanho, elapsed))
19    tamanho *= 2
```



Medindo o tempo de execução de um algoritmo (cont.)

- Resultado:

Tamanho	Segundos
1000	0.131
2000	0.488
4000	1.945
8000	7.784
16000	31.129



Medindo o tempo de execução de um algoritmo (cont.)

- Medição de tempo no Linux

```
tiago@omsk:~/Dropbox/aulas/2019/aed1/codigos/chp11$ time python timing1.py
```

Problem Size	Seconds
10000000	0.915
20000000	1.826
40000000	3.657
80000000	7.302
160000000	14.623

```
real    0m28.333s
user    0m28.328s
sys     0m0.000s
```



Medindo o tempo de execução de um algoritmo (cont.)

Medindo o tempo de execução de um algoritmo (cont.)

- Este método realiza a medição do tempo de execução de muitos algoritmos.

Medindo o tempo de execução de um algoritmo (cont.)

- Este método realiza a medição do tempo de execução de muitos algoritmos.
- Problemas:

Medindo o tempo de execução de um algoritmo (cont.)

- Este método realiza a medição do tempo de execução de muitos algoritmos.
- Problemas:
 - Diferentes plataformas de hardware têm diferentes velocidades. Portanto, o tempo de execução de um algoritmo pode variar de acordo com a máquina.

Medindo o tempo de execução de um algoritmo (cont.)

- Este método realiza a medição do tempo de execução de muitos algoritmos.
- Problemas:
 - Diferentes plataformas de hardware têm diferentes velocidades. Portanto, o tempo de execução de um algoritmo pode variar de acordo com a máquina.
 - O tempo de execução varia também de acordo com o sistema operacional e a linguagem de programação.

Medindo o tempo de execução de um algoritmo (cont.)

- Este método realiza a medição do tempo de execução de muitos algoritmos.
- Problemas:
 - Diferentes plataformas de hardware têm diferentes velocidades. Portanto, o tempo de execução de um algoritmo pode variar de acordo com a máquina.
 - O tempo de execução varia também de acordo com o sistema operacional e a linguagem de programação.
 - É impraticável determinar o tempo de execução para alguns algoritmos com um conjunto de dados (*dataset*) muito grande.

Quantidade de instruções

Quantidade de instruções

- Outra técnica é contar o número de instruções executadas com diferentes tamanhos de dados.

Quantidade de instruções

- Outra técnica é contar o número de instruções executadas com diferentes tamanhos de dados.
 - Nós contamos as instruções em código de alto-nível em que o algoritmo é escrito. A contagem não é feita das instruções do programa na sua linguagem de máquina executável.

Quantidade de instruções

- Outra técnica é contar o número de instruções executadas com diferentes tamanhos de dados.
 - Nós contamos as instruções em código de alto-nível em que o algoritmo é escrito. A contagem não é feita das instruções do programa na sua linguagem de máquina executável.
- Distinção entre:

Quantidade de instruções

- Outra técnica é contar o número de instruções executadas com diferentes tamanhos de dados.
 - Nós contamos as instruções em código de alto-nível em que o algoritmo é escrito. A contagem não é feita das instruções do programa na sua linguagem de máquina executável.
- Distinção entre:
 - Instruções que executam o mesmo número de vezes independentemente do tamanho do problema.

Quantidade de instruções

- Outra técnica é contar o número de instruções executadas com diferentes tamanhos de dados.
 - Nós contamos as instruções em código de alto-nível em que o algoritmo é escrito. A contagem não é feita das instruções do programa na sua linguagem de máquina executável.
- Distinção entre:
 - Instruções que executam o mesmo número de vezes independentemente do tamanho do problema.
 - Instruções cuja quantidade varia de acordo com o tamanho do problema.

Quantidade de instruções (cont.)

```
1 tamanho = 1000
2 print ("%12s%15s" % ("Tamanho", "Iterações"))
3
4 for count in range(5):
5     quantidade = 0
6
7     #início do algoritmo
8     work = 1
9     for j in range(tamanho):
10        for k in range(tamanho):
11            quantidade+=1
12            work+=1
13            work-=1
14        #fim do algoritmo
15
16    print ("%12d%15d" % (tamanho, quantidade))
17    tamanho *= 2
```



Quantidade de instruções (cont.)

- Saída do programa anterior:

Tamanho	Iterações
1000	10000000
2000	40000000
4000	160000000
8000	640000000
16000	2560000000

```

1 class Counter(object):
2     def __init__(self):
3         self._number = 0
4
5     def increment(self):
6         self._number += 1
7
8     def __str__(self):
9         return str(self._number)
10
11 def fib (n, counter):
12     #conta o numero de chamadas para a função
13     counter.increment()
14     if n < 3:
15         return 1
16     else:
17         return (fib(n-1, counter) + fib(n-2, counter))
18
19 tamanho = 2
20 print ("%12s%15s" % ("Tamanho", "Chamadas"))
21 for count in range(5):
22     counter = Counter()
23
24     #início do algoritmo
25     fib (tamanho, counter)
26     # término do algoritmo
27
28     print ("%12s%15s" % (tamanho, counter))
29     tamanho *= 2

```



Quantidade de instruções (cont.)

- Saída do programa anterior:

Tamanho	Chamadas
2	1
4	5
8	41
16	1973
32	4356617

Medir a memória usada por um algoritmo

Medir a memória usada por um algoritmo

- Uma análise completa de recursos usados por um algoritmo inclui a quantidade de memória exigida para execução do algoritmo.

Medir a memória usada por um algoritmo

- Uma análise completa de recursos usados por um algoritmo inclui a quantidade de memória exigida para execução do algoritmo.
- Nós focamos nas taxas potenciais de crescimento.

Medir a memória usada por um algoritmo

- Uma análise completa de recursos usados por um algoritmo inclui a quantidade de memória exigida para execução do algoritmo.
- Nós focamos nas taxas potenciais de crescimento.
- Alguns algoritmos requerem a mesma quantidade de memória para resolver qualquer problema.

Medir a memória usada por um algoritmo

- Uma análise completa de recursos usados por um algoritmo inclui a quantidade de memória exigida para execução do algoritmo.
- Nós focamos nas taxas potenciais de crescimento.
- Alguns algoritmos requerem a mesma quantidade de memória para resolver qualquer problema.
- Outros algoritmos requerem mais memória conforme o crescimento do tamanho do problema.

Análise de complexidade

Análise de complexidade

- Análise de complexidade implica ler o algoritmo e usar lápis e papel para manipular algumas análises algébricas.

Análise de complexidade

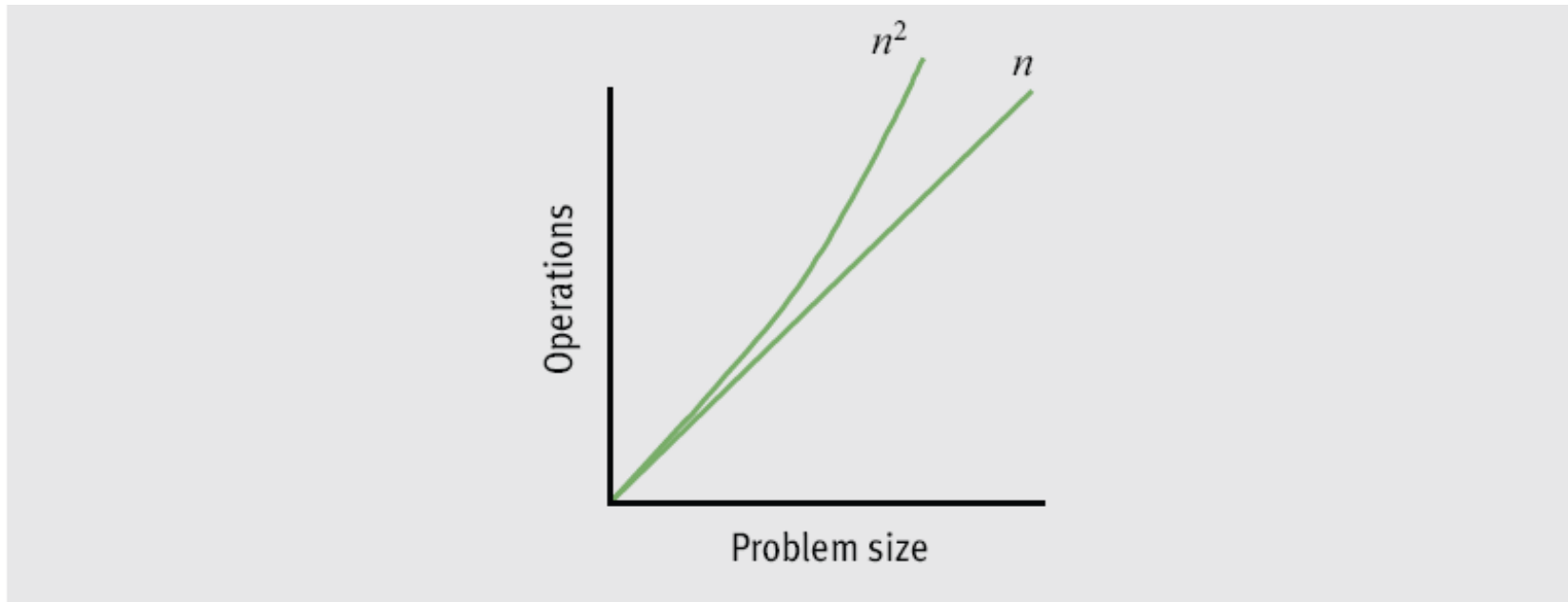
- Análise de complexidade implica ler o algoritmo e usar lápis e papel para manipular algumas análises algébricas.
- É usada para determinar a eficiência de algoritmos.

Análise de complexidade

- Análise de complexidade implica ler o algoritmo e usar lápis e papel para manipular algumas análises algébricas.
- É usada para determinar a eficiência de algoritmos.
- Ela nos permite medir a eficiência dos algoritmos independentemente da plataforma ou da quantidade de instruções.

Ordens de complexidade

- Considere os dois resultados abaixo:



[FIGURE 11.5] A graph of the amounts of work done in the tester programs

Ordens de complexidade (cont.)

PROBLEM SIZE	WORK OF THE FIRST ALGORITHM	WORK OF THE SECOND ALGORITHM
2	2	4
10	10	100
1000	1000	1,000,000

[TABLE 11.1] The amounts of work in the tester programs

Ordens de complexidade (cont.)

PROBLEM SIZE	WORK OF THE FIRST ALGORITHM	WORK OF THE SECOND ALGORITHM
2	2	4
10	10	100
1000	1000	1,000,000

[TABLE 11.1] The amounts of work in the tester programs

- As performances desses algoritmos variam pelo que costumamos chamar de **ordem de complexidade**:

Ordens de complexidade (cont.)

PROBLEM SIZE	WORK OF THE FIRST ALGORITHM	WORK OF THE SECOND ALGORITHM
2	2	4
10	10	100
1000	1000	1,000,000

[TABLE 11.1] The amounts of work in the tester programs

- As performances desses algoritmos variam pelo que costumamos chamar de **ordem de complexidade**:
 - O primeiro algoritmo é **linear**.

Ordens de complexidade (cont.)

PROBLEM SIZE	WORK OF THE FIRST ALGORITHM	WORK OF THE SECOND ALGORITHM
2	2	4
10	10	100
1000	1000	1,000,000

[TABLE 11.1] The amounts of work in the tester programs

- As performances desses algoritmos variam pelo que costumamos chamar de **ordem de complexidade**:
 - O primeiro algoritmo é **linear**.
 - O segundo algoritmo é **quadrático**.

Ordens de complexidade (cont.)

Ordens de complexidade (cont.)

- Ordens de complexidade

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Constante

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Constante
 - Requer o mesmo número de operações independentemente do tamanho do problema.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Constante
 - Requer o mesmo número de operações independentemente do tamanho do problema.
 - É o caso, por exemplo, do acesso direto a um elemento de uma matriz.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Constante
 - Requer o mesmo número de operações independentemente do tamanho do problema.
 - É o caso, por exemplo, do acesso direto a um elemento de uma matriz.
 - Representamos: $O(1)$

Ordens de complexidade (cont.)

Ordens de complexidade (cont.)

- Ordens de complexidade

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Logaritmo

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Logaritmo
 - O volume de operações cresce na proporção ao log n do tamanho do problema.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Logaritmo
 - O volume de operações cresce na proporção ao log n do tamanho do problema.
 - O crescimento do número de operações é menor do que o do número de itens.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Logaritmo
 - O volume de operações cresce na proporção ao log n do tamanho do problema.
 - O crescimento do número de operações é menor do que o do número de itens.
 - É o caso, por exemplo, de algoritmos de busca em árvores binárias ordenadas (*Binary Search Trees*).

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Logaritmo
 - O volume de operações cresce na proporção ao log n do tamanho do problema.
 - O crescimento do número de operações é menor do que o do número de itens.
 - É o caso, por exemplo, de algoritmos de busca em árvores binárias ordenadas (*Binary Search Trees*).
 - Representamos: $O(\log n)$

Ordens de complexidade (cont.)

Ordens de complexidade (cont.)

- Ordens de complexidade

Ordens de complexidade (cont.)

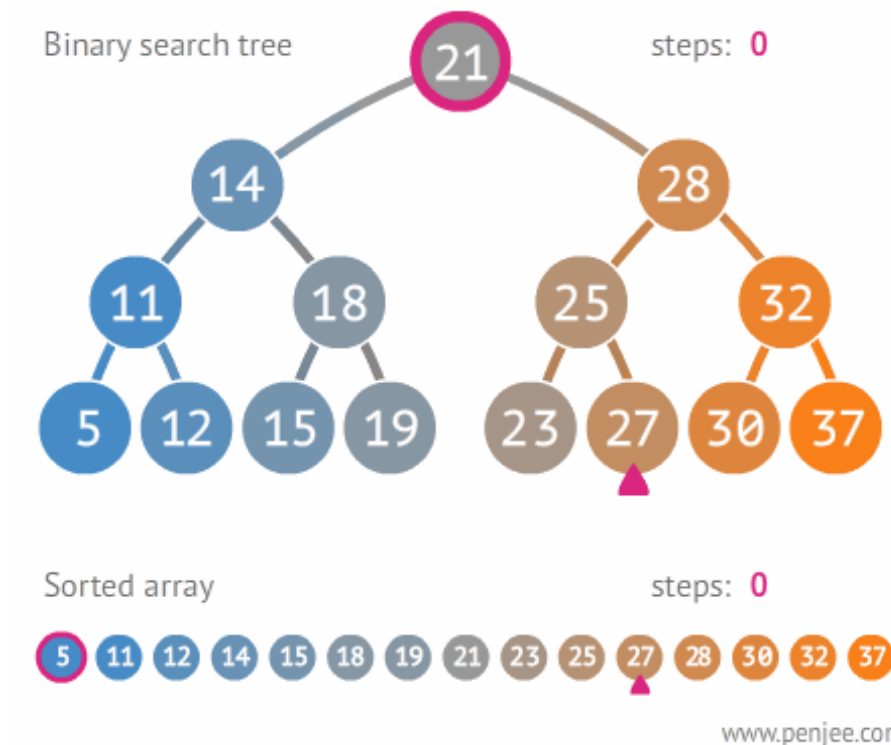
- Ordens de complexidade
 - Logaritmo

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Logaritmo
 - Árvores binárias ordenadas (*Binary Search Trees*).

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Logaritmo
 - Árvores binárias ordenadas (*Binary Search Trees*).



Ordens de complexidade (cont.)

Ordens de complexidade (cont.)

- Ordens de complexidade

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Linear

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Linear
 - O crescimento do número de operações cresce na proporção direta do crescimento do tamanho do problema.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Linear
 - O crescimento do número de operações cresce na proporção direta do crescimento do tamanho do problema.
 - É o caso de algoritmos de busca em um vetor não ordenado.

Ordens de complexidade (cont.)

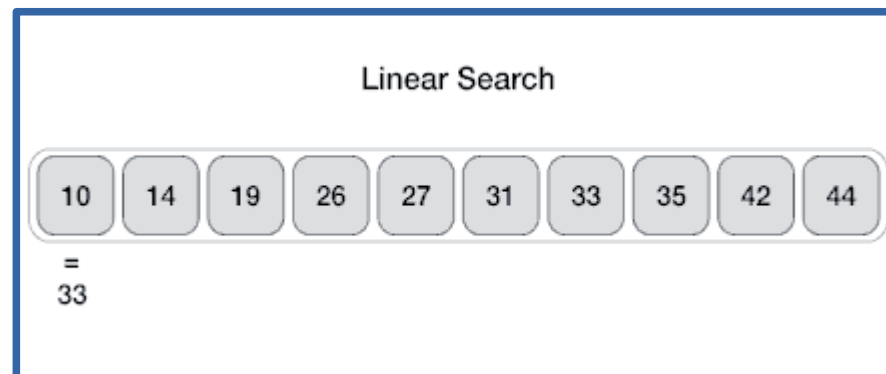
- Ordens de complexidade
 - Linear
 - O crescimento do número de operações cresce na proporção direta do crescimento do tamanho do problema.
 - É o caso de algoritmos de busca em um vetor não ordenado.
 - Representamos: $O(n)$

Ordens de complexidade (cont.)

- Ordens de complexidade

- Linear

- O crescimento do número de operações cresce na proporção direta do crescimento do tamanho do problema.
 - É o caso de algoritmos de busca em um vetor não ordenado.
 - Representamos: $O(n)$



Ordens de complexidade (cont.)

Ordens de complexidade (cont.)

- Ordens de complexidade

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Sub-quadrático (ou super-linear)

Ordens de complexidade (cont.)

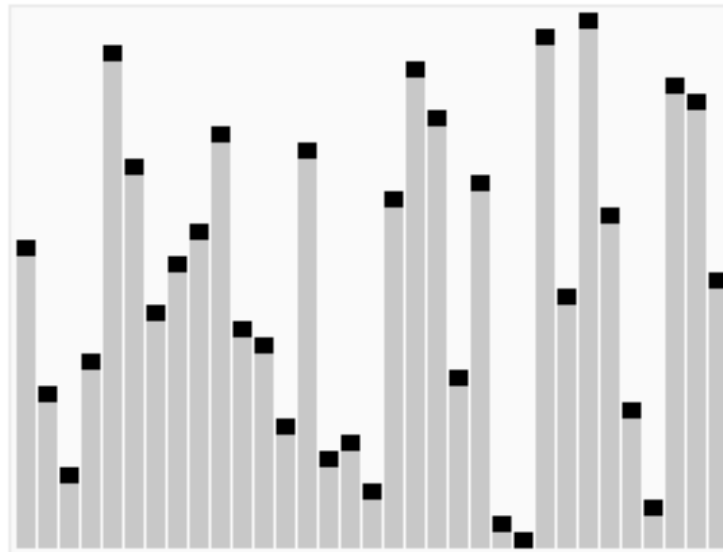
- Ordens de complexidade
 - Sub-quadrático (ou super-linear)
 - É o caso, por exemplo, do algoritmo de ordenação Quicksort. Ele tem essa complexidade no caso médio.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Sub-quadrático (ou super-linear)
 - É o caso, por exemplo, do algoritmo de ordenação Quicksort. Ele tem essa complexidade no caso médio.
 - Representamos: $O(n \log n)$

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Sub-quadrático (ou super-linear)
 - É o caso, por exemplo, do algoritmo de ordenação Quicksort. Ele tem essa complexidade no caso médio.
 - Representamos: $O(n \log n)$



Ordens de complexidade (cont.)

Ordens de complexidade (cont.)

- Ordens de complexidade

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Polinomial

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Polinomial
 - O tempo de execução do algoritmo cresce a uma taxa de n^k , onde k é uma constante maior do que 1. Exemplos: n^2 , n^3 etc.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Polinomial
 - O tempo de execução do algoritmo cresce a uma taxa de n^k , onde k é uma constante maior do que 1. Exemplos: n^2 , n^3 etc.
 - n^2 e n^3 são pertencem à mesma ordem de complexidade.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Polinomial
 - O tempo de execução do algoritmo cresce a uma taxa de n^k , onde k é uma constante maior do que 1. Exemplos: n^2 , n^3 etc.
 - n^2 e n^3 são pertencem à mesma ordem de complexidade.
 - São algoritmos factíveis, mas tendem a se tornar muito ruins quando a quantidade de dados é suficientemente grande.

Ordens de complexidade (cont.)

- Ordens de complexidade

- Polinomial

- O tempo de execução do algoritmo cresce a uma taxa de n^k , onde k é uma constante maior do que 1. Exemplos: n^2 , n^3 etc.
 - n^2 e n^3 são pertencem à mesma ordem de complexidade.
 - São algoritmos factíveis, mas tendem a se tornar muito ruins quando a quantidade de dados é suficientemente grande.
 - É o caso de algoritmos que têm dois laços encadeados. Exemplo: processamento de uma matriz bidimensional.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Polinomial
 - O tempo de execução do algoritmo cresce a uma taxa de n^k , onde k é uma constante maior do que 1. Exemplos: n^2 , n^3 etc.
 - n^2 e n^3 são pertencem à mesma ordem de complexidade.
 - São algoritmos factíveis, mas tendem a se tornar muito ruins quando a quantidade de dados é suficientemente grande.
 - É o caso de algoritmos que têm dois laços encadeados. Exemplo: processamento de uma matriz bidimensional.
 - Representamos: $O(n^k)$

Ordens de complexidade (cont.)

Ordens de complexidade (cont.)

- Ordens de complexidade

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Exponencial

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Exponencial
 - É o tipo de algoritmo que não consegue executar problemas grandes.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Exponencial
 - É o tipo de algoritmo que não consegue executar problemas grandes.
 - É o caso de algoritmos que fazem busca em árvores binárias não ordenadas.

Ordens de complexidade (cont.)

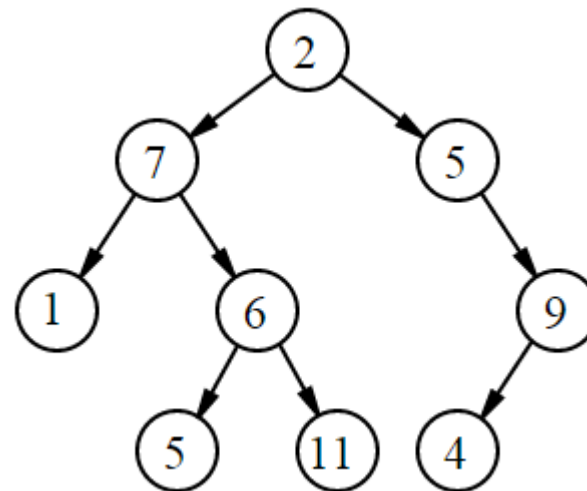
- Ordens de complexidade
 - Exponencial
 - É o tipo de algoritmo que não consegue executar problemas grandes.
 - É o caso de algoritmos que fazem busca em árvores binárias não ordenadas.
 - Representamos: $O(2^n)$

Ordens de complexidade (cont.)

- Ordens de complexidade

- Exponencial

- É o tipo de algoritmo que não consegue executar problemas grandes.
 - É o caso de algoritmos que fazem busca em árvores binárias não ordenadas.
 - Representamos: $O(2^n)$



Ordens de complexidade (cont.)

Ordens de complexidade (cont.)

- Ordens de complexidade

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Fatorial

Ordens de complexidade (cont.)

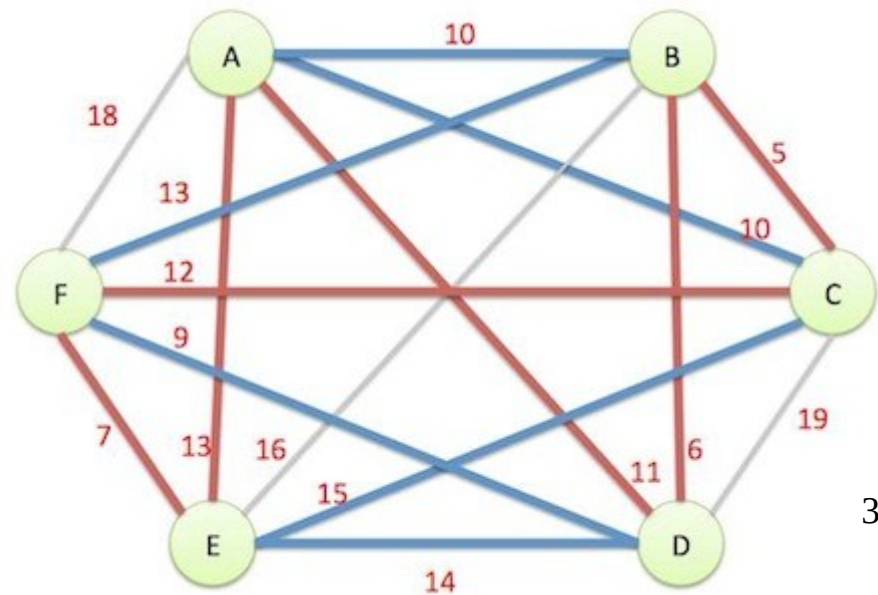
- Ordens de complexidade
 - Fatorial
 - O número de instruções executadas cresce muito rapidamente para um pequeno número de itens processados.

Ordens de complexidade (cont.)

- Ordens de complexidade
 - Fatorial
 - O número de instruções executadas cresce muito rapidamente para um pequeno número de itens processados.
 - Exemplo: problema do caixeiro viajante.

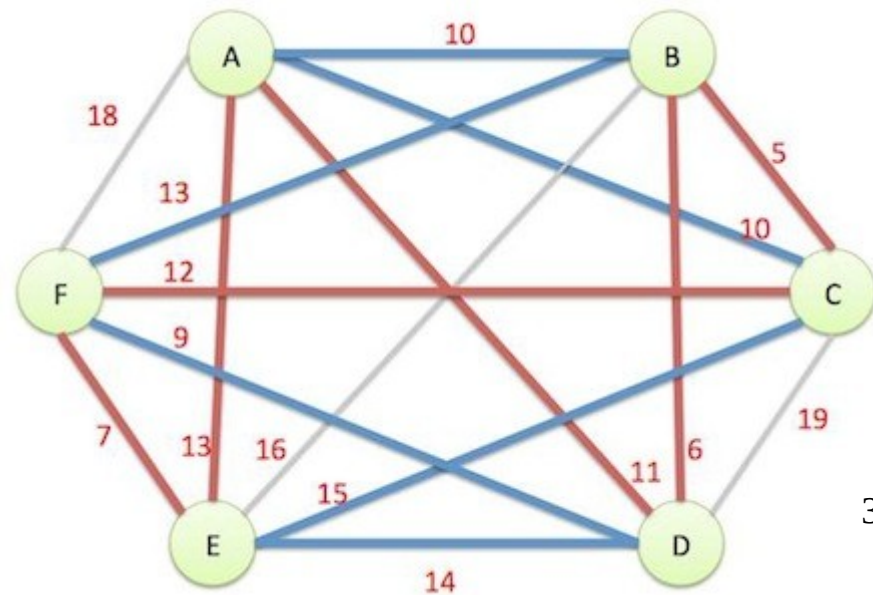
Ordens de complexidade (cont.)

- Ordens de complexidade
 - Fatorial
 - O número de instruções executadas cresce muito rapidamente para um pequeno número de itens processados.
 - Exemplo: problema do caixeiro viajante.



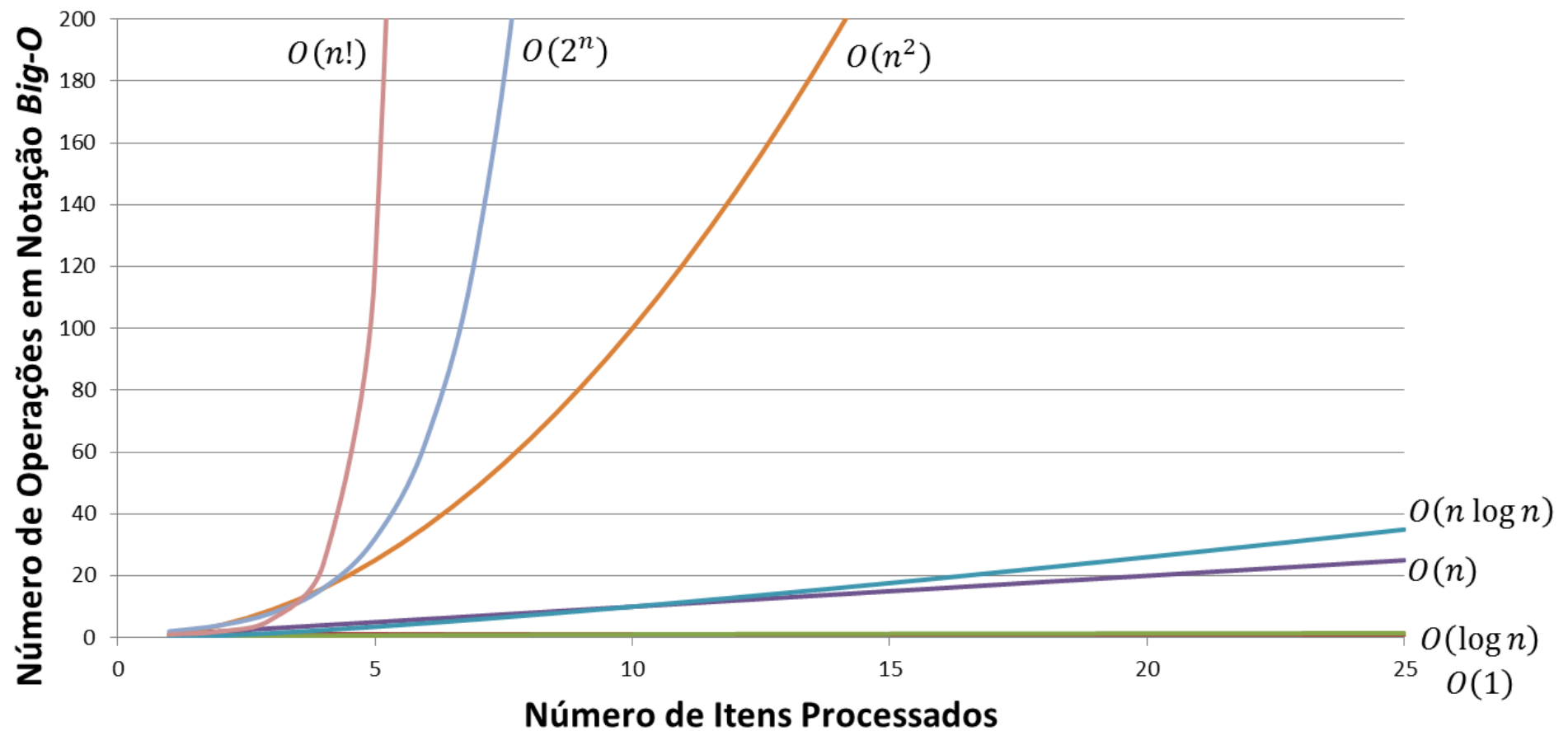
Ordens de complexidade (cont.)

- Ordens de complexidade
 - Fatorial
 - O número de instruções executadas cresce muito rapidamente para um pequeno número de itens processados.
 - Exemplo: problema do caixeiro viajante.
 - Representamos: $O(n!)$

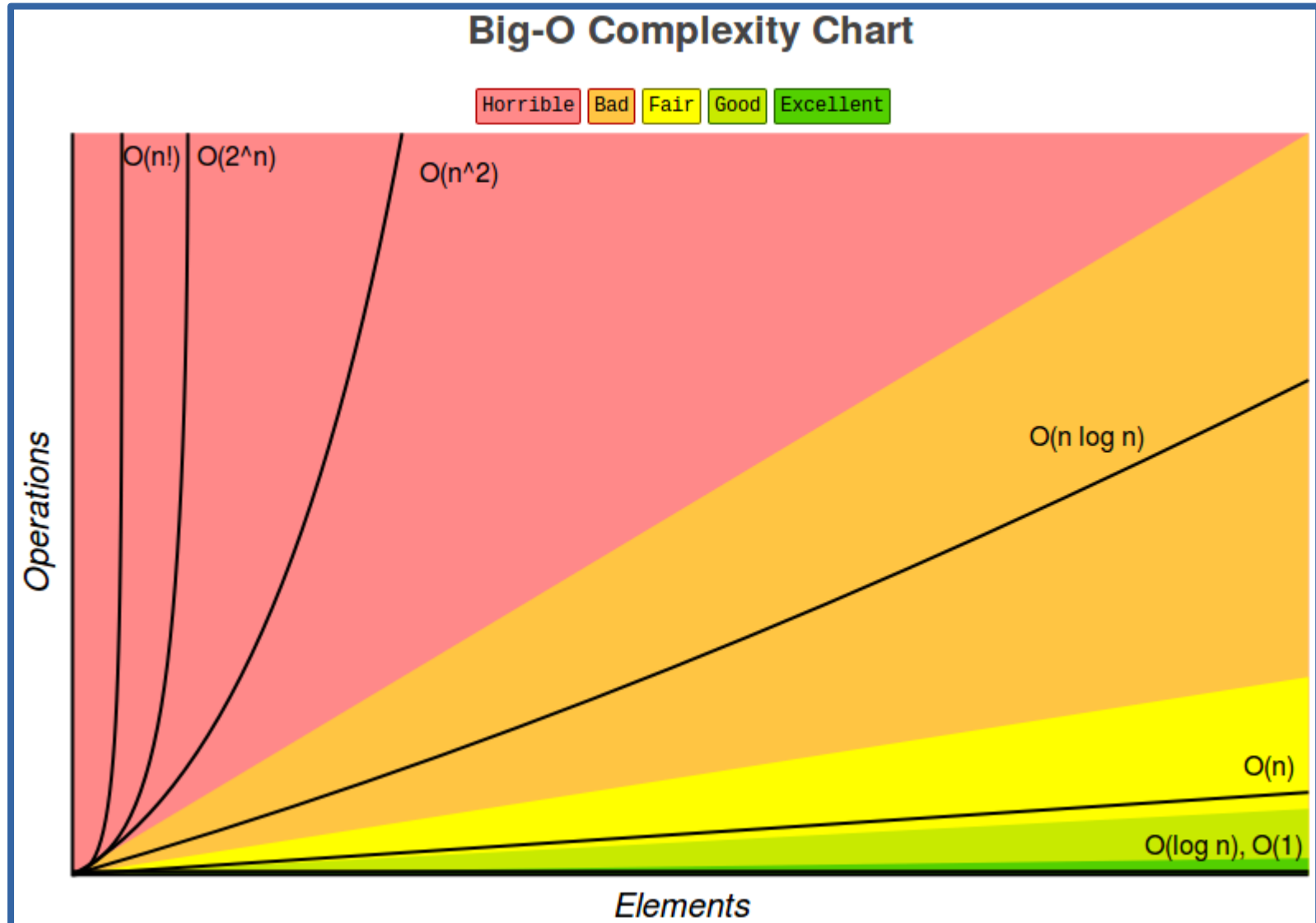


Ordens de complexidade (cont.)

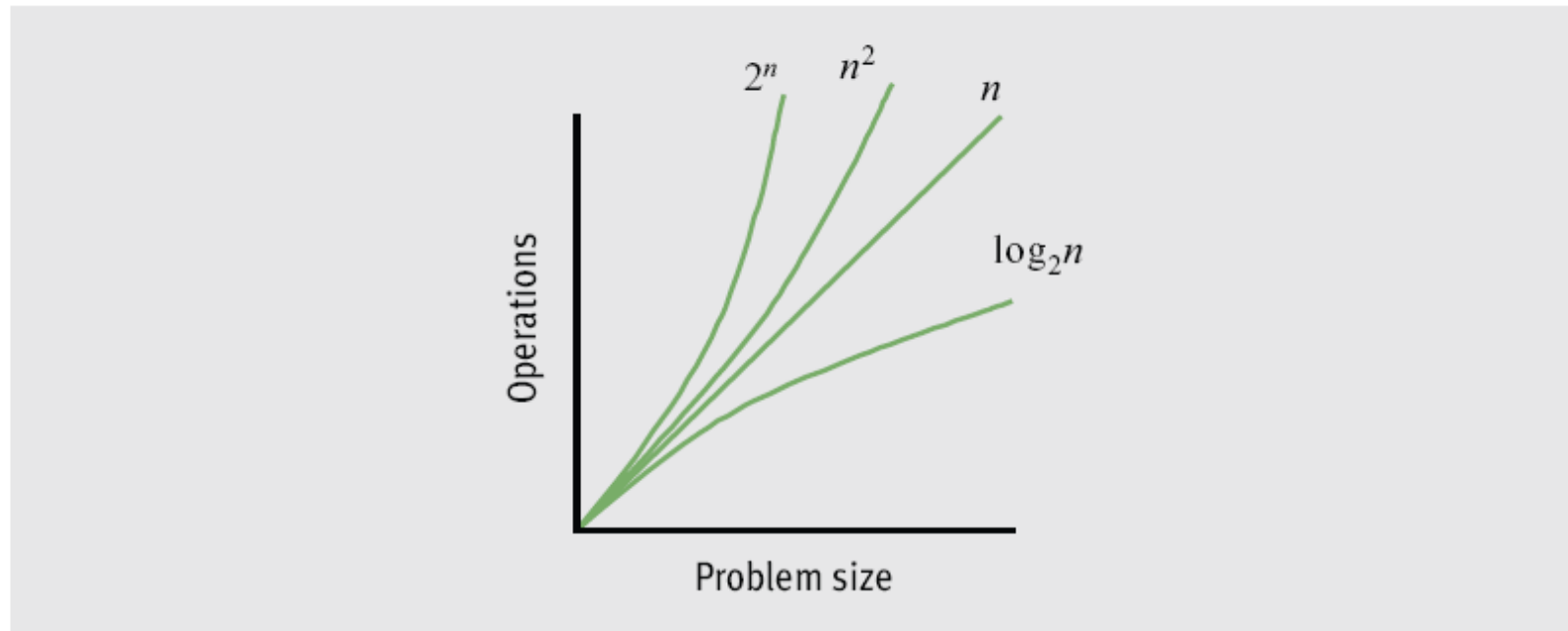
Ilustração das Complexidades Mais Comuns - Notação *Big-O*



Ordens de complexidade (cont.)



Ordens de complexidade (cont.)



[FIGURE 11.6] A graph of some sample orders of complexity

n	LOGARITHMIC ($\log_2 n$)	LINEAR (n)	QUADRATIC (n^2)	EXPONENTIAL (2^n)
100	7	100	10,000	Off the charts
1000	10	1000	1,000,000	Off the charts
1,000,000	20	1,000,000	1,000,000,000,000	Really off the charts

[TABLE 11.2] Some sample orders of complexity

Notação Big-O

Notação Big-O

- A quantidade de trabalho (“*work*”, “*job*”) em um algoritmo tipicamente é a soma de vários termos em um polinômio.

Notação Big-O

- A quantidade de trabalho (“*work*”, “*job*”) em um algoritmo tipicamente é a soma de vários termos em um polinômio.
 - Nós focamos o termo chamado de **dominante**.

Notação Big-O

- A quantidade de trabalho (“*work*”, “*job*”) em um algoritmo tipicamente é a soma de vários termos em um polinômio.
 - Nós focamos o termo chamado de **dominante**.
- Como n se torna grande, o termo dominante se torna tão grande que a quantidade de trabalho representada pelos demais termos pode ser ignorada.

Notação Big-O

- A quantidade de trabalho (“*work*”, “*job*”) em um algoritmo tipicamente é a soma de vários termos em um polinômio.
 - Nós focamos o termo chamado de **dominante**.
- Como n se torna grande, o termo dominante se torna tão grande que a quantidade de trabalho representada pelos demais termos pode ser ignorada.
 - Chamamos isso de **análise assintótica**.

Notação Big-O

- A quantidade de trabalho (“*work*”, “*job*”) em um algoritmo tipicamente é a soma de vários termos em um polinômio.
 - Nós focamos o termo chamado de **dominante**.
- Como n se torna grande, o termo dominante se torna tão grande que a quantidade de trabalho representada pelos demais termos pode ser ignorada.
 - Chamamos isso de **análise assintótica**.
- **Notação Big-O**: utilizada para expressar a eficiência ou complexidade computacional de um algoritmo.

O papel da constante de proporcionalidade

O papel da constante de proporcionalidade

- A **constante de proporcionalidade** envolve os termos e coeficientes que são geralmente ignorados durante a análise de complexidade (Big-O).

O papel da constante de proporcionalidade

- A **constante de proporcionalidade** envolve os termos e coeficientes que são geralmente ignorados durante a análise de complexidade (Big-O).
 - Porém, quando esses itens são grandes, eles podem ter um impacto no algoritmo, especialmente quando lidamos com conjunto de dados de tamanho pequeno ou médio.

Algoritmos de Busca

Algoritmos de Busca

- Nós agora apresentamos alguns algoritmos que podem ser usados para busca e ordenação de listas (ou vetores).

Algoritmos de Busca

- Nós agora apresentamos alguns algoritmos que podem ser usados para busca e ordenação de listas (ou vetores).
 - Nós iniciaremos com a discussão sobre projeto de um algoritmo.

Algoritmos de Busca

- Nós agora apresentamos alguns algoritmos que podem ser usados para busca e ordenação de listas (ou vetores).
 - Nós iniciaremos com a discussão sobre projeto de um algoritmo.
 - Então nós mostraremos a sua implementação como uma função de Python.

Algoritmos de Busca

- Nós agora apresentamos alguns algoritmos que podem ser usados para busca e ordenação de listas (ou vetores).
 - Nós iniciaremos com a discussão sobre projeto de um algoritmo.
 - Então nós mostraremos a sua implementação como uma função de Python.
 - Finalmente, nós faremos uma análise da complexidade computacional dos algoritmos.

Algoritmos de Busca

- Nós agora apresentamos alguns algoritmos que podem ser usados para busca e ordenação de listas (ou vetores).
 - Nós iniciaremos com a discussão sobre projeto de um algoritmo.
 - Então nós mostraremos a sua implementação como uma função de Python.
 - Finalmente, nós faremos uma análise da complexidade computacional dos algoritmos.
- Por questão de organização da aula, cada função processará uma lista de inteiros.

Busca pelo Mínimo

Busca pelo Mínimo

- A função `min()` de Python retorna a posição do menor item de uma lista.



Busca pelo Mínimo

- A função `min()` de Python retorna a posição do menor item de uma lista.

```
1 def buscaMin(lista):
2     minpos = 0
3     atual = 1
4     while atual < len(lista):
5         if lista[atual] < lista[minpos]:
6             minpos = atual
7         atual += 1
8     return minpos
9
10 L = [10, 8, 2, 1, 7, 6]
11 min = buscaMin(L)
12 print (min)
```



Busca pelo Mínimo

- A função `min()` de Python retorna a posição do menor item de uma lista.

```
1 def buscaMin(lista):
2     minpos = 0
3     atual = 1
4     while atual < len(lista):
5         if lista[atual] < lista[minpos]:
6             minpos = atual
7         atual += 1
8     return minpos
9
10 L = [10, 8, 2, 1, 7, 6]
11 min = buscaMin(L)
12 print (min)
```



Busca pelo Mínimo

- A função `min()` de Python retorna a posição do menor item de uma lista.

```
1 def buscaMin(lista):
2     minpos = 0
3     atual = 1
4     while atual < len(lista):
5         if lista[atual] < lista[minpos]:
6             minpos = atual
7         atual += 1
8     return minpos
9
10 L = [10, 8, 2, 1, 7, 6]
11 min = buscaMin(L)
12 print (min)
```

- $n - 1$ comparações para uma lista de tamanho n .



Busca pelo Mínimo

- A função `min()` de Python retorna a posição do menor item de uma lista.

```
1 def buscaMin(lista):
2     minpos = 0
3     atual = 1
4     while atual < len(lista):
5         if lista[atual] < lista[minpos]:
6             minpos = atual
7         atual += 1
8     return minpos
9
10 L = [10, 8, 2, 1, 7, 6]
11 min = buscaMin(L)
12 print (min)
```

- $n - 1$ comparações para uma lista de tamanho n
- $O(n)$

Melhor caso, pior caso e caso
médio

Melhor caso, pior caso e caso médio

- A busca sequencial (linear) considera três casos:

Melhor caso, pior caso e caso médio

- A busca sequencial (linear) considera três casos:
 - **Pior caso.**

Melhor caso, pior caso e caso médio

- A busca sequencial (linear) considera três casos:
 - **Pior caso.**
 - O item está no final da lista ou não está na lista.

Melhor caso, pior caso e caso médio

- A busca sequencial (linear) considera três casos:
 - **Pior caso.**
 - O item está no final da lista ou não está na lista.
 - $O(n)$

Melhor caso, pior caso e caso médio

- A busca sequencial (linear) considera três casos:
 - **Pior caso.**
 - O item está no final da lista ou não está na lista.
 - $O(n)$
 - **Melhor caso.**

Melhor caso, pior caso e caso médio

- A busca sequencial (linear) considera três casos:
 - **Pior caso.**
 - O item está no final da lista ou não está na lista.
 - $O(n)$
 - **Melhor caso.**
 - O item é o primeiro da lista.

Melhor caso, pior caso e caso médio

- A busca sequencial (linear) considera três casos:
 - **Pior caso.**
 - O item está no final da lista ou não está na lista.
 - $O(n)$
 - **Melhor caso.**
 - O item é o primeiro da lista.
 - $O(1)$

Melhor caso, pior caso e caso médio

- A busca sequencial (linear) considera três casos:
 - **Pior caso.**
 - O item está no final da lista ou não está na lista.
 - $O(n)$
 - **Melhor caso.**
 - O item é o primeiro da lista.
 - $O(1)$
 - **Caso médio.**

Melhor caso, pior caso e caso médio

- A busca sequencial (linear) considera três casos:
 - **Pior caso.**
 - O item está no final da lista ou não está na lista.
 - $O(n)$
 - **Melhor caso.**
 - O item é o primeiro da lista.
 - $O(1)$
 - **Caso médio.**
 - $O(n)$

Busca binária em uma lista

Busca binária em uma lista

- Uma busca linear (sequencial) é necessária se os dados não estão ordenados.

Busca binária em uma lista

- Uma busca linear (sequencial) é necessária se os dados não estão ordenados.
- Quando os dados já estão ordenados, podemos usar a busca binária.

Busca binária em uma lista

- Uma busca linear (sequencial) é necessária se os dados não estão ordenados.
- Quando os dados já estão ordenados, podemos usar a busca binária.

```
def binarySearch(target, lyst):  
    left = 0  
    right = len(lyst) - 1  
    while left <= right:  
        midpoint = (left + right) / 2  
        if target == lyst[midpoint]:  
            return midpoint  
        elif target < lyst[midpoint]:  
            right = midpoint - 1  
        else:  
            left = midpoint + 1  
    return -1
```

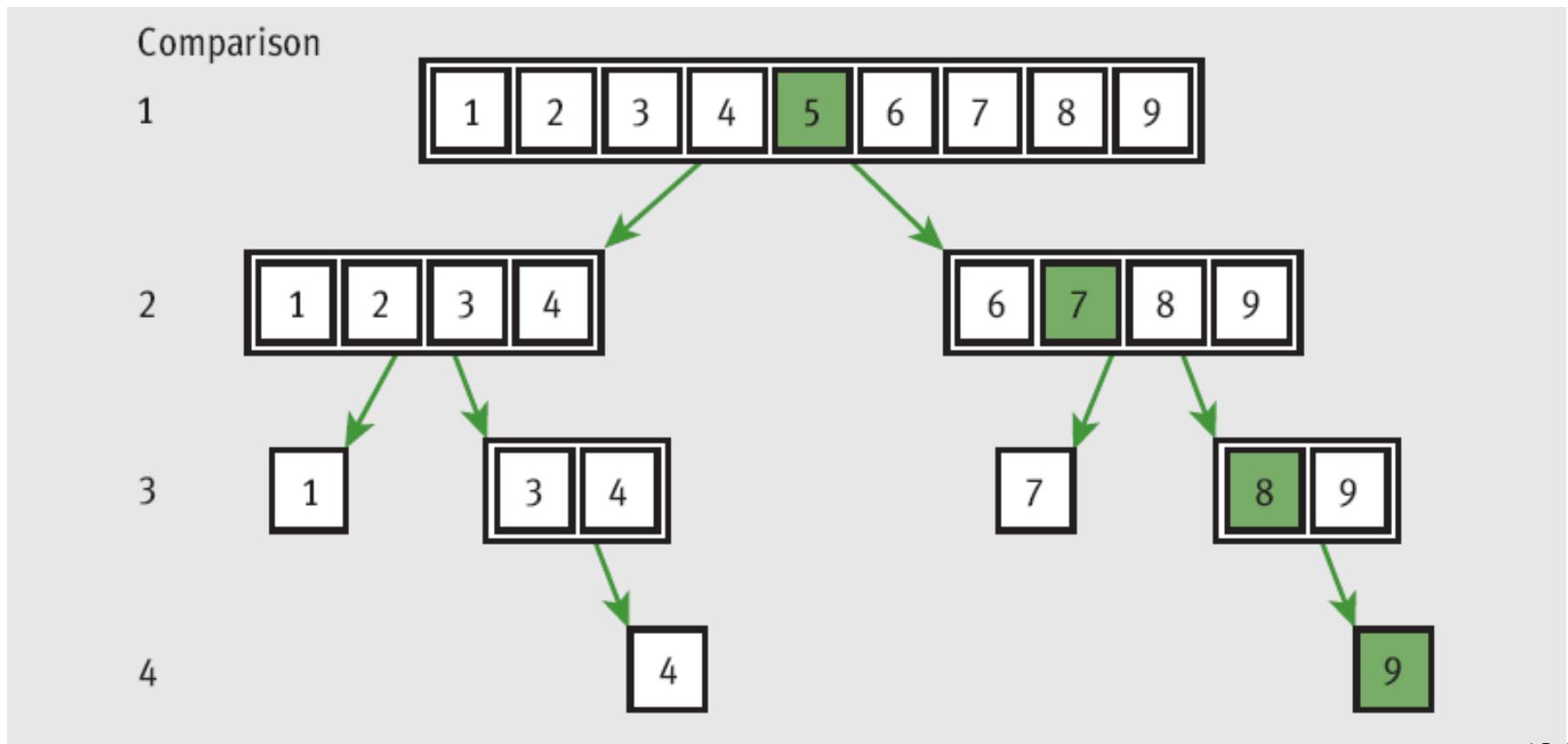
Busca binária em uma lista (cont.)

Busca binária em uma lista (cont.)

- Mais eficiente do que a busca linear.
 - Custo adicional para manter a lista ordenada.

Busca binária em uma lista (cont.)

- Mais eficiente do que a busca linear.
 - Custo adicional para manter a lista ordenada.



[FIGURE 11.7] The items of a list visited during a binary search for 10

Algoritmos de Ordenação

Algoritmos de Ordenação

- A função de ordenação que desenvolvemos aqui opera sobre uma lista de inteiros e utiliza uma função swap para trocar as posições de dois elementos da lista.

Algoritmos de Ordenação

- A função de ordenação que desenvolvemos aqui opera sobre uma lista de inteiros e utiliza uma função swap para trocar as posições de dois elementos da lista.

```
def swap(lyst, i, j):  
    """Exchanges the items at positions i and j."""  
    # You could say lyst[i], lyst[j] = lyst[j], lyst[i]  
    # but the following code shows what is really going on  
    temp = lyst[i]  
    lyst[i] = lyst[j]  
    lyst[j] = temp
```

Algoritmo de Seleção (ordenação)

Algoritmo de Seleção (ordenação)

- Talvez a mais simples estratégia é pesquisar na lista pela posição do menor item.
 - Se a posição não for igual à primeira posição, o algoritmo troca (*swap*) os itens nessas posições.

Algoritmo de Seleção (ordenação)

- Talvez a mais simples estratégia é pesquisar na lista pela posição do menor item.
 - Se a posição não for igual à primeira posição, o algoritmo troca (*swap*) os itens nessas posições.

UNSORTED LIST	AFTER 1st PASS	AFTER 2nd PASS	AFTER 3rd PASS	AFTER 4th PASS
5	1*	1	1	1
3	3	2*	2	2
1	5*	5	3*	3
2	2	3*	5*	4*
4	4	4	4	5*

[TABLE 11.3] A trace of the data during a selection sort

Algoritmo de Seleção (cont.)

Algoritmo de Seleção (cont.)

- O algoritmo de seleção é $O(n^2)$ em todos os casos.

Algoritmo de Seleção (cont.)

- O algoritmo de seleção é $O(n^2)$ em todos os casos.
- Para grandes conjuntos de dados, o custo de trocar os elementos pode também ser significativo.

Algoritmo de Seleção (cont.)

- O algoritmo de seleção é $O(n^2)$ em todos os casos.
- Para grandes conjuntos de dados, o custo de trocar os elementos pode também ser significativo.

```
def selectionSort(lyst):
    i = 0
    while i < len(lyst) - 1:           # Do n - 1 searches
        minIndex = i                  # for the smallest
        j = i + 1
        while j < len(lyst):          # Start a search
            if lyst[j] < lyst[minIndex]:
                minIndex = j
            j += 1
        if minIndex != i:              # Exchange if needed
            swap(lyst, minIndex, i)
        i += 1
```

Algoritmo da Bolha

Algoritmo da Bolha

- Começa no início da lista e compara pares de itens conforme percorre a lista.

Algoritmo da Bolha

- Começa no início da lista e compara pares de itens conforme percorre a lista.
 - Quando pares de itens estão fora de ordem, ele faz a troca dos itens.

Algoritmo da Bolha

- Começa no início da lista e compara pares de itens conforme percorre a lista.
 - Quando pares de itens estão fora de ordem, ele faz a troca dos itens.

UNSORTED LIST	AFTER 1st PASS	AFTER 2nd PASS	AFTER 3rd PASS	AFTER 4th PASS
5	4*	4	4	4
4	5*	2*	2	2
2	2	5*	1*	1
1	1	1	5*	3*
3	3	3	3	5*

[TABLE 11.4] A trace of the data during a bubble sort

Algoritmo da Bolha (cont.)

Algoritmo da Bolha (cont.)

- Algoritmo como uma função:

Algoritmo da Bolha (cont.)

- Algoritmo como uma função:

```
def bubbleSort(lyst):
    n = len(lyst)
    while n > 1:
        i = 1
        while i < n:
            if lyst[i] < lyst[i - 1]:
                swap(lyst, i, i - 1)
            i += 1
        n -= 1
```

Do n - 1 bubbles
Start each bubble
Exchange if needed

Algoritmo da Bolha (cont.)

Algoritmo da Bolha (cont.)

- Algoritmo da Bolha é $O(n^2)$.

Algoritmo da Bolha (cont.)

- Algoritmo da Bolha é $O(n^2)$.
- Ele não fará trocas se a lista já estiver ordenada.

Algoritmo da Bolha (cont.)

- Algoritmo da Bolha é $O(n^2)$.
- Ele não fará trocas se a lista já estiver ordenada.
- O comportamento para o pior caso para as trocas (swap) é muito maior que linear.

Algoritmo da Bolha (cont.)

- Algoritmo da Bolha é $O(n^2)$.
- Ele não fará trocas se a lista já estiver ordenada.
- O comportamento para o pior caso para as trocas (swap) é muito maior que linear.
- Pode ser melhorado, mas o caso médio ainda será $O(n^2)$.

Algoritmo de Inserção

```
def insertionSort(lyst):
    i = 1
    while i < len(lyst):
        itemToInsert = lyst[i]
        j = i - 1
        while j >= 0:
            if itemToInsert < lyst[j]:
                lyst[j + 1] = lyst[j]
                j -= 1
            else:
                break
        lyst[j + 1] = itemToInsert
        i += 1
```

- O pior caso do algoritmo é $O(n^2)$.

Algoritmo de Inserção (cont.)

UNSORTED LIST	AFTER 1st PASS	AFTER 2nd PASS	AFTER 3rd PASS	AFTER 4th PASS
2	2	1*	1	1
5 ←	5 (no insertion)	2	2	2
1	1←	5	4*	3*
4	4	4 ←	5	4
3	3	3	3 ←	5

[TABLE 11.5] A trace of the data during an insertion sort

Algoritmo de Inserção (cont.)

Algoritmo de Inserção (cont.)

- Quanto mais ordenada estiver a lista de entrada, mais eficiente será o algoritmo.

Algoritmo de Inserção (cont.)

- Quanto mais ordenada estiver a lista de entrada, mais eficiente será o algoritmo.
- No melhor caso, quando a lista está ordenada, o algoritmo de ordenação tem complexidade linear.

Algoritmo de Inserção (cont.)

- Quanto mais ordenada estiver a lista de entrada, mais eficiente será o algoritmo.
- No melhor caso, quando a lista está ordenada, o algoritmo de ordenação tem complexidade linear.
- No caso médio, o algoritmo de inserção ainda é quadrático.

Melhor caso, pior caso e caso médio

Melhor caso, pior caso e caso médio

- Através da análise de complexidade de um algoritmo, o seu comportamento pode ser analisado como:

Melhor caso, pior caso e caso médio

- Através da análise de complexidade de um algoritmo, o seu comportamento pode ser analisado como:
 - Melhor caso.

Melhor caso, pior caso e caso médio

- Através da análise de complexidade de um algoritmo, o seu comportamento pode ser analisado como:
 - Melhor caso.
 - Pior caso.

Melhor caso, pior caso e caso médio

- Através da análise de complexidade de um algoritmo, o seu comportamento pode ser analisado como:
 - Melhor caso.
 - Pior caso.
 - Caso médio.

Melhor caso, pior caso e caso médio

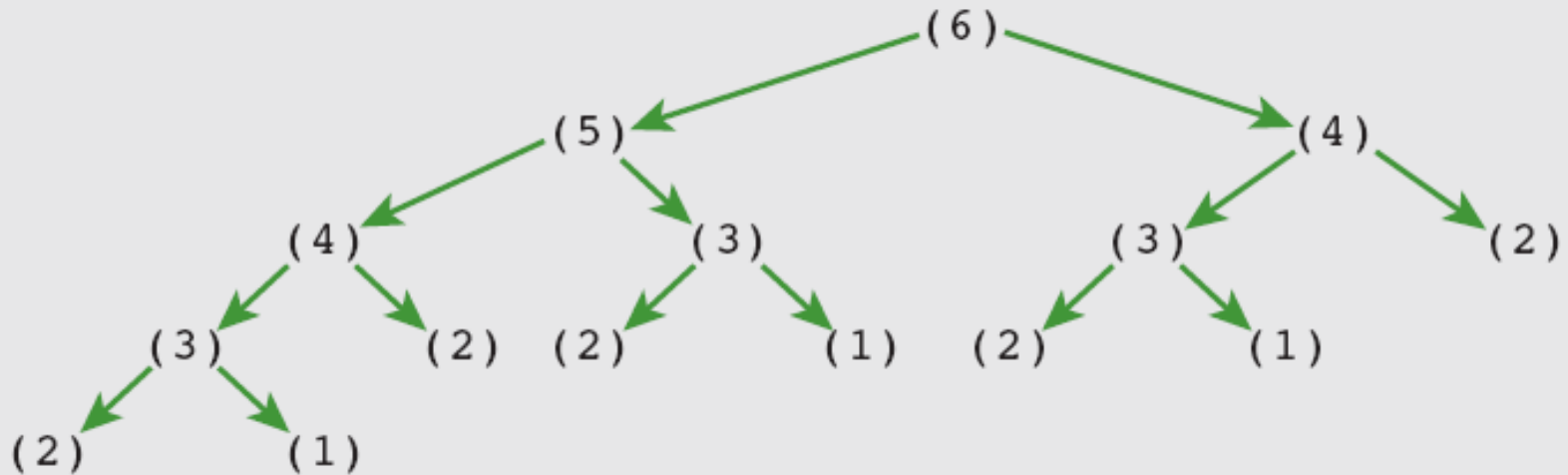
- Através da análise de complexidade de um algoritmo, o seu comportamento pode ser analisado como:
 - Melhor caso.
 - Pior caso.
 - Caso médio.
- Existem algoritmos cuja performance do melhor caso e do caso médio sejam similares, mas a eficiência pode cair significativamente no pior caso.

Melhor caso, pior caso e caso médio

- Através da análise de complexidade de um algoritmo, o seu comportamento pode ser analisado como:
 - Melhor caso.
 - Pior caso.
 - Caso médio.
- Existem algoritmos cuja performance do melhor caso e do caso médio sejam similares, mas a eficiência pode cair significativamente no pior caso.
- Quando nós desenvolvemos algoritmos, é importante estarmos cientes dessas distinções.

Fibonacci Recursivo: um algoritmo exponencial

```
def fib(n):  
    """The recursive Fibonacci function."""  
    if n < 3:  
        return 1  
    else:  
        return fib(n - 1) + fib(n - 2)
```



[FIGURE 11.8] A call tree for `fib(6)`

Fibonacci Recursivo: um algoritmo exponencial (cont.)

Fibonacci Recursivo: um algoritmo exponencial (cont.)

- Algoritmos exponenciais são geralmente impraticáveis de serem usados, mesmo com pequeno conjunto de dados de entrada.

Fibonacci Recursivo: um algoritmo exponencial (cont.)

- Algoritmos exponenciais são geralmente impraticáveis de serem usados, mesmo com pequeno conjunto de dados de entrada.
- Funções recursivas que são chamadas repetidamente com os mesmos argumentos podem ser mais eficientemente utilizadas através de uma técnica chamada de ***memoization***.

Fibonacci Recursivo: um algoritmo exponencial (cont.)

Fibonacci Recursivo: um algoritmo exponencial (cont.)

- Técnica de *memoization*:

Fibonacci Recursivo: um algoritmo exponencial (cont.)

- Técnica de *memoization*:
 - O programa mantém uma tabela de valores para cada argumento usado com a função.

Fibonacci Recursivo: um algoritmo exponencial (cont.)

- Técnica de *memoization*:
 - O programa mantém uma tabela de valores para cada argumento usado com a função.
 - Antes da função recursivamente processar um valor para um dado argumento, ele checa a tabela para ver se o argumento já tem um valor.

Convertendo o Fibonnaci para um algoritmo linear

Pseudocode:

- Set sum to 1

- Set first to 1

- Set second to 1

- Set count to 3

- While count \leq N

 - Set sum to first + second

 - Set first to second

 - Set second to sum

 - Increment count

Convertendo o Fibonacci para um algoritmo linear (cont.)

```
def fib(n, counter):  
    """Count the number of iterations in the Fibonacci  
    function."""  
    sum = 1  
    first = 1  
    second = 1  
    count = 3  
    while count <= n:  
        counter.increment()  
        sum = first + second  
        first = second  
        second = sum  
        count += 1  
    return sum
```

Problem Size	Iterations
2	0
4	2
8	6
16	14
32	30

Notação Omega (Ω)

- A notação Omega (Ω) descreve o limite inferior dos algoritmos.
- Essa notação é empregada para medir o melhor caso do algoritmo.

Notação Theta (Θ)

- É frequentemente o caso onde os limites superior e inferior de uma dada função são iguais.

Resumo

Resumo

- Diferentes algoritmos podem ser ranqueados de acordo com os recursos de tempo e memória que eles exigem para rodar.

Resumo

- Diferentes algoritmos podem ser ranqueados de acordo com os recursos de tempo e memória que eles exigem para rodar.
- Tempo de execução de um algoritmo pode ser medido empiricamente utilizando o *clock* do computador.

Resumo

- Diferentes algoritmos podem ser ranqueados de acordo com os recursos de tempo e memória que eles exigem para rodar.
- Tempo de execução de um algoritmo pode ser medido empiricamente utilizando o *clock* do computador.
- A contagem das instruções de um algoritmo fornece uma medida da quantidade de trabalho que o algoritmo faz.

Resumo

- Diferentes algoritmos podem ser ranqueados de acordo com os recursos de tempo e memória que eles exigem para rodar.
- Tempo de execução de um algoritmo pode ser medido empiricamente utilizando o *clock* do computador.
- A contagem das instruções de um algoritmo fornece uma medida da quantidade de trabalho que o algoritmo faz.
- A taxa de crescimento de trabalho de um algoritmo pode ser representada como uma função do tamanho das instâncias do problema (tamanho da entrada).

Resumo

- Diferentes algoritmos podem ser ranqueados de acordo com os recursos de tempo e memória que eles exigem para rodar.
- Tempo de execução de um algoritmo pode ser medido empiricamente utilizando o *clock* do computador.
- A contagem das instruções de um algoritmo fornece uma medida da quantidade de trabalho que o algoritmo faz.
- A taxa de crescimento de trabalho de um algoritmo pode ser representada como uma função do tamanho das instâncias do problema (tamanho da entrada).
- A notação Big-O é uma forma comum de expressar um comportamento do tempo de execução de um algoritmo.

Resumo (cont.)

Resumo (cont.)

- Classes comuns do tempo de execução de algoritmos são: $O(\log_2 n)$, $O(n)$, $O(n^2)$, and $O(k^n)$.

Resumo (cont.)

- Classes comuns do tempo de execução de algoritmos são: $O(\log_2 n)$, $O(n)$, $O(n^2)$, and $O(k^n)$.
- Um algoritmo pode ter diferenças de eficiência no melhor, pior e caso médio.

Resumo (cont.)

- Classes comuns do tempo de execução de algoritmos são: $O(\log_2 n)$, $O(n)$, $O(n^2)$, and $O(k^n)$.
- Um algoritmo pode ter diferenças de eficiência no melhor, pior e caso médio.
- A busca binária é muito mais rápida do que a busca linear. Porém, os dados devem estar ordenados.

Resumo (cont.)

- Classes comuns do tempo de execução de algoritmos são: $O(\log_2 n)$, $O(n)$, $O(n^2)$, and $O(k^n)$.
- Um algoritmo pode ter diferenças de eficiência no melhor, pior e caso médio.
- A busca binária é muito mais rápida do que a busca linear. Porém, os dados devem estar ordenados.
- Algoritmos exponenciais são impraticáveis de serem usados para grandes conjuntos de dados.