Introdução aos Algoritmos de Ordenação

Prof. Martín Vigil

Objetivo dos Algoritmos de Ordenação

- Seja uma sequência de objetos com alguma característica comparável (i.e, *chave*), por exemplo:
  - Sequência de pessoas, cada uma com um CPF distinto
  - Sequência de pessoas, cada uma com uma idade
- Não necessariamente todos os objetos são distintos
- Um algoritmo de ordenação ordenada a sequência em ordem
  - Não-decrescente: do menor para o maior
  - Não-crescente: dos maior para os menor

Algoritmos de ordenação ditos *simples*

Fáceis de implementar
porém
ineficientes ($O(n^2)$)

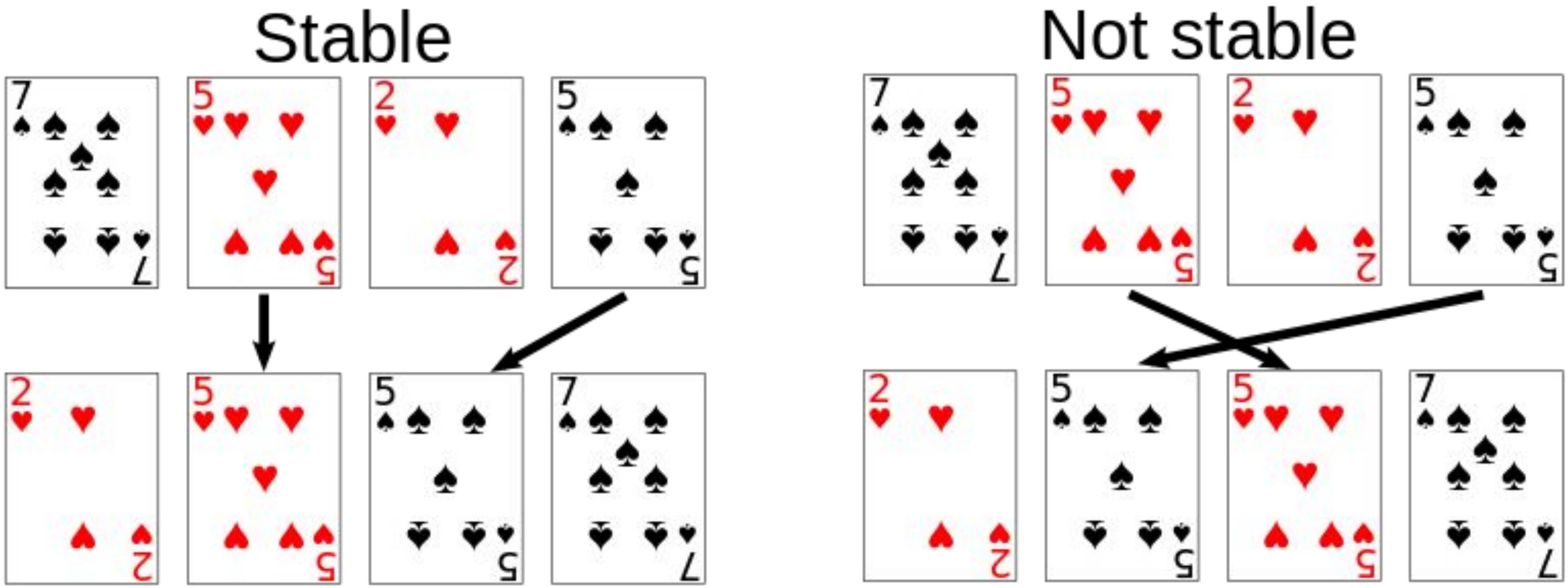
Algoritmos de ordenação ditos *eficientes*

Eficientes O(n log n)
porém
difíceis de implementar

Algoritmos *estáveis*

Não mudam a ordem relativa de dois objetos com chaves iguais

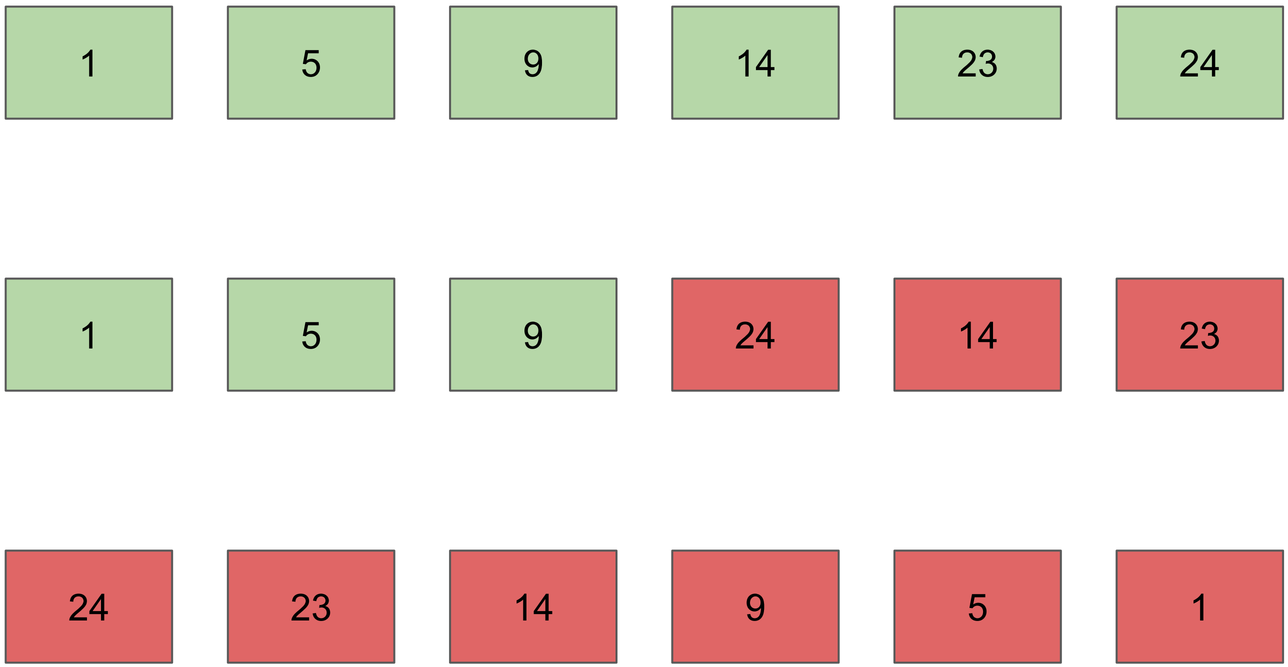
Estabilidade: ordem pelo valor numérico



Seja $v = v[1], v[2], ..., v[n]$ um vetor de $n > 0$ inteiros

Para $1 \leq i \leq n' \leq n$, $v' = v[i], v[i+1], ..., v[n']$ é um subvetor de v

Estados iniciais do vetor a ordenar: melhor, pior e médio casos



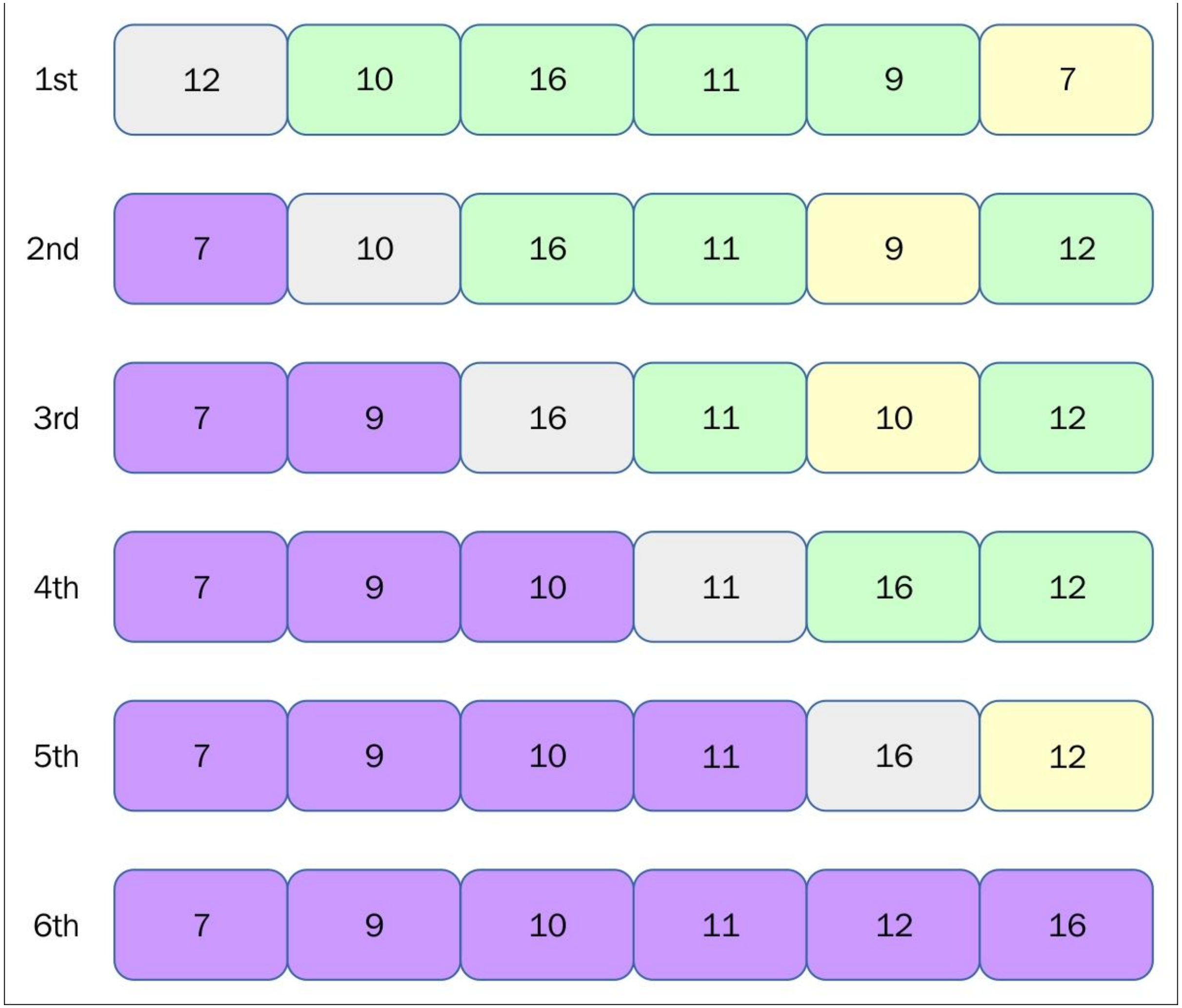
Algoritmos Simples

Selection Sort

Algoritmo da Seleção: Ideia

Selecionar o elemento mínimo de um vetor (resp. subvetor) e movê-lo para a primeira posição do vetor (resp. subvetor)

Algoritmo da Seleção: Exemplo



Algoritmo da Seleção

```
função selectionSort(vetor v, tamanho n)
    para j=1,2,..., n-1
        minimo = j
        para k=j+1,j+2,...,n
            se v[k] < v[minimo]
                minimo = k

        troca(v[minimo], v[j])
```

Algoritmo da Seleção: Complexidade

```
função selectionSort(vetor v, tamanho n)
    para j=1,2,..., n-1
        minimo = j
        para k=j+1,j+2,...,n
            se v[k] < v[minimo]
                minimo = k

        troca(v[minimo], v[j])
```

Constante

Algoritmo da Seleção: Complexidade

```
função selectionSort(vetor v, tamanho n)
    para j=1,2,..., n-1
        minimo = j
        para k=j+1,j+2,...,n
            se v[k] < v[minimo]
                minimo = k

        troca(v[minimo], v[j])
```

Depende de n

$$T(n) = \sum_{j=1}^{n-1}\left(c + \sum_{k=j+1}^{n} c'\right)$$

Algoritmo da Seleção: Complexidade

$$T(n) = \sum_{j=1}^{n-1}(c + \sum_{k=j+1}^{n} c')$$

$$= \sum_{j=1}^{n-1}(\sum_{k=j+1}^{n} c')$$

$$= \sum_{j=1}^{n-1}(n-j)c'$$

$$= \sum_{j=1}^{n-1} n-j$$

$$= \sum_{j=1}^{n-1} n-j \leq \sum_{j=1}^{n} n-j$$

$$\leq n-1+n-2+\ldots n-n$$

$$\leq n^2 - (1+2+\ldots+n) = n^2 - \frac{n^2+n}{2}$$

$$T(n) \in O(n^2)$$

Algoritmo da Seleção: Complexidade

$$
\begin{aligned}
T(n) &= \sum_{j=1}^{n-1}(c + \sum_{k=j+1}^{n} c') \\
&= \sum_{j=1}^{n-1}(\sum_{k=j+1}^{n} c') \\
&= \sum_{j=1}^{n-1}(n-j)c' \\
&= \sum_{j=1}^{n-1} n-j \\
&= \sum_{j=1}^{n-1} n-j \leq \sum_{j=1}^{n} n-j \\
&\leq n-1+n-2+\ldots n-n \\
&\leq n^2-(1+2+\ldots+n) = n^2 - \frac{n^2+n}{2}
\end{aligned}
$$

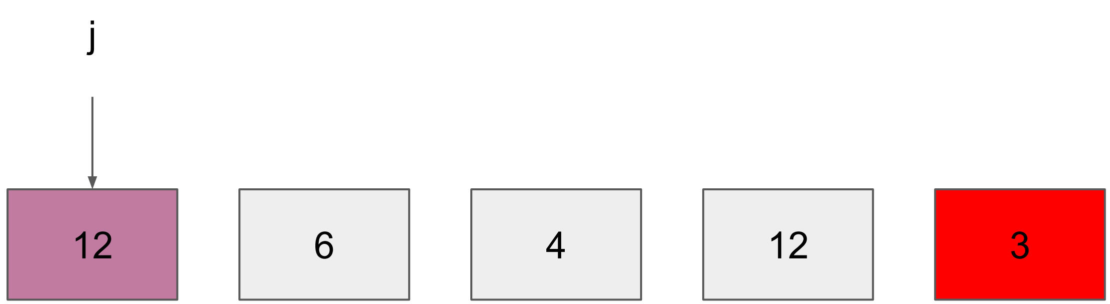
$$T(n) \in O(n^2)$$

Melhor caso?

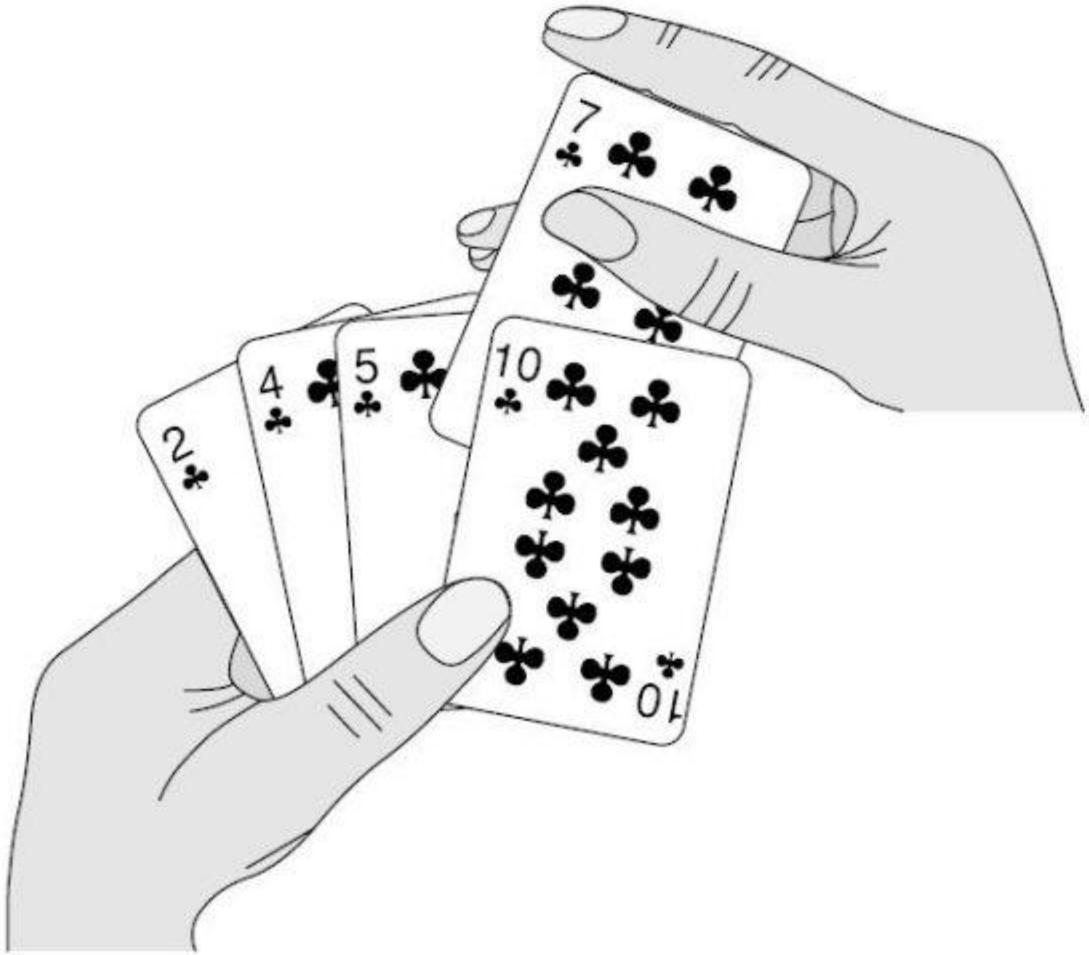
Pior caso?

Algoritmo da Seleção: Estável?

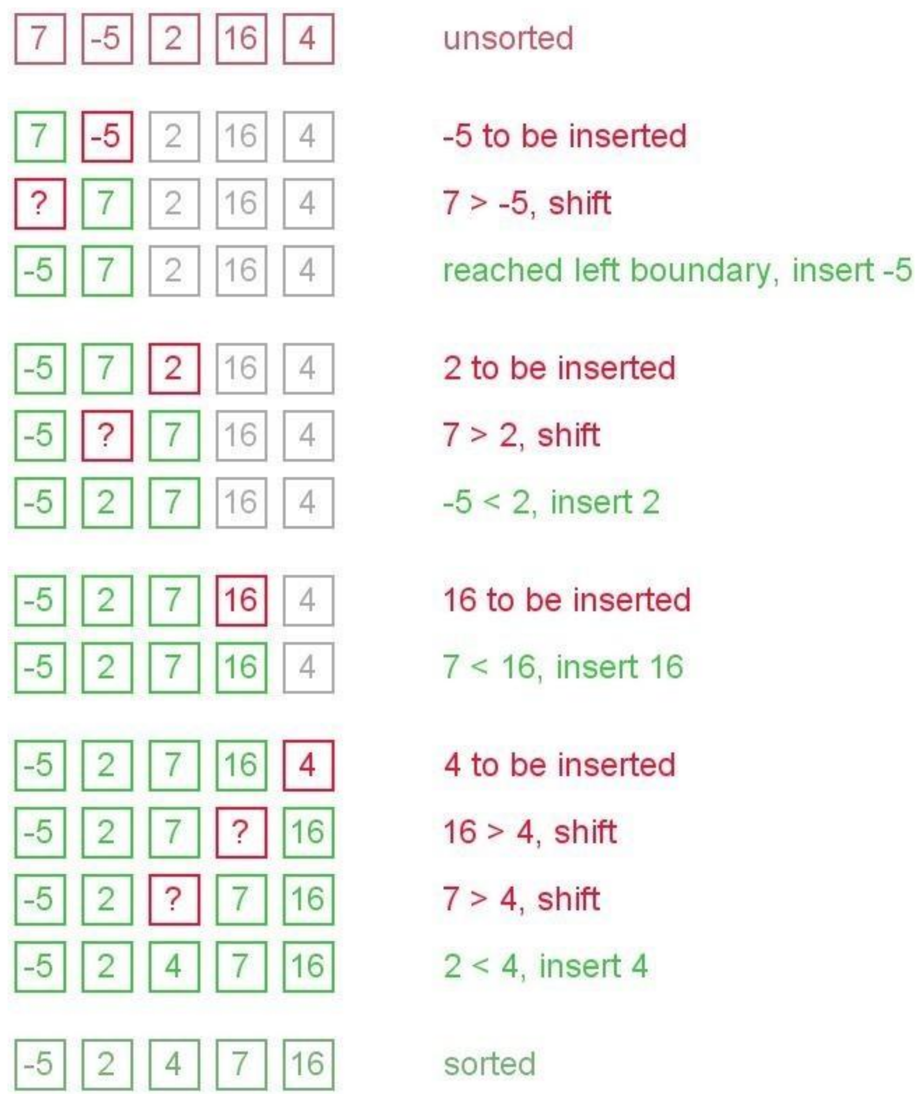
```
função selectionSort(vetor v, tamanho n)
    para j=1,2,..., n-1
        minimo = j
        para k=j+1,j+2,...,n
            se v[k] < v[minimo]
                minimo = k

        troca(v[minimo], v[j])
```



Algoritmo da Inserção (*Insertion Sort*)



Algoritmo da Inserção: Exemplo



Algoritmo da Inserção

```
função insertionSort(vetor v, tamanho n)
    para j=2,3,...,n
        tmp = v[j]
        k=j-1

        enquanto( k≥1 e v[k] > tmp)
            v[k+1] = v[k]
            k = k -1

        v[k+1] = tmp
fim
```

Algoritmo da Inserção: Complexidade

```
função insertionSort(vetor v, tamanho n)
    para j=2,3,...,n
        tmp = v[j]
        k=j-1

        enquanto( k≥1 e v[k] > tmp)
            v[k+1] = v[k]
            k = k -1

        v[k+1] = tmp
fim
```

Constante

Algoritmo da Inserção: Complexidade

```
função insertionSort(vetor v, tamanho n)
    para j=2,3,...,n
        tmp = v[j]
        k=j-1
        enquanto( k≥1 e v[k] > tmp)
            v[k+1] = v[k]
            k = k -1
        v[k+1] = tmp
fim
```

Depende de n

$$T(n) = \sum_{j=2}^{n} (c + \sum_{k=1}^{j-1} c')$$

Algoritmo da Inserção: Complexidade

$$
\begin{aligned}
T(n) &= \sum_{j=2}^{n}(c + \sum_{k=1}^{j-1} c') \\
&= \sum_{j=2}^{n}\sum_{k=1}^{j-1} c' \\
&= \sum_{j=2}^{n}(j-1)c' \\
&= \sum_{j=2}^{n} j-1 \\
&\leq \sum_{j=1}^{n} j-1 = 0+1+\ldots+n = \frac{n^2+n}{2}
\end{aligned}
$$

$$T(n) \in O(n^2)$$

Algoritmo da Inserção: Complexidade

$$T(n) = \sum_{j=2}^{n}(c + \sum_{k=1}^{j-1} c')$$

$$= \sum_{j=2}^{n}\sum_{k=1}^{j-1} c'$$

$$= \sum_{j=2}^{n}(j-1)c'$$

$$= \sum_{j=2}^{n} j - 1$$

$$\leq \sum_{j=1}^{n} j - 1 = 0 + 1 + \ldots + n = \frac{n^2+n}{2}$$

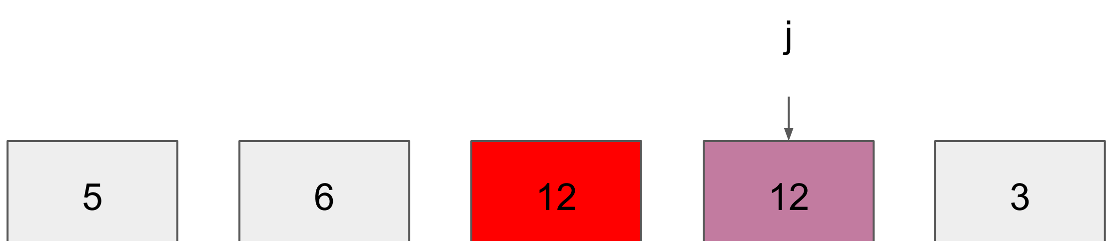
$$T(n) \in O(n^2)$$

Melhor caso?

Pior caso?

Algoritmo da Inserção: Estável?

```
função insertionSort(vetor v, tamanho n)
    para j=2,3,...,n
        tmp = v[j]
        k=j-1

        enquanto( k≥1 e v[k] > v[j])
            v[k+1] = v[k]
            k = k -1

        v[k+1] = tmp
fim
```

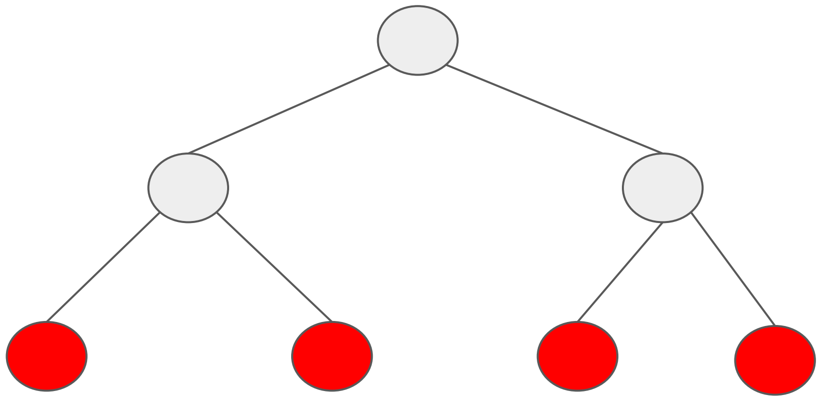


Algoritmos Eficientes

Heap Sort

Árvores binárias completas

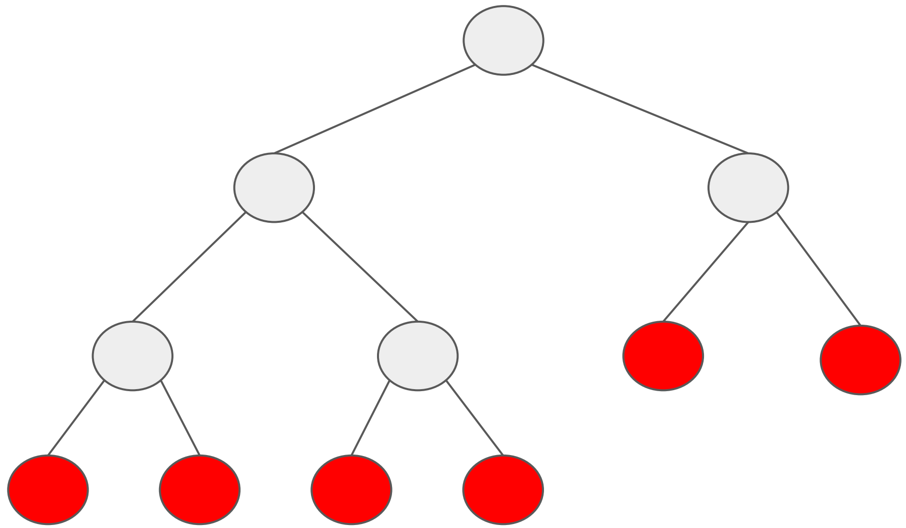
- Todo nodo tem 2 filhos ou nenhum



#nodos com 2 filhos = k
#nodos com 0 filhos = k + 1

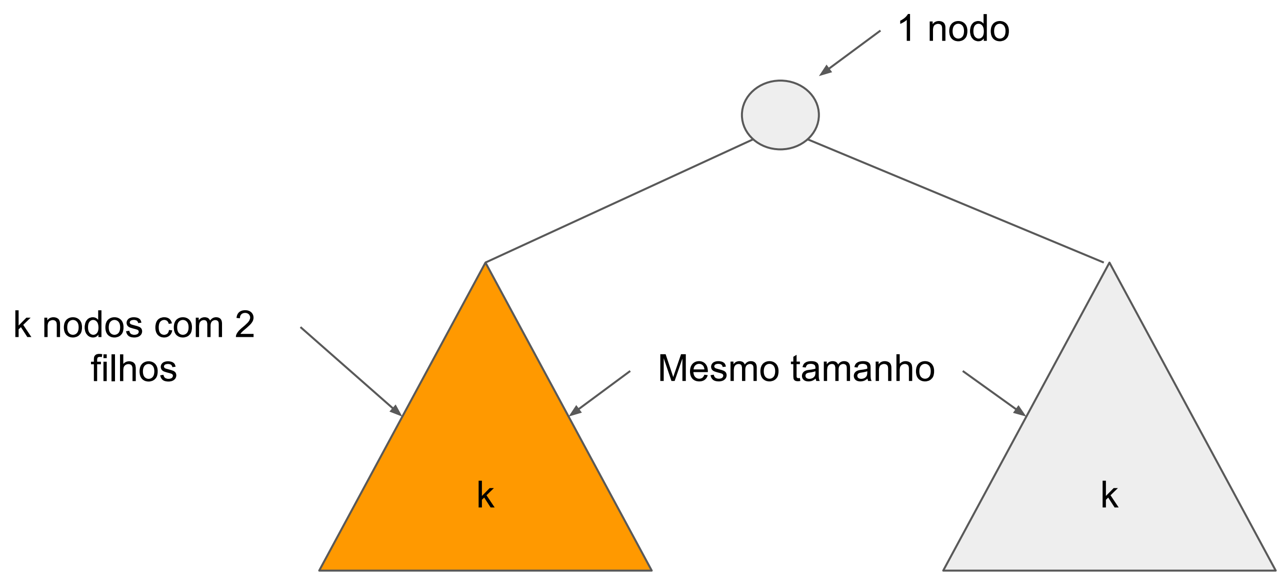
Árvores binárias completas

- Todo nodo tem 2 filhos ou nenhum



#nodos com 2 filhos = k
#nodos com 0 filhos = k + 1

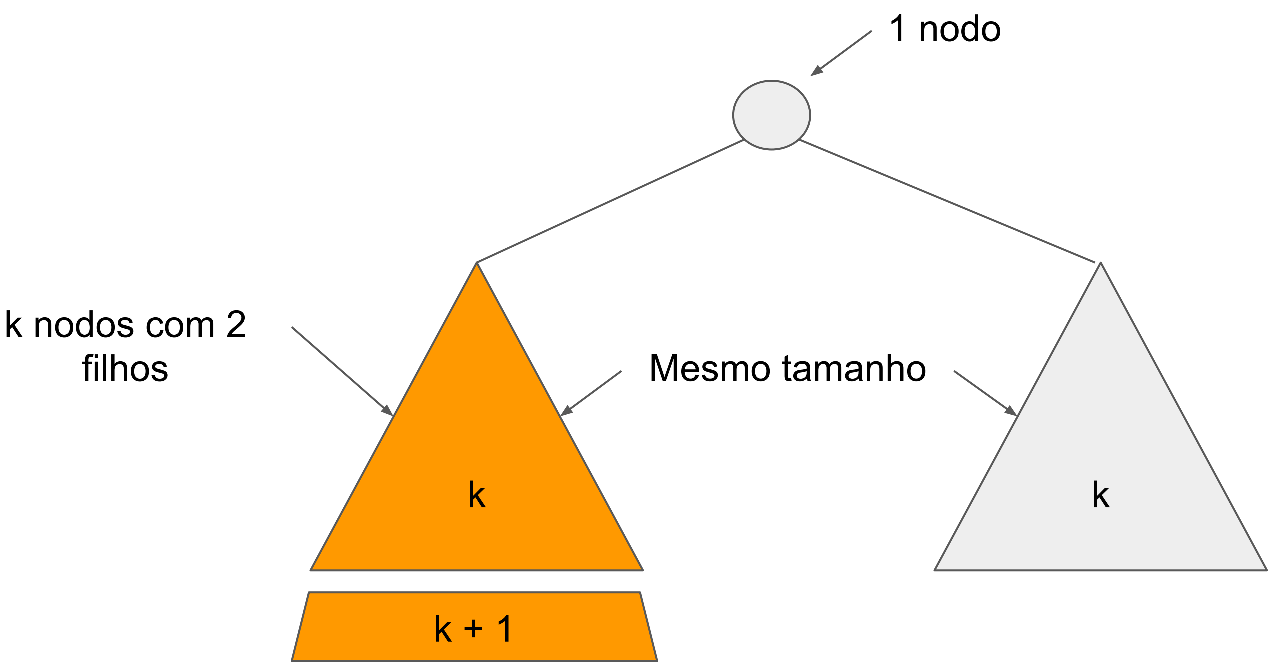
Árvores binárias completas: proporção



Total = 1 + k + k = 2k+1

SubárvoreEsq/Total = k / (2k+1) ≅ 1/2 do total para um k com valor muito grande

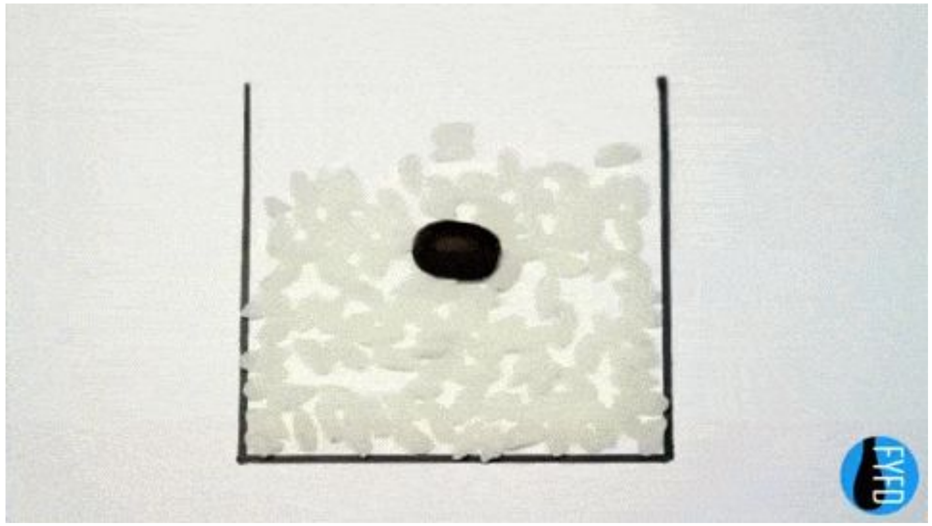
Árvores binárias completas: proporção

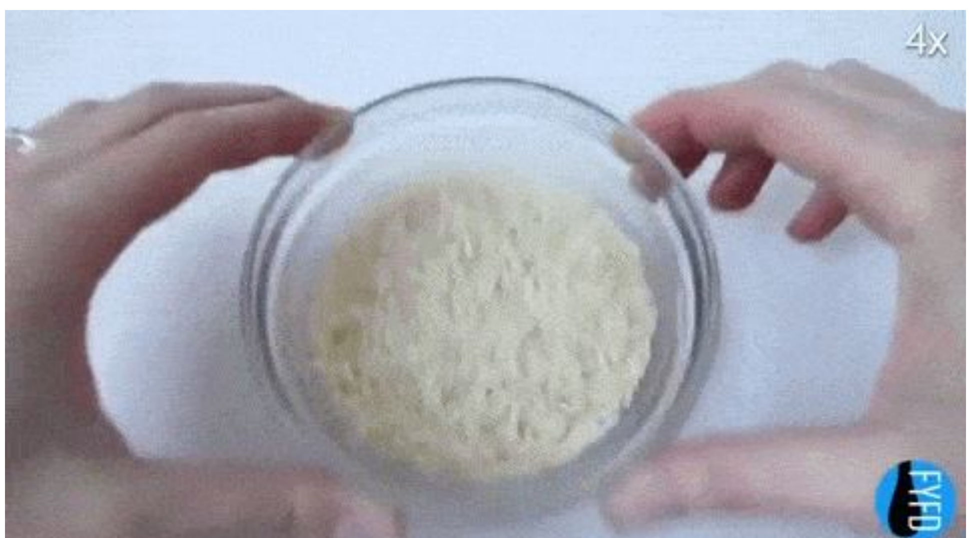


Total = 1 + k + k + 1 + k = 3k+2

SubárvoreEsq/Total = (2k+1) / (3k+2) ≅2/3 do total para um k com valor muito grande

O Efeito "Castanha do Pará"

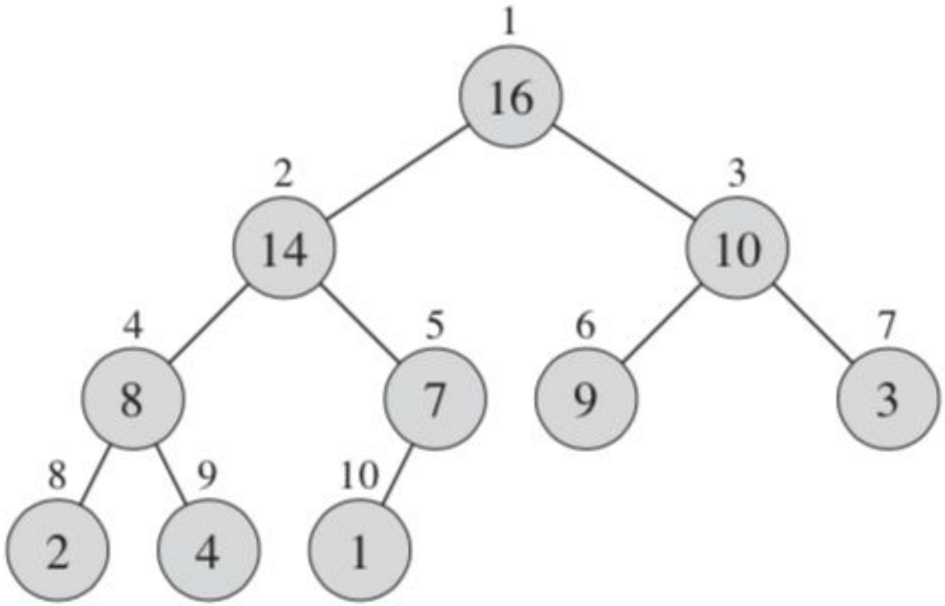


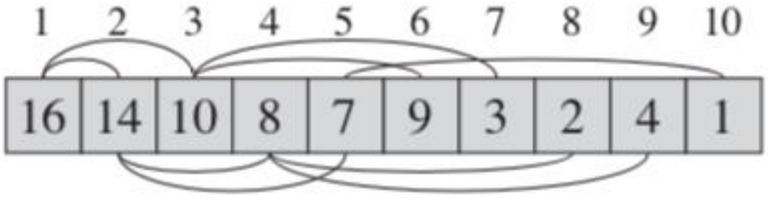


Heap (binária) máxima

- Sequência de objetos <o_1, o_2, … , o_n> que satisfaz a condição

$$o_i \geq o_{2i} \text{ e } o_i \geq o_{2i+1} \text{ para todo } i=1, 2, ..., n/2$$

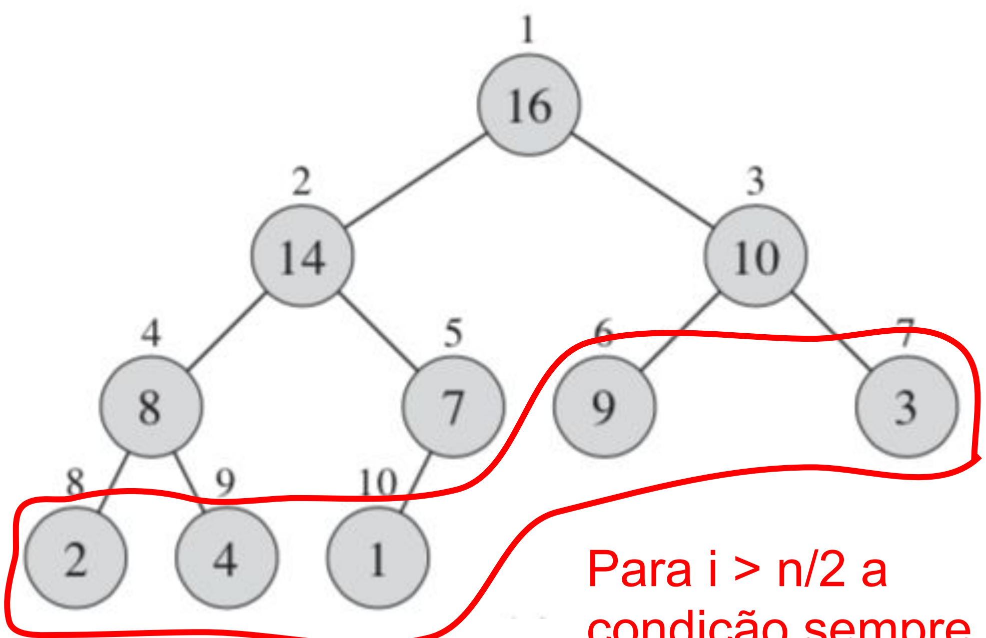




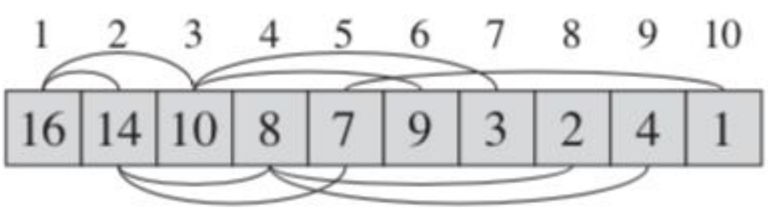
Heap (binária) máxima

- Sequência de objetos <o_1, o_2, … , o_n> que satisfaz a condição

$$o_i \geq o_{2i} \text{ e } o_i \geq o_{2i+1} \text{ para todo } i=1, 2, ..., n/2$$

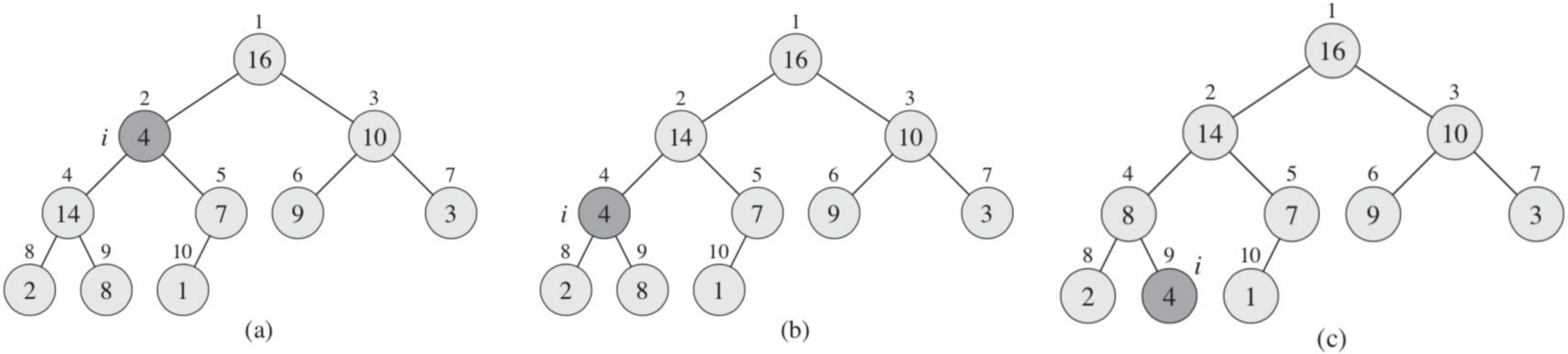


Para i > n/2 a condição sempre é satisfeita



Função *max-heapify*

- Move o i-ésimo objeto para *alguma* posição à direita para satisfazer a condição de heap máxima.
- Exemplo: i=2 em (a) abaixo



Função *max-heapify*

- Seja $A=\{v,t\}$ uma heap contendo $t>0$ objetos $o_1, o_2, \ldots, o_t$ armazenados no vetor $v=v[1], v[2], \ldots, v[n]$ onde $t \leq n$.
- Seja i o i-ésimo objeto o_i da heap

Função *max-heapify*

```
função max-heapify(A, i)
      esq = i*2
      dir  = i*2 + 1
      maximo = i
      se esq ≤ A.t E A.v[esq] > A.v[i]
            maximo  = esq
      se dir ≤ A.t E  A.v[dir] > A.v[maximo]
            maximo = dir
      se maximo ≠ i
           troca(A.v[i], A.v[maximo])
           max-Heapify(A, maximo)
```

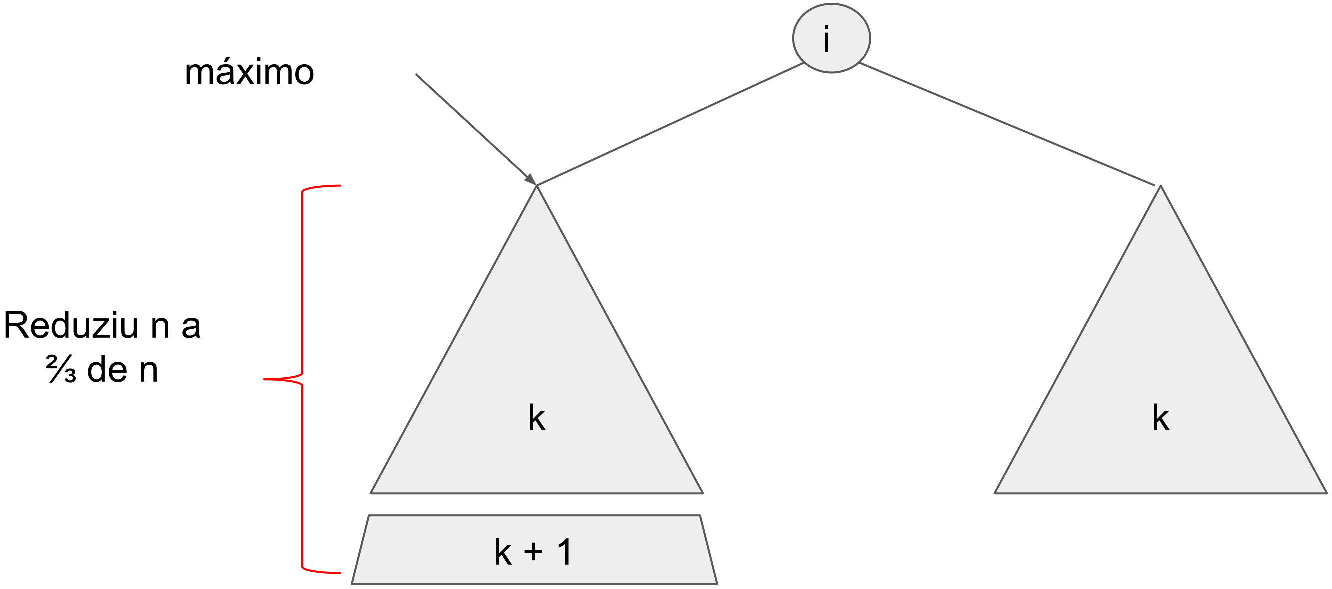
Função *max-heapify*

```
função max-heapify(A, i)
        esq = i*2
        dir  = i*2 + 1
        maximo = i
        se esq ≤ A.t E A.v[esq] > A.v[i]
              maximo  = esq
        se dir ≤ A.t E  A.v[dir] > A.v[maximo]
              maximo = dir
        se maximo ≠ i
             troca(A.v[i], A.v[maximo])
             max-Heapify(A, maximo)
```

Constante: $\Theta(1)$

Depende de n.
Reduziu qto n?

Pior caso: subárvores com alturas diferentes



Função *max-heapify*

```
função max-heapify(A, i)
      esq = i*2
      dir  = i*2 + 1
      maximo = i
      se esq ≤ A.t E A.v[esq] > A.v[i]
            maximo  = esq
      se dir ≤ A.t E  A.v[dir] > A.v[maximo]
            maximo = dir
      se maximo ≠ i
            troca(A.v[i], A.v[maximo])
            max-Heapify(A, maximo)
```

Constante: $\Theta(1)$

Reduz a ⅔ n

$$T(n) = T(\frac{2}{3}n) + \Theta(1) \in O(\log n)$$

Função *construirHeapMax*

- Organiza vetor v de modo que v seja uma heap máxima

```
função construirHeapMax(vetor v, tamanho n)
    A.v = v
    A.t = n
    para i = ⌊n/2⌋, ⌊n/2⌋-1, ...,1
        max-heapify(A,i)
    retorne A
```

Função *construirHeapMax*

- Organiza vetor v de modo que v seja uma heap máxima

```
função construirHeapMax(vetor v, tamanho n)
    A.v = v
    A.t = n
    para i = ⌊n/2⌋, ⌊n/2⌋-1, ...,1
        max-heapify(A,i)
    retorne A
```

O(n log n)?

Função *construirHeapMax*: Complexidade

- Organiza vetor v de modo que v seja uma heap máxima

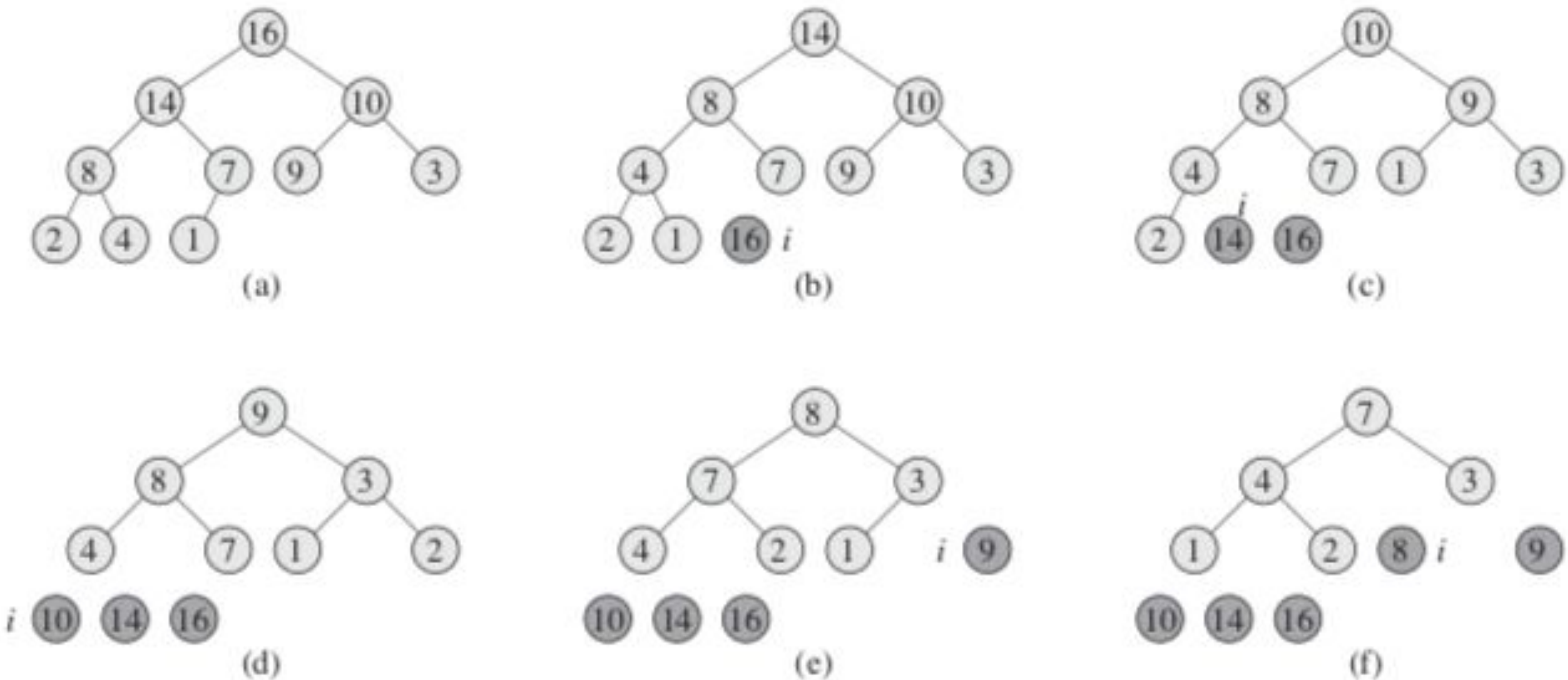
```
função construirHeapMax(vetor v, tamanho n)
    A.v = v
    A.t = n
    para i = ⌊n/2⌋, ⌊n/2⌋-1, ...,1
        max-heapify(A,i)
    retorne A
```

O(n)
Ver Cormen, Sec 6.3

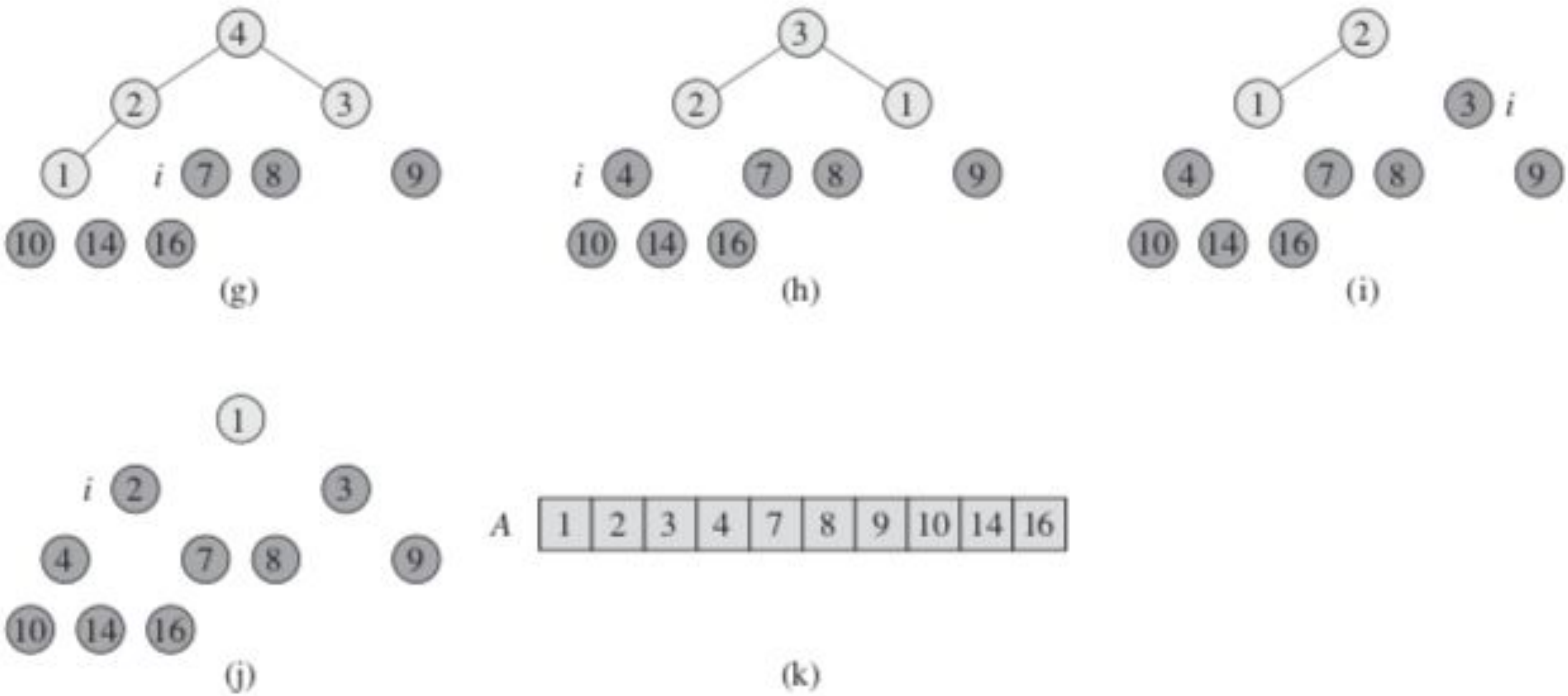
Heap Sort: Ideia

Na i-ésima iteração, troque primeiro objeto da heap de tamanho n-i com (n-i)-ésimo objeto do vetor

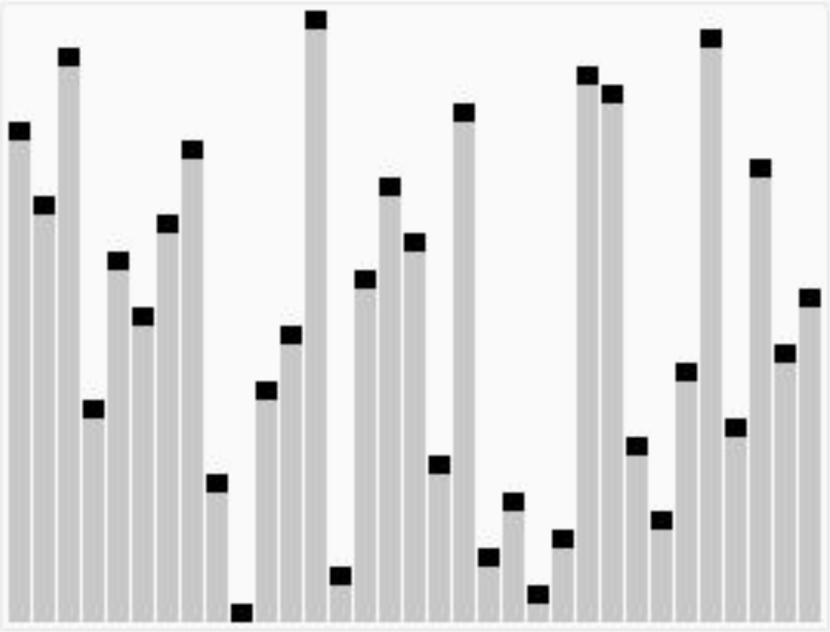
Heap Sort: Exemplo (1/2)



Heap Sort: Exemplo (2/2)



Heap Sort



Heap Sort: Algoritmo

```
função heapSort(vetor v, tamanho n)
    A = contruirHeapMáxima(v,n)
      para k=n, n-1, …, 2
           trocar(A.v[1], A.v[k])
           A.t = A.t - 1
           max-heapify(A, 1)
```

Heap Sort: Complexidade

```
função heapSort(vetor v, tamanho n)
    A = contruirHeapMáxima(v,n)
      para k=n, n-1, …, 2
            trocar(A.v[1], A.v[k])
            A.t = A.t - 1
            max-heapify(A, 1)
```

$O(n)$

Ver Cormen, Sec 6.3

$O(n \log n)$

$O(n \log n) + O(n) = O(n \log n)$