MC448 — Análise de Algoritmos I

Cid Carvalho de Souza   Cândida Nunes da Silva
Orlando Lee

10 de setembro de 2009

Ordenação

O problema da ordenação

Problema:
Rearranjar um vetor $A[1 \dots n]$ de inteiros de modo que fique em ordem crescente.

Ou simplesmente:

Problema:
Ordenar um vetor $A[1 \dots n]$ de inteiros.

Algoritmos de ordenação

Veremos vários algoritmos de ordenação:

- *Insertion sort*
- *Selection sort*
- *Mergesort*
- *Heapsort*
- *Quicksort*

Insertion sort

- **Idéia básica:** a cada passo mantemos o subvetor $A[1 \dots j-1]$ ordenado e inserimos o elemento $A[j]$ neste subvetor.
- Repetimos o processo para $j = 2, \dots, n$ e ordenamos o vetor.

Insertion sort – pseudo-código

```
INSERTION-SORT(A, n)
para j ← 2 até n faça
    chave ← A[j]
    ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
    i ← j − 1
    enquanto i ≥ 1 e A[i] > chave faça
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← chave
```

Já analisamos antes a corretude e complexidade.

Vamos analisar novamente a complexidade usando a notação assintótica.

Complexidade de tempo de Insertion sort

INSERTION-SORT(A, n)	Tempo
1 **para** $j \leftarrow 2$ **até** n **faça**	?
2 *chave* $\leftarrow A[j]$	?
3 ▷ Insere $A[j]$ em $A[1 \dots j-1]$	?
4 $i \leftarrow j - 1$	?
5 **enquanto** $i \geq 1$ **e** $A[i] >$ *chave* **faça**	?
6 $A[i+1] \leftarrow A[i]$	?
7 $i \leftarrow i - 1$	?
8 $A[i+1] \leftarrow$ *chave*	?

Consumo de tempo no pior caso: ?

Complexidade de tempo de Insertion sort

INSERTION-SORT(A, n)	Tempo
1 **para** $j \leftarrow 2$ **até** n **faça**	$\Theta(n)$
2 *chave* $\leftarrow A[j]$	$\Theta(n)$
3 ▷ Insere $A[j]$ em $A[1 \dots j-1]$	
4 $i \leftarrow j - 1$	$\Theta(n)$
5 **enquanto** $i \geq 1$ **e** $A[i] >$ *chave* **faça**	$nO(n) = O(n^2)$
6 $A[i+1] \leftarrow A[i]$	$nO(n) = O(n^2)$
7 $i \leftarrow i - 1$	$nO(n) = O(n^2)$
8 $A[i+1] \leftarrow$ *chave*	$O(n)$

Consumo de tempo: $O(n^2)$

Insertion sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  Vetor em ordem decrescente
  $\Theta(n^2)$ comparações
  $\Theta(n^2)$ movimentações
- Complexidade de tempo no melhor caso: $\Theta(n)$
  (vetor em ordem crescente)
  $O(n)$ comparações
  zero movimentações
- Complexidade de espaço/consumo espaço: $\Theta(n)$

Um pouco de terminologia

- Um algoritmo A tem complexidade de tempo (no **pior caso**) $O(f(n))$ se para **qualquer** entrada de tamanho n ele gasta tempo no **máximo** $O(f(n))$.
- Um algoritmo A tem complexidade de tempo no pior caso $\Theta(f(n))$ se para qualquer entrada de tamanho n ele gasta tempo no **máximo** $O(f(n))$ e para alguma entrada de tamanho n ele gasta tempo pelo menos $\Omega(f(n))$.
- Por exemplo, INSERTION SORT tem complexidade de tempo no **pior caso** $\Theta(n^2)$.

Um pouco de terminologia

- Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** pelo menos $O(f(n))$?
- Faz sentido dizer que um algoritmo tem complexidade de tempo no **pior caso** $\Omega(f(n))$?
- Faz sentido dizer que um algoritmo tem complexidade de tempo no **melhor caso** $\Omega(f(n))$?

Selection sort

- Mantemos um subvetor $A[1 \dots i-1]$ tal que:
  1. $A[1 \dots i-1]$ está **ordenado** e
  2. $A[1 \dots i-1] \leq A[i \dots n]$.

  A cada passo selecionamos o menor elemento em $A[i \dots n]$ e o colocamos em $A[i]$.
- Repetimos o processo para $i = 1, \dots, n-1$ e ordenamos vetor.

Selection sort – pseudo-código

$\text{SELECTION-SORT}(A, n)$

para $i \leftarrow 1$ **até** $n-1$ **faça**
    $min \leftarrow i$
    para $j \leftarrow i+1$ **até** n **faça**
        se $A[j] < A[min]$ **então** $min \leftarrow j$
    $A[i] \leftrightarrow A[min]$

Invariantes:

1. $A[1 \ldots i-1]$ está ordenado,
2. $A[1 \ldots i-1] \leq A[i \ldots n]$.

Complexidade de Selection sort

$\text{SELECTION-SORT}(A, n)$	Tempo
para $i \leftarrow 1$ **até** $n-1$ **faça**	?
$min \leftarrow i$	?
para $j \leftarrow i+1$ **até** n **faça**	?
se $A[j] < A[min]$ **então** $min \leftarrow j$	?
$A[i] \leftrightarrow A[min]$	?

Consumo de tempo no pior caso: ?

Complexidade de Selection sort

$\text{SELECTION-SORT}(A, n)$	Tempo
para $i \leftarrow 1$ **até** $n-1$ **faça**	$\Theta(n)$
$min \leftarrow i$	$\Theta(n)$
para $j \leftarrow i+1$ **até** n **faça**	$\Theta(n^2)$
se $A[j] < A[min]$ **então** $min \leftarrow j$	$\Theta(n^2)$
$A[i] \leftrightarrow A[min]$	$\Theta(n)$

Consumo de tempo no pior caso: $O(n^2)$

Selection sort

- Complexidade de tempo no pior caso: $\Theta(n^2)$
  $\Theta(n^2)$ comparações
  $\Theta(n)$ movimentações
- Complexidade de tempo no melhor caso: $\Theta(n^2)$
  Mesmo que o pior caso.
- Complexidade de espaço/consumo espaço: $\Theta(n)$

Conhecimento geral

- Para vetores com no máximo 10 elementos, o melhor algoritmo de ordenação costuma ser *Insertion sort*.
- Para um vetor que está quase ordenado, *Insertion sort* também é a melhor escolha.
- Algoritmos super-eficientes assintoticamente tendem a fazer muitas movimentações, enquanto *Insertion sort* faz poucas movimentações quando o vetor está quase ordenado.

Mergesort

O algoritmo *Mergesort* é um exemplo clássico de paradigma de divisão-e-conquista.

- Divisão: divida o vetor de n elementos em subvetores de tamanhos $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$.
- Conquista: recursivamente ordene cada subvetor.
- Combinação: intercale os subvetores ordenados para obter o vetor ordenado.

Mergesort – pseudo-código

```
MERGESORT(A, p, r)
  se p < r
     então q ← ⌊(p + r)/2⌋
           MERGESORT(A, p, q)
           MERGESORT(A, q + 1, r)
           INTERCALA(A, p, q, r)
```

A complexidade de MERGESORT é dada pela recorrência:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(f(n)),$$

onde $f(n)$ é a complexidade de INTERCALA.

Intercalação

Q que significa intercalar dois (sub)vetores ordenados?

Problema: Dados $A[p \dots q]$ e $A[q{+}1 \dots r]$ crescentes, rearranjar $A[p \dots r]$ de modo que ele fique em ordem crescente.

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Intercalação

	p				*q*				*r*
A	22	33	55	77	99	11	44	66	88

B									

Intercalação

	k								
A									

	i								*j*
B	22	33	55	77	99	88	66	44	11

Intercalação

		k							
A	11								

	i							*j*	
B	22	33	55	77	99	88	66	44	11

Intercalação

			k						
A	11	22							

		i						*j*	
B	22	33	55	77	99	88	66	44	11

Intercalação

				k					
A	11	22	33						
			i					j	
B	22	33	55	77	99	88	66	44	11

Intercalação

					k				
A	11	22	33	44					
			i				j		
B	22	33	55	77	99	88	66	44	11

Intercalação

						k			
A	11	22	33	44	55				
				i			j		
B	22	33	55	77	99	88	66	44	11

Intercalação

							k		
A	11	22	33	44	55	66			
				i		j			
B	22	33	55	77	99	88	66	44	11

Intercalação

								k	
A	11	22	33	44	55	66	77		
					i	j			
B	22	33	55	77	99	88	66	44	11

Intercalação

									k
A	11	22	33	44	55	66	77	88	
					$i = j$				
B	22	33	55	77	99	88	66	44	11

Intercalação

A	11	22	33	44	55	66	77	88	99
					j	i			
B	22	33	55	77	99	88	66	44	11

Intercalação

Pseudo-código

```
INTERCALA(A, p, q, r)
para i ← p até q faça
    B[i] ← A[i]
para j ← q + 1 até r faça
    B[r + q + 1 − j] ← A[j]
i ← p
j ← r
para k ← p até r faça
    se B[i] ≤ B[j]
        então A[k] ← B[i]
              i ← i + 1
        senão A[k] ← B[j]
              j ← j − 1
```

Complexidade de Intercala

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Tamanho da entrada: $n = r - p + 1$

Consumo de tempo: $\Theta(n)$

Corretude de Intercala

Invariante principal de Intercala:

No começo de cada iteração do laço das linhas 7–12, vale que:

1. $A[p \dots k-1]$ está ordenado,
2. $A[p \dots k-1]$ contém todos os elementos de $B[p \dots i-1]$ e de $B[j+1 \dots r]$,
3. $B[i] \geq A[k-1]$ e $B[j] \geq A[k-1]$.

Exercício. Prove que a afirmação acima é de fato um invariante de INTERCALA.

Exercício. (fácil) Mostre usando o invariante acima que INTERCALA é correto.

Corretude do Mergesort

```
MERGESORT(A, p, r)
  se p < r
     então q ← ⌊(p + r)/2⌋
           MERGESORT(A, p, q)
           MERGESORT(A, q + 1, r)
           INTERCALA(A, p, q, r)
```

O algoritmo está correto?

A corretude do algoritmo Mergesort apóia-se na corretude do algoritmo Intercala e segue facilmente **por indução** em $n := r - p + 1$.

Você consegue ver por quê?

Corretude do Mergesort

Base: Mergesort ordena vetores de tamanho 0 ou 1.

Hipótese de indução: Mergesort ordena vetores com $< n$ elementos.

Passo de indução: por hipótese de indução, Mergesort ordena os dois subvetores (de tamanho $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$).

Pela corretude de Intercala, segue que o vetor resultante da intercalação é um vetor ordenado de n elementos.

Complexidade de Mergesort

MERGESORT(A, p, r)
se $p < r$
então $q \leftarrow \lfloor (p + r)/2 \rfloor$
MERGESORT(A, p, q)
MERGESORT$(A, q + 1, r)$
INTERCALA(A, p, q, r)

$T(n)$: complexidade de pior caso de MERGESORT

Então

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n).$$

A solução da recorrência é $T(n) = \Theta(n \lg n)$.

Mergesort

- Complexidade de tempo: $\Theta(n \lg n)$
  $\Theta(n \lg n)$ comparações
  $\Theta(n \lg n)$ movimentações

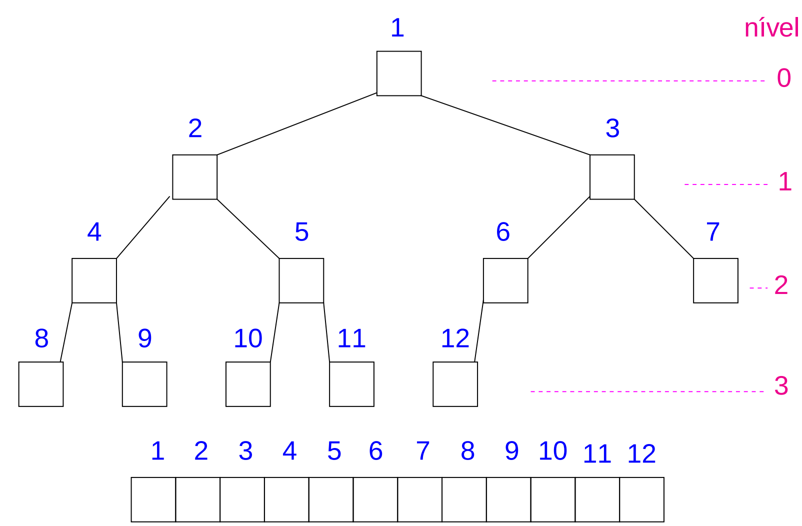
  O pior caso e o melhor caso têm a mesma complexidade.
- Complexidade de espaço/consumo espaço: $\Theta(n)$

  O *Mergesort* usa um vetor auxiliar de tamanho n para fazer a intercalação, mas o espaço ainda é $\Theta(n)$.
- O *Mergesort* é util para ordenação externa, quando não é possível armazenar todos os elementos na memória primária.

Heapsort

- O *Heapsort* é um algoritmo de ordenação que usa uma estrutura de dados sofisticada chamada *heap*.
- A complexidade de pior caso é $\Theta(n \lg n)$.
- *Heaps* podem ser utilizados para implementar filas de prioridade que são extremamente úteis em outros algoritmos.
- Um *heap* é um vetor A que simula uma árvore binária completa, com exceção possivelmente do último nível.

Heaps



Heaps

Considere um vetor $A[1 \dots n]$ representando um heap.

- Cada posição do vetor corresponde a um nó do heap.
- O pai de um nó i é $\lfloor i/2 \rfloor$.
- O nó 1 não tem pai.

Heaps

- Um nó i tem
  $2i$ como filho esquerdo e
  $2i + 1$ como filho direito.
- Naturalmente, o nó i
  tem filho esquerdo apenas se $2i \leq n$ e
  tem filho direito apenas se $2i + 1 \leq n$.
- Um nó i é uma folha se não tem filhos, ou seja, se $2i > n$.
- As folhas são $\lfloor n/2 \rfloor + 1, \dots, n-1, n$.

Níveis

Cada nível p, exceto talvez o último, tem exatamente 2^p nós e esses são

$$2^p, 2^p + 1, 2^p + 2, \dots, 2^{p+1} - 1.$$

O nó i pertence ao nível ???.
O nó i pertence ao nível $\lfloor \lg i \rfloor$.

Prova: Se p é o nível do nó i, então

$$\begin{array}{rcccccl} 2^p & \leq & i & < & 2^{p+1} & \Rightarrow \\ \lg 2^p & \leq & \lg i & < & \lg 2^{p+1} & \Rightarrow \\ p & \leq & \lg i & < & p+1 \end{array}$$

Logo, $p = \lfloor \lg i \rfloor$.

Portanto o número total de níveis é ???.
Portanto, o número total de níveis é $1 + \lfloor \lg n \rfloor$.

Altura

A altura de um nó i é o maior comprimento de um caminho de i a uma folha.

Os nós que têm altura zero são as folhas.

Qual é a altura de um nó i?

Altura

A altura de um nó i é o comprimento da seqüência

$$2\,i, 2^2\,i, 2^3\,i, \ldots, 2^h\,i$$

onde $2^h\,i \leq n < 2^{(h+1)}\,i$.

Assim,

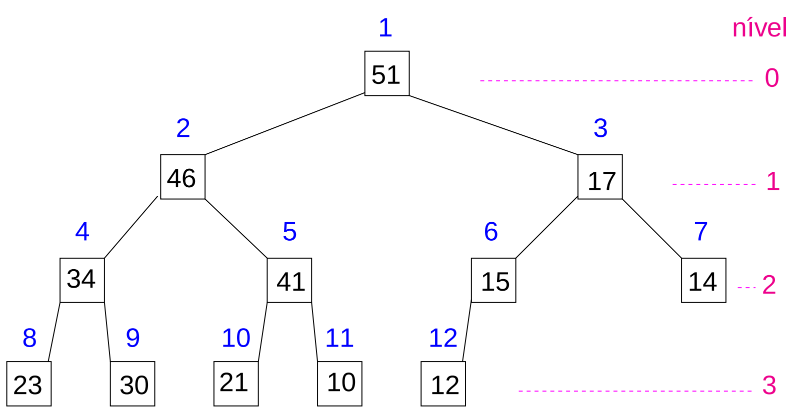
$$\begin{array}{rcccl} 2^h\,i & \leq & n & < & 2^{h+1} i \quad \Rightarrow \\ 2^h & \leq & n/i & < & 2^{h+1} \quad \Rightarrow \\ h & \leq & \lg(n/i) & < & h+1 \end{array}$$

Portanto, a altura de i é $\lfloor \lg(n/i) \rfloor$.

Max-heaps

- Um nó i satisfaz a propriedade de (max-)heap se $A[\lfloor i/2 \rfloor] \geq A[i]$ (ou seja, pai $\geq$ filho).
- Uma árvore binária completa é um *max-heap* se todo nó distinto da raiz satisfaz a propriedade de heap.
- O máximo ou maior elemento de um max-heap está na raiz.

Max-heap

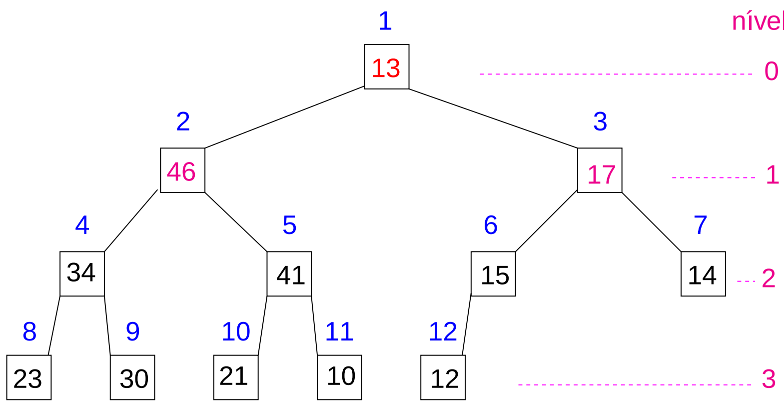


1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Min-heaps

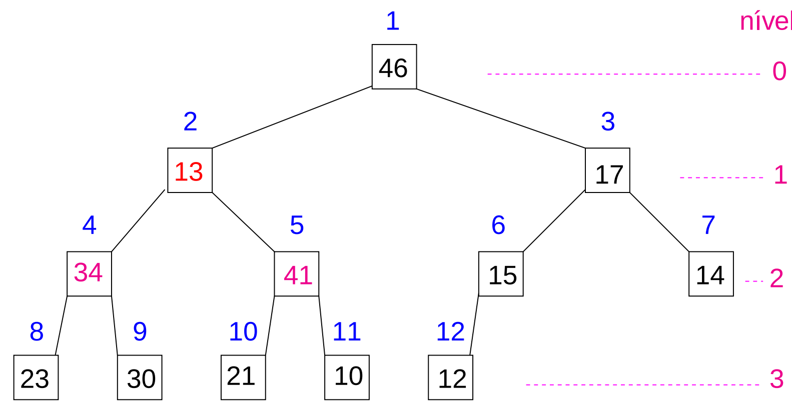
- Um nó i satisfaz a propriedade de (min-)heap se $A[\lfloor i/2 \rfloor] \leq A[i]$ (ou seja, pai $\leq$ filho).
- Uma árvore binária completa é um *min-heap* se todo nó distinto da raiz satisfaz a propriedade de min-heap.
- Vamos nos concentrar apenas em max-heaps.
- Os algoritmos que veremos podem ser facilmente modificados para trabalhar com min-heaps.

Manipulação de max-heap



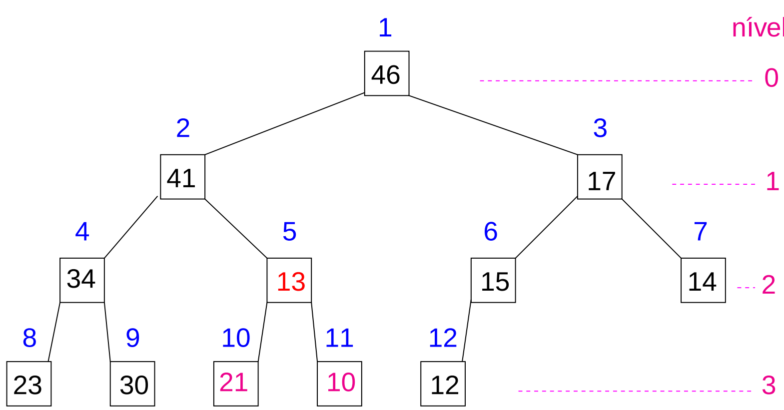
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



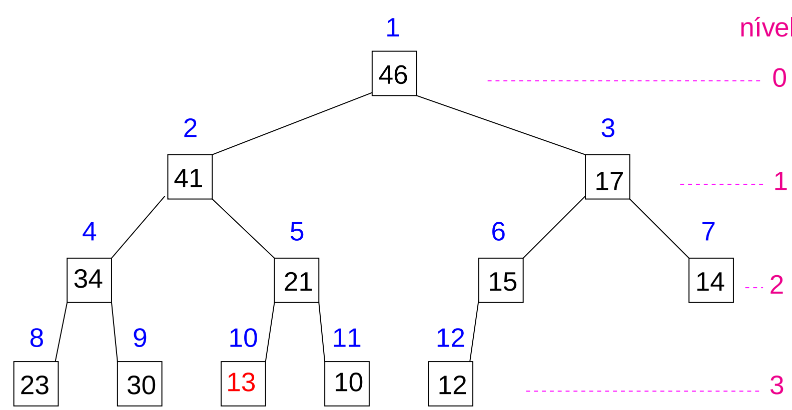
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



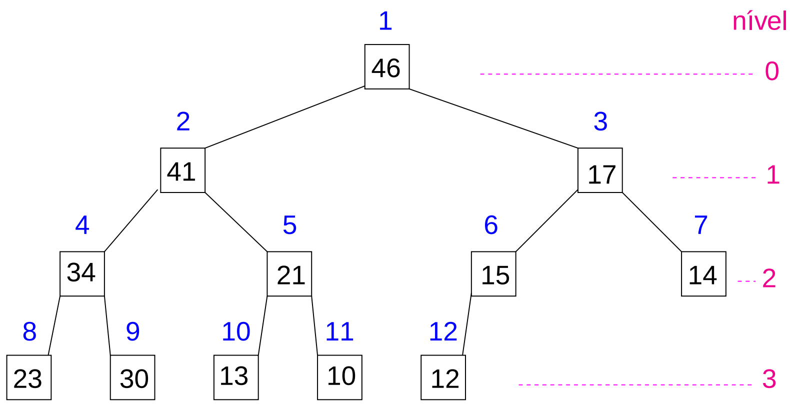
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

Manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

Manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

Manipulação de max-heap

Recebe $A[1 \dots n]$ e $i \geq 1$ tais que subárvores com raízes $2i$ e $2i + 1$ são max-heaps e rearranja A de modo que subárvore com raiz i seja um max-heap.

MAX-HEAPIFY(A, n, i)
1 $e \leftarrow 2i$
2 $d \leftarrow 2i + 1$
3 $\text{maior} \leftarrow i$
4 **se** $e \leq n$ **e** $A[e] > A[\text{maior}]$
5 **então** $\text{maior} \leftarrow e$
6 **se** $d \leq n$ e $A[d] > A[\text{maior}]$
7 **então** $\text{maior} \leftarrow d$
8 **se** $\text{maior} \neq i$
9 **então** $A[i] \leftrightarrow A[\text{maior}]$
10 MAX-HEAPIFY(A, n, maior)

Corretude de MAXHEAPIFY

A corretude de MAX-HEAPIFY segue por indução na altura h do nó i.

Base: para $h = 0$, o algoritmo funciona.

Hipótese de indução: MAX-HEAPIFY funciona para heaps de altura $< h$.

Passo de indução:

A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.
Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz maior em um max-heap (hipótese de indução).

Corretude de MAXHEAPIFY

Passo de indução:

A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i + 1]$.
Após a troca na linha 9, temos $A[2i], A[2i + 1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz maior em um max-heap (hipótese de indução).

A subárvore cuja raiz é o irmão de maior continua sendo um max-heap.

Logo, a subárvore com raiz i torna-se um max-heap e portanto, o algoritmo MAX-HEAPIFY está correto.

Complexidade de MAXHEAPIFY

MAX-HEAPIFY(A, n, i)	Tempo
$e \leftarrow 2i$	?
$d \leftarrow 2i + 1$	?
maior $\leftarrow i$	?
se $e \leq n$ **e** $A[e] > A[\text{maior}]$	?
então maior $\leftarrow e$	?
se $d \leq n$ e $A[d] > A[\text{maior}]$	?
então maior $\leftarrow d$	?
se maior $\neq i$	?
então $A[i] \leftrightarrow A[\text{maior}]$	?
MAX-HEAPIFY(A, n, maior)	?

h := altura de i = $\lfloor \lg \frac{n}{i} \rfloor$

$T(h)$:= complexidade de tempo no pior caso

Complexidade de MAXHEAPIFY

MAX-HEAPIFY(A, n, i)	Tempo
$e \leftarrow 2i$	$\Theta(1)$
$d \leftarrow 2i + 1$	$\Theta(1)$
maior $\leftarrow i$	$\Theta(1)$
se $e \leq n$ **e** $A[e] > A[\text{maior}]$	$\Theta(1)$
então maior $\leftarrow e$	$O(1)$
se $d \leq n$ e $A[d] > A[\text{maior}]$	$\Theta(1)$
então maior $\leftarrow d$	$O(1)$
se maior $\neq i$	$\Theta(1)$
então $A[i] \leftrightarrow A[\text{maior}]$	$O(1)$
MAX-HEAPIFY(A, n, maior)	$T(h-1)$

h := altura de i = $\lfloor \lg \frac{n}{i} \rfloor$

$T(h) \leq T(h-1) + \Theta(5) + O(2)$.

Complexidade de MAXHEAPIFY

h := altura de i = $\lfloor \lg \frac{n}{i} \rfloor$

$T(h)$:= complexidade de tempo no pior caso

$T(h) \leq T(h-1) + \Theta(1)$

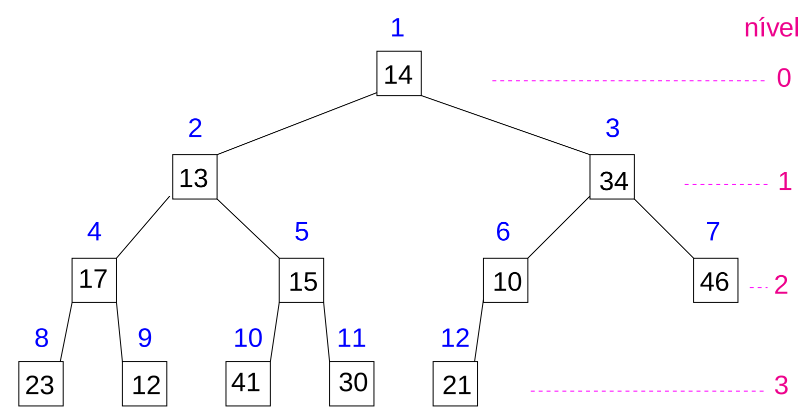
Solução assintótica: $T(n)$ é ???.

Solução assintótica: $T(n)$ é $O(h)$.

Como $h \leq \lg n$, podemos dizer que:

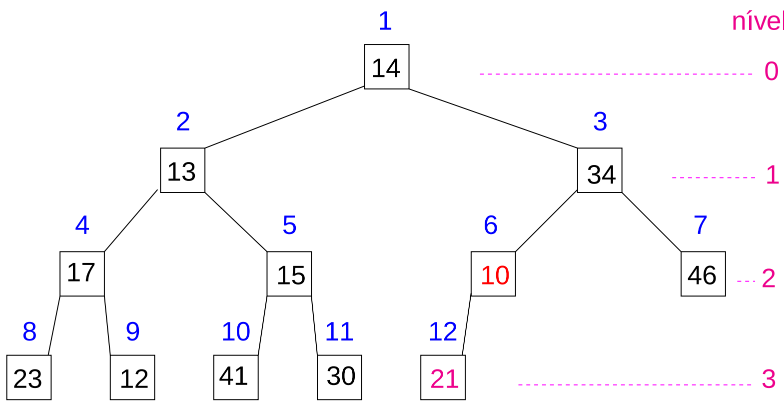
O consumo de tempo do algoritmo MAX-HEAPIFY é $O(\lg n)$ (ou melhor ainda, $O(\lg \frac{n}{i})$).

Construção de um max-heap



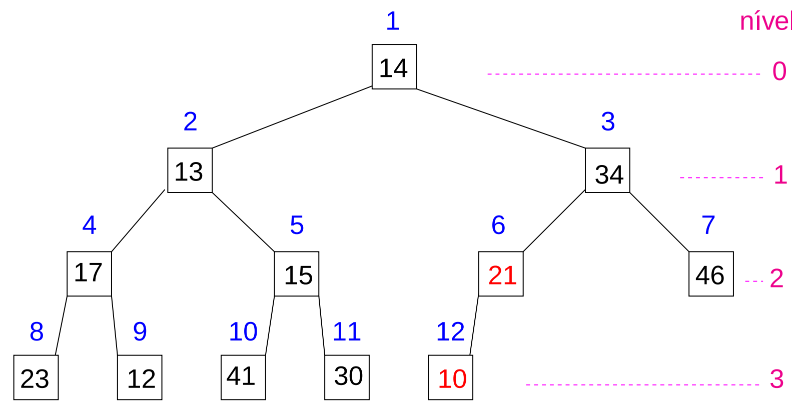
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



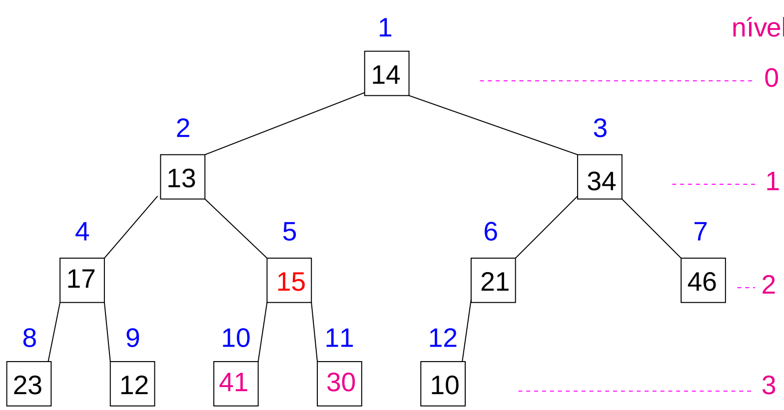
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



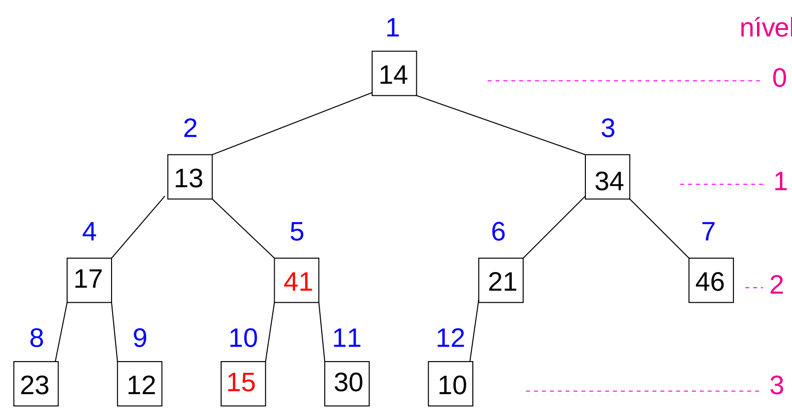
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



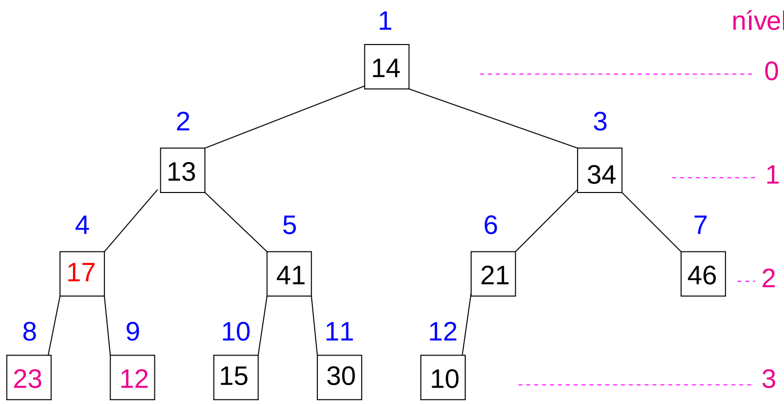
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



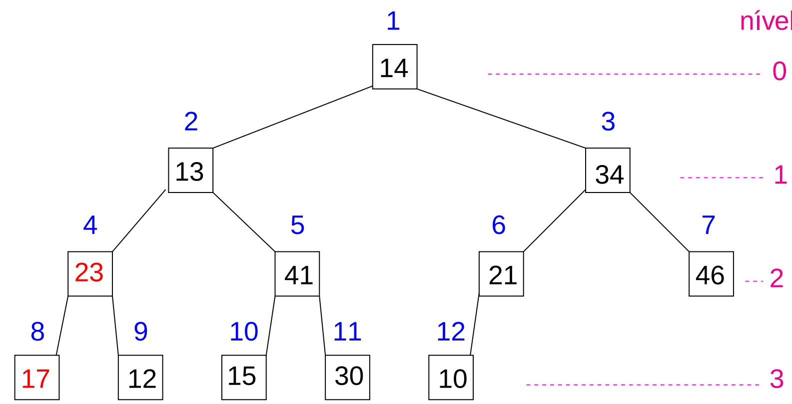
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



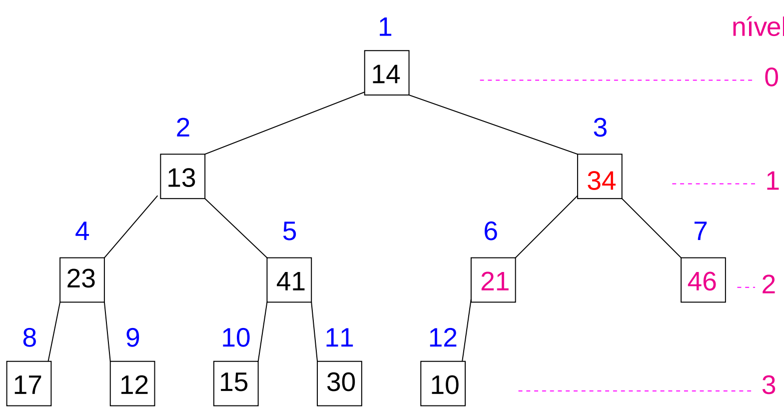
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



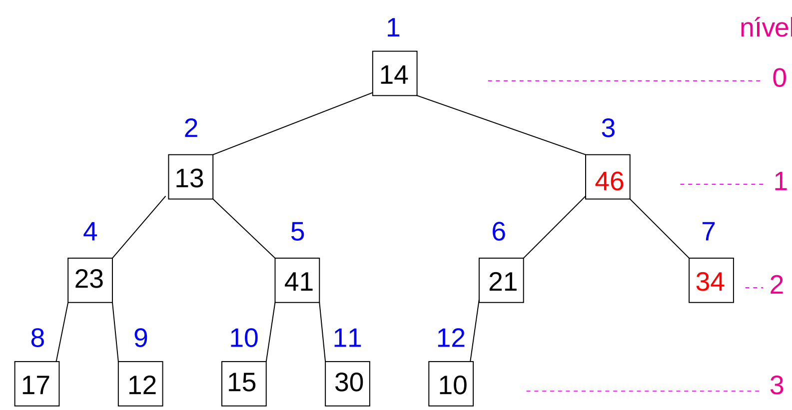
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



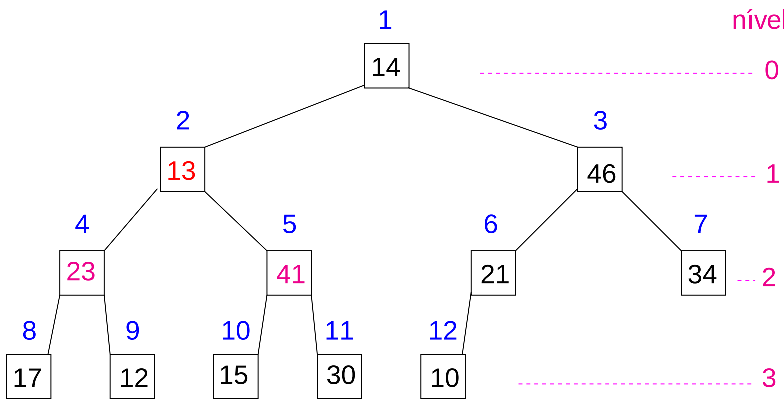
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



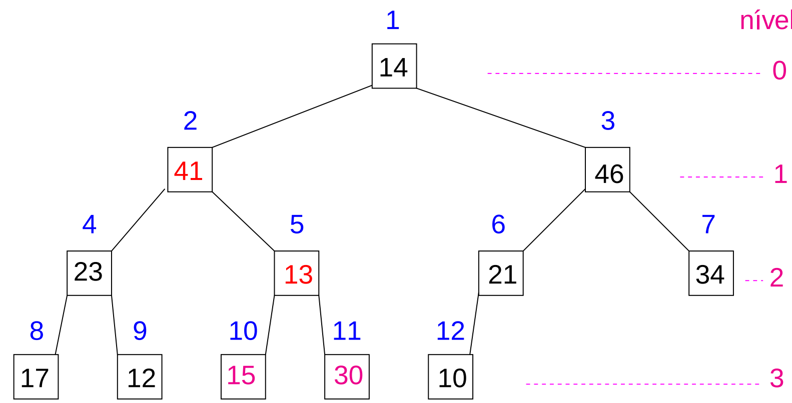
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



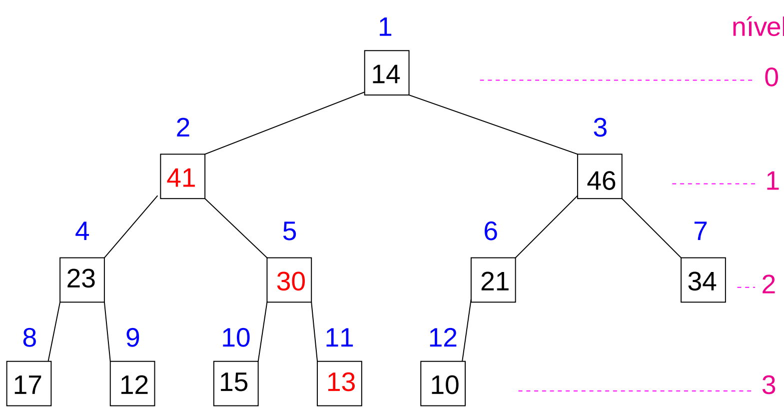
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



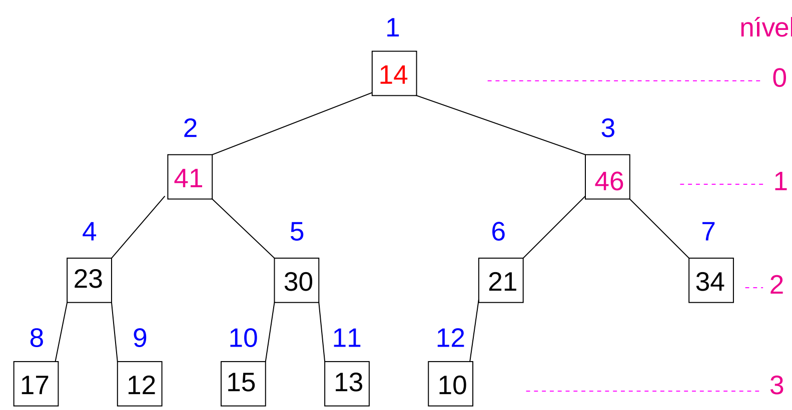
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	13	21	34	17	12	15	30	10

Construção de um max-heap



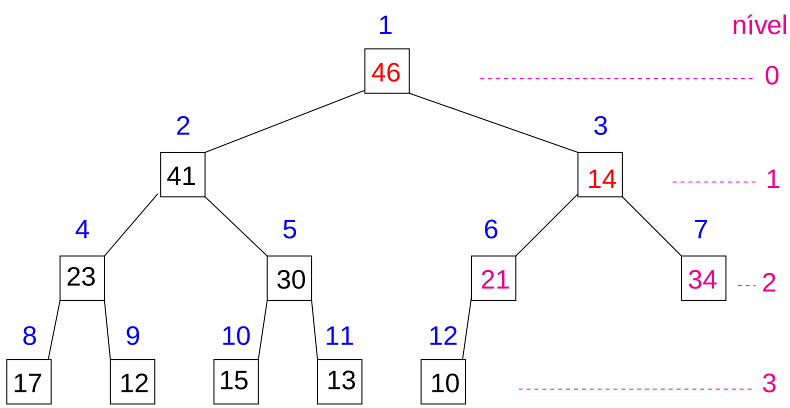
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



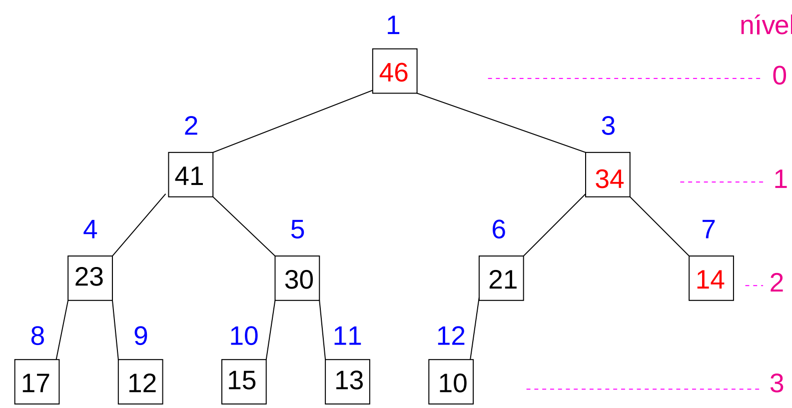
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



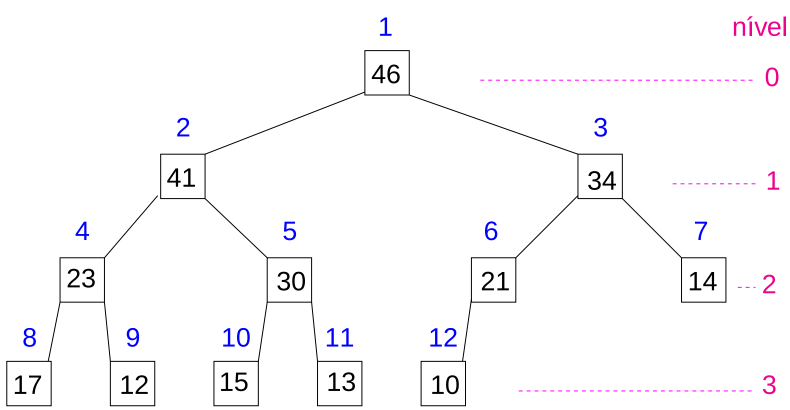
1	2	3	4	5	6	7	8	9	10	11	12
46	41	14	23	30	21	34	17	12	15	13	10

Construção de um max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Construção de um max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Construção de um max-heap

Recebe um vetor $A[1 \dots n]$ e rearranja A para que seja max-heap.

BUILDMAXHEAP(A, n)
para $i \leftarrow \lfloor n/2 \rfloor$ **decrescendo até** 1 **faça**
    MAX-HEAPIFY(A, n, i)

Invariante:
No início de cada iteração, $i+1, \dots, n$ são raízes de max-heaps.

$T(n)$ = complexidade de tempo no pior caso

Análise grosseira: $T(n)$ é $\frac{n}{2}\, O(\lg n) = O(n \lg n)$.

Construção de um max-heap

Análise mais cuidadosa: $T(n)$ é $O(n)$.

- Na iteração i são feitas $O(h_i)$ comparações e trocas no pior caso, onde h_i é a altura da subárvore de raiz i.
- Seja $S(h)$ a soma das alturas de todos os nós de uma árvore binária completa de altura h.
- A altura de um heap é $\lfloor \lg n \rfloor + 1$.

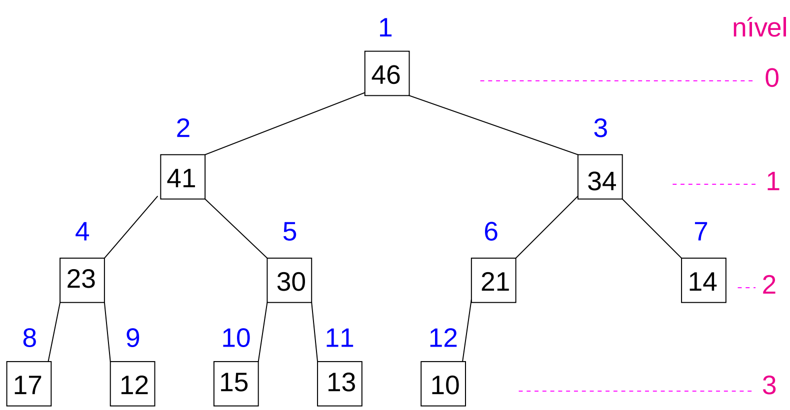
  A complexidade de BUILDMAXHEAP é $T(n) = O(S(\lg n))$.

Construção de um max-heap

- Pode-se provar por indução que $S(h) = 2^{h+1} - h - 2$.
- Logo, a complexidade de BUILDMAXHEAP é $T(n) = O(S(\lg n)) = O(n)$.

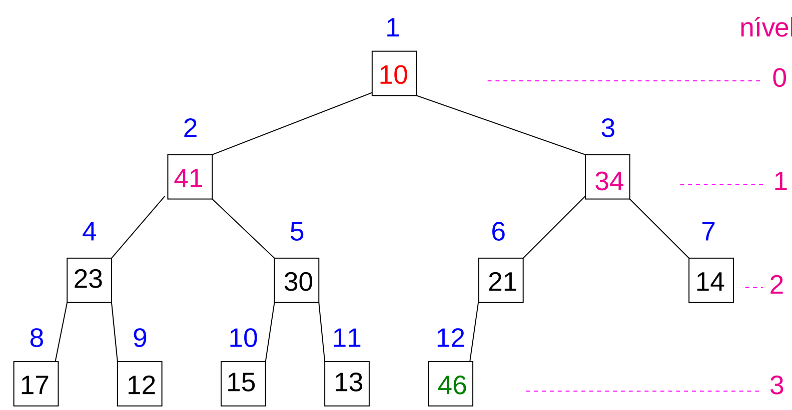
  Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)
- Veja no CLRS uma prova diferente deste fato.

HeapSort



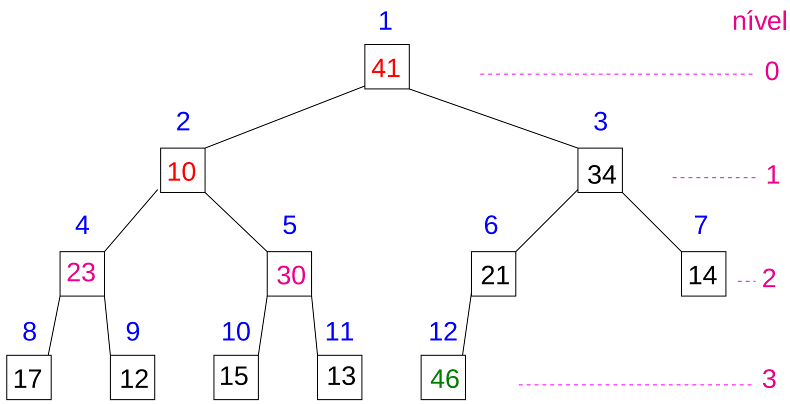
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

HeapSort



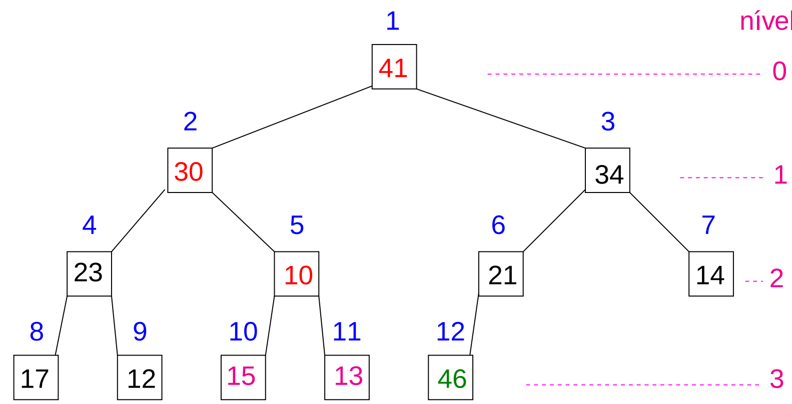
1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46

HeapSort



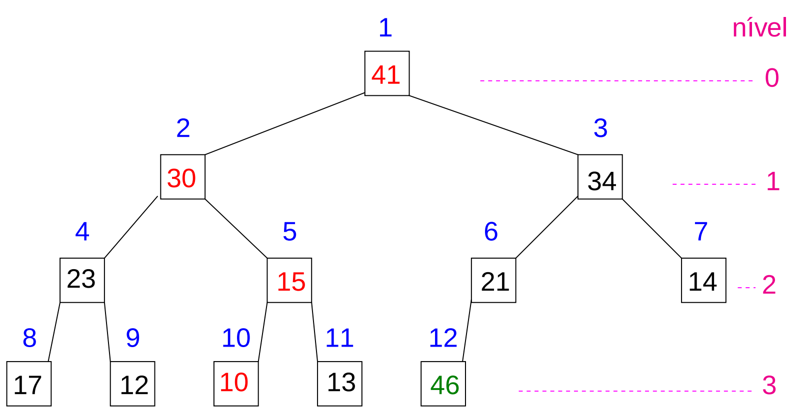
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

HeapSort



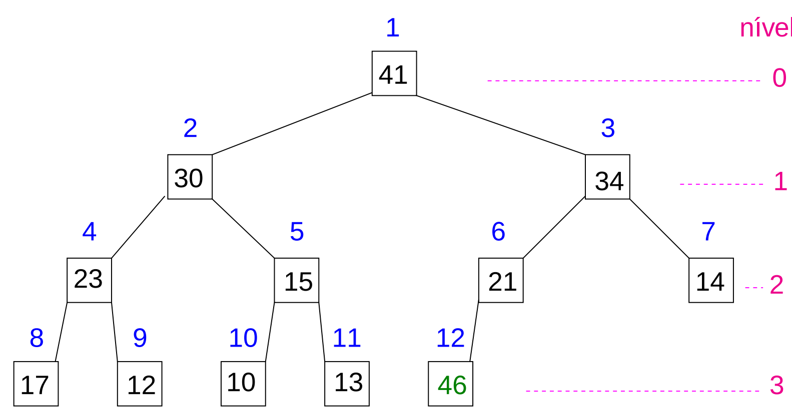
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

HeapSort



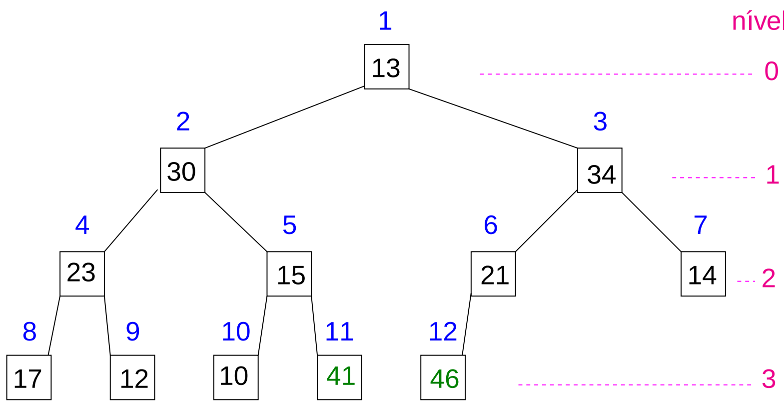
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



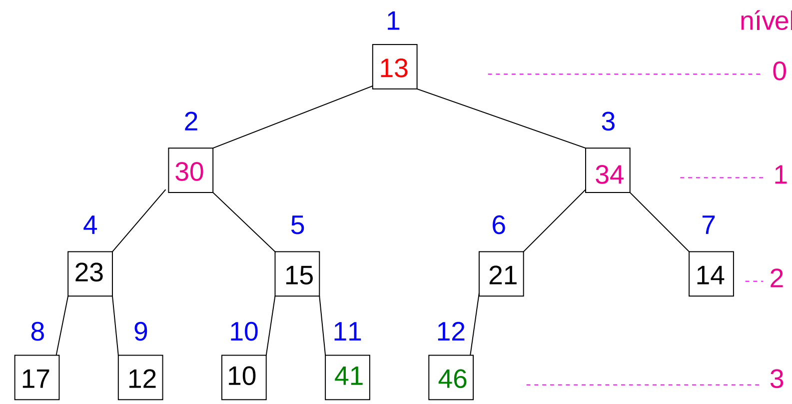
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



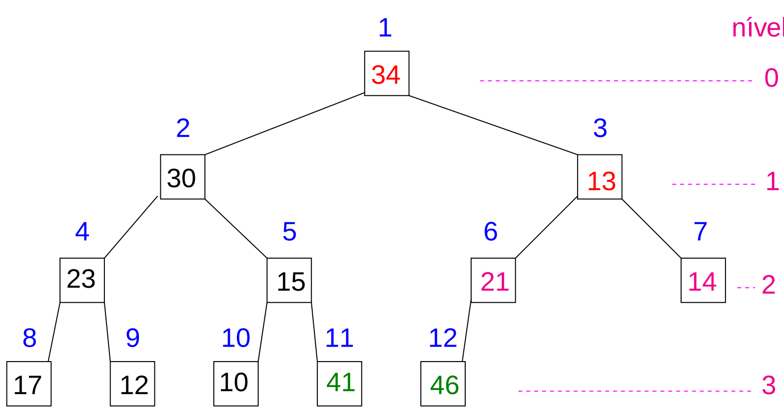
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



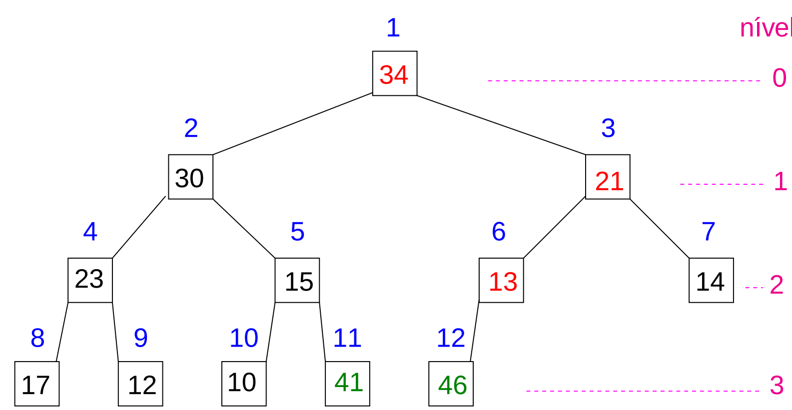
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



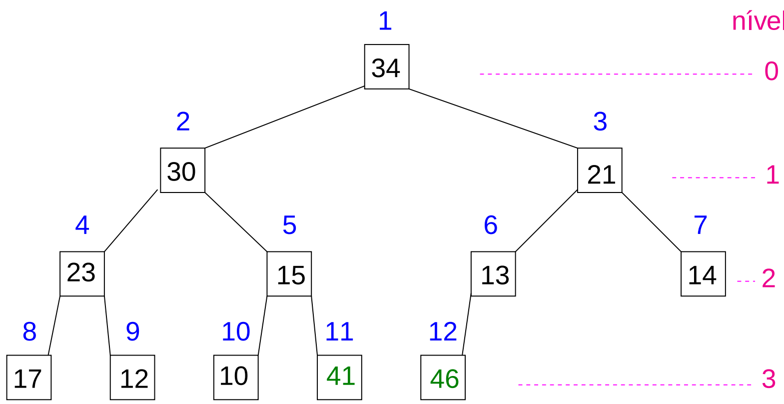
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

HeapSort



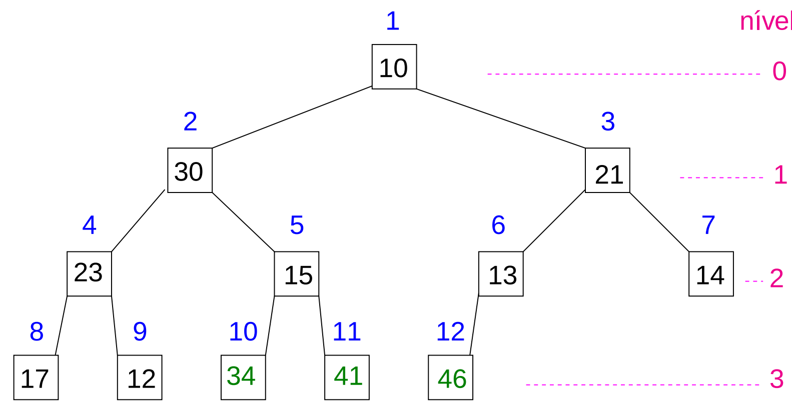
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



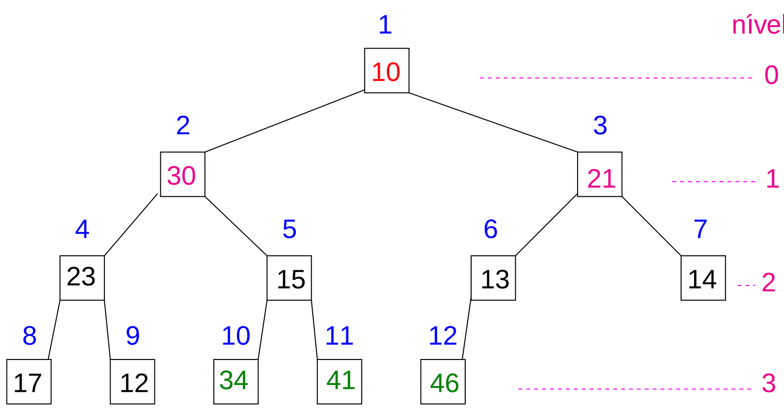
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



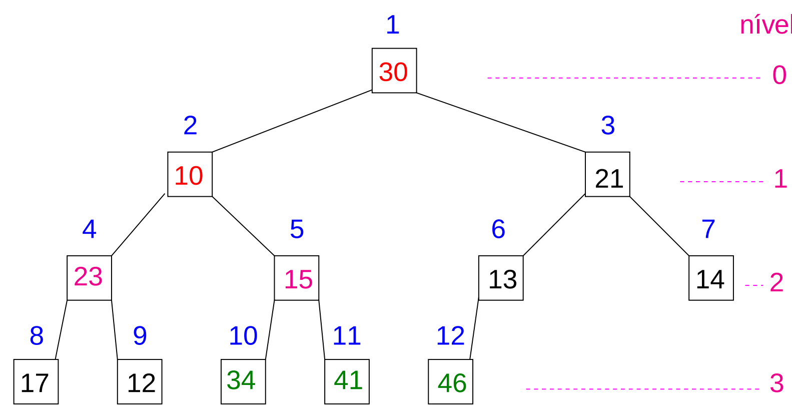
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



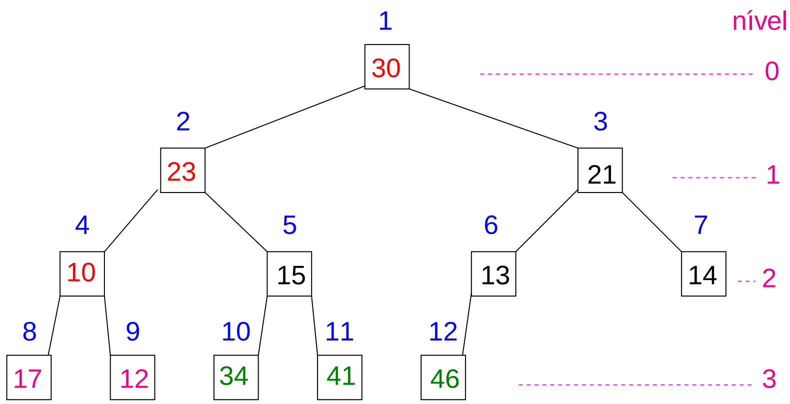
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



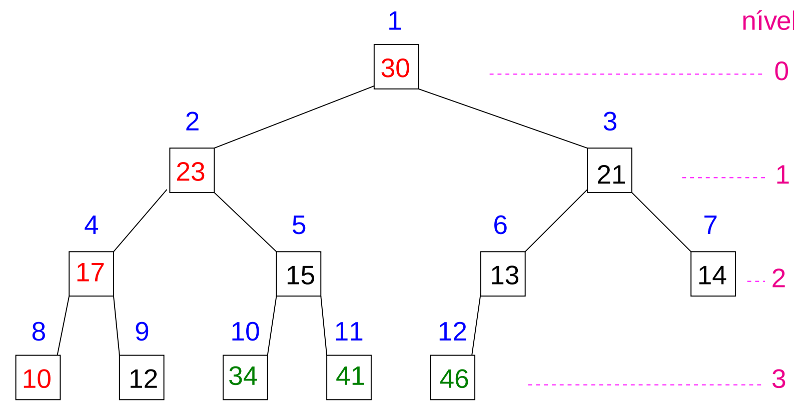
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

HeapSort



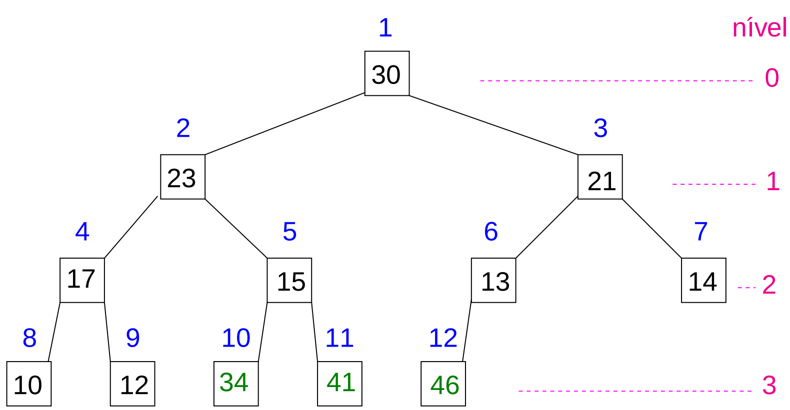
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

HeapSort



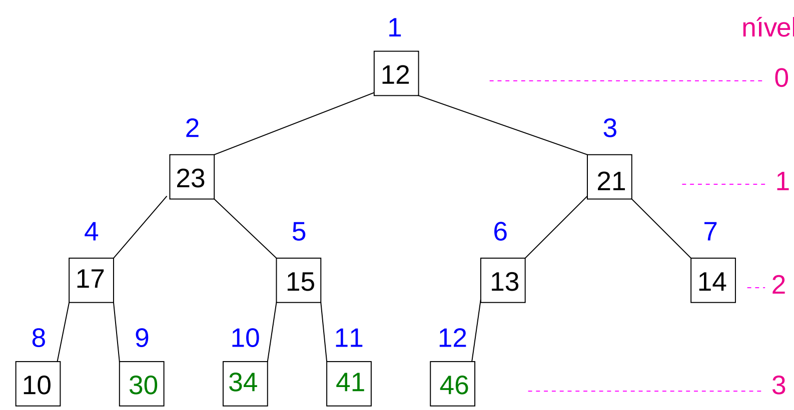
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

HeapSort



1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

HeapSort



1	2	3	4	5	6	7	8	9	10	11	12
12	23	21	17	15	13	14	10	30	34	41	46

HeapSort

Algoritmo rearranja $A[1 \dots n]$ em ordem crescente.

```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  m ← n
3  para i ← n decrescendo até 2 faça
4      A[1] ↔ A[i]
5      m ← m − 1
6      MAX-HEAPIFY(A, m, 1)
```

Invariantes:

No início de cada iteração na linha 3 vale que:

1. $A[m \dots n]$ é crescente;
2. $A[1 \dots m] \leq A[m+1]$;
3. $A[1 \dots m]$ é um max-heap.

HeapSort

Algoritmo rearranja $A[1 \dots n]$ em ordem crescente.

HEAPSORT(A, n)	Tempo
1 BUILD-MAX-HEAP(A, n)	?
2 $m \leftarrow n$	?
3 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**	?
4 $A[1] \leftrightarrow A[i]$	?
5 $m \leftarrow m - 1$	?
6 MAX-HEAPIFY$(A, m, 1)$	?

$T(n) =$ complexidade de tempo no pior caso

HeapSort

Algoritmo rearranja $A[1 \dots n]$ em ordem crescente.

HEAPSORT(A, n)	Tempo
1 BUILD-MAX-HEAP(A, n)	$\Theta(n)$
2 $m \leftarrow n$	$\Theta(1)$
3 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
4 $A[1] \leftrightarrow A[i]$	$\Theta(n)$
5 $m \leftarrow m - 1$	$\Theta(n)$
6 MAX-HEAPIFY$(A, m, 1)$	$nO(\lg n)$

$T(n) = ??$ $T(n) = nO(\lg n) + \Theta(4n + 1) = O(n \lg n)$

A complexidade de HEAPSORT no pior caso é $O(n \lg n)$.

Como seria a complexidade de tempo no melhor caso?

Filas com prioridades

Uma fila com prioridades é um tipo abstrato de dados que consiste de uma coleção S de itens, cada um com um valor ou prioridade associada.

Algumas operações típicas em uma fila com prioridades são:

MAXIMUM(S): devolve o elemento de S com a maior prioridade;

EXTRACT-MAX(S): remove e devolve o elemento em S com a maior prioridade;

INCREASE-KEY(S, x, p): aumenta o valor da prioridade do elemento x para p; e

INSERT(S, x, p): insere o elemento x em S com prioridade p.

Implementação com max-heap

HEAP-MAX(A, n)
devolva $A[1]$

Complexidade de tempo: $\Theta(1)$.

HEAP-EXTRACT-MAX(A, n)
$\triangleright$ $n \geq 1$
max $\leftarrow A[1]$
$A[1] \leftarrow A[n]$
$n \leftarrow n - 1$
MAX-HEAPIFY$(A, n, 1)$
devolva max

Complexidade de tempo: $O(\lg n)$.

Implementação com max-heap

HEAP-INCREASE-KEY$(A, i, prior)$
$\triangleright$ Supõe que *prior* $\geq A[i]$
$A[i] \leftarrow prior$
enquanto $i > 1$ e $A[\lfloor i/2 \rfloor] < A[i]$ **faça**
    $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
    $i \leftarrow \lfloor i/2 \rfloor$

Complexidade de tempo: $O(\lg n)$.

MAX-HEAP-INSERT$(A, n, prior)$
$n \leftarrow n + 1$
$A[n] \leftarrow -\infty$
HEAP-INCREASE-KEY$(A, n, prior)$

Complexidade de tempo: $O(\lg n)$.

Cid Carvalho de Souza, Cândida Nunes da Silva, Orlando Lee MC448 — Análise de Algoritmos

QuickSort

O algoritmo QUICKSORT segue o paradigma de divisão-e-conquista.

Divisão: divida o vetor em dois subvetores $A[p \ldots q-1]$ e $A[q+1 \ldots r]$ tais que

	p	q	r
A	$\leq x$	x	$> x$

$$A[p \ldots q-1] \leq A[q] < A[q+1 \ldots r]$$

Conquista: ordene os dois subvetores recursivamente usando o QUICKSORT;

Combinação: nada a fazer, o vetor está ordenado.

Cid Carvalho de Souza, Cândida Nunes da Silva, Orlando Lee MC448 — Análise de Algoritmos

Partição

Problema: Rearranjar um dado vetor $A[p \ldots r]$ e devolver um índice q, $p \leq q \leq r$, tais que

$$A[p \ldots q-1] \leq A[q] < A[q+1 \ldots r]$$

Entrada:

	p									r
A	99	33	55	77	11	22	88	66	33	44

Saída:

	p				q					r
A	33	11	22	33	44	55	99	66	77	88

Cid Carvalho de Souza, Cândida Nunes da Silva, Orlando Lee MC448 — Análise de Algoritmos

Particione

		p									r
A		99	33	55	77	11	22	88	66	33	44
	i	j									x
A		99	33	55	77	11	22	88	66	33	44
	i		j								x
A		99	33	55	77	11	22	88	66	33	44
		i		j							x
A		33	99	55	77	11	22	88	66	33	44
		i			j						x
A		33	99	55	77	11	22	88	66	33	44
		i				j					x
A		33	99	55	77	11	22	88	66	33	44
			i				j				x
A		33	11	55	77	99	22	88	66	33	44

Particione

		i				j				x
A	33	11	55	77	99	22	88	66	33	44
			i				j			x
A	33	11	22	77	99	55	88	66	33	44
			i					j		x
A	33	11	22	77	99	55	88	66	33	44
			i						j	x
A	33	11	22	77	99	55	88	66	33	44
				i						j
A	33	11	22	33	99	55	88	66	77	44
	p				q					r
A	33	11	22	33	44	55	88	66	77	99

Particione

Rearranja $A[p \dots r]$ de modo que $p \leq q \leq r$ e
$A[p \dots q-1] \leq A[q] < A[q+1 \dots r]$

PARTICIONE(A, p, r)
1 $x \leftarrow A[r]$ ▷ x é o "pivô"
2 $i \leftarrow p-1$
3 **para** $j \leftarrow p$ **até** $r-1$ **faça**
4   **se** $A[j] \leq x$
5   **então** $i \leftarrow i+1$
6             $A[i] \leftrightarrow A[j]$
7 $A[i+1] \leftrightarrow A[r]$
8 **devolva** $i+1$

Invariantes:
No começo de cada iteração da linha 3 vale que:
(1) $A[p \dots i] \leq x$ (2) $A[i+1 \dots j-1] > x$ (3) $A[r] = x$

Complexidade de PARTICIONE

PARTICIONE(A, p, r)	Tempo
1 $x \leftarrow A[r]$ ▷ x é o "pivô"	?
2 $i \leftarrow p-1$	?
3 **para** $j \leftarrow p$ **até** $r-1$ **faça**	?
4 **se** $A[j] \leq x$	?
5 **então** $i \leftarrow i+1$	?
6 $A[i] \leftrightarrow A[j]$	?
7 $A[i+1] \leftrightarrow A[r]$	?
8 **devolva** $i+1$	?

$T(n)$ = complexidade de tempo no pior caso sendo
$n := r - p + 1$

Complexidade de PARTICIONE

PARTICIONE(A, p, r)		Tempo
1	$x \leftarrow A[r]$ $\triangleright$ x é o "pivô"	$\Theta(1)$
2	$i \leftarrow p-1$	$\Theta(1)$
3	**para** $j \leftarrow p$ **até** $r-1$ **faça**	$\Theta(n)$
4	**se** $A[j] \leq x$	$\Theta(n)$
5	**então** $i \leftarrow i+1$	$O(n)$
6	$A[i] \leftrightarrow A[j]$	$O(n)$
7	$A[i+1] \leftrightarrow A[r]$	$\Theta(1)$
8	**devolva** $i+1$	$\Theta(1)$

$T(n) = \Theta(2n+4) + O(2n) = \Theta(n)$

Conclusão:
A complexidade de PARTICIONE é $\Theta(n)$.

QuickSort

Rearranja um vetor $A[p \dots r]$ em ordem crescente.

QUICKSORT(A, p, r)
1 **se** $p < r$
2 **então** $q \leftarrow$ PARTICIONE(A, p, r)
3 QUICKSORT$(A, p, q-1)$
4 QUICKSORT$(A, q+1, r)$

	p									r
A	99	33	55	77	11	22	88	66	33	44

QuickSort

Rearranja um vetor $A[p \dots r]$ em ordem crescente.

QUICKSORT(A, p, r)
1 **se** $p < r$
2 **então** $q \leftarrow$ PARTICIONE(A, p, r)
3 QUICKSORT$(A, p, q-1)$
4 QUICKSORT$(A, q+1, r)$

	p				q					r
A	33	11	22	33	44	55	88	66	77	99

No começo da linha 3,

$$A[p \dots q-1] \leq A[q] < A[q+1 \dots r]$$

QuickSort

Rearranja um vetor $A[p \dots r]$ em ordem crescente.

QUICKSORT(A, p, r)
1 **se** $p < r$
2 **então** $q \leftarrow$ PARTICIONE(A, p, r)
3 QUICKSORT$(A, p, q-1)$
4 QUICKSORT$(A, q+1, r)$

	p				q					r
A	11	22	33	33	44	55	88	66	77	99

QuickSort

Rearranja um vetor $A[p \dots r]$ em ordem crescente.

```
QUICKSORT(A, p, r)
se p < r
    então q ← PARTICIONE(A, p, r)
          QUICKSORT(A, p, q − 1)
          QUICKSORT(A, q + 1, r)
```

	p				q					r
A	11	22	33	33	44	55	66	77	88	99

Complexidade de QuickSort

QUICKSORT(A, p, r)	Tempo
se $p < r$	?
então $q \leftarrow$ PARTICIONE(A, p, r)	?
QUICKSORT$(A, p, q-1)$	?
QUICKSORT$(A, q+1, r)$	?

$T(n) :=$ complexidade de tempo no pior caso sendo
$n := r - p + 1$

Complexidade de QuickSort

QUICKSORT(A, p, r)	Tempo
se $p < r$	$\Theta(1)$
então $q \leftarrow$ PARTICIONE(A, p, r)	$\Theta(n)$
QUICKSORT$(A, p, q-1)$	$T(k)$
QUICKSORT$(A, q+1, r)$	$T(n-k-1)$

$T(n) = T(k) + T(n-k-1) + \Theta(n+1)$

$$0 \leq k := q - p \leq n - 1$$

Recorrência

$T(n) :=$ consumo de tempo no pior caso

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) = T(k) + T(n-k-1) + \Theta(n) \quad \text{para } n = 2, 3, 4, \dots$$

Recorrência grosseira:

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

$T(n)$ é $\Theta(???)$.

Recorrência grosseira:

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

$T(n)$ é $\Theta(n^2)$.

Recorrência cuidadosa

$T(n) :=$ complexidade de tempo no pior caso

$T(0) = \Theta(1)$
$T(1) = \Theta(1)$
$T(n) = \max_{0 \le k \le n-1} \{T(k) + T(n-k-1)\} + \Theta(n)$ para $n = 2, 3, 4, \dots$

$T(n) = \max_{0 \le k \le n-1} \{T(k) + T(n-k-1)\} + bn$

Quero mostrar que $T(n) = \Theta(n^2)$.

Demonstração – $T(n) = O(n^2)$

Vou provar que $T(n) \le cn^2 + c$ para n grande.

$$\begin{aligned}
T(n) &= \max_{0 \le k \le n-1} \left\{ T(k) + T(n-k-1) \right\} + bn \\
&\le \max_{0 \le k \le n-1} \left\{ ck^2 + c + c(n-k-1)^2 + c \right\} + bn \\
&= \max_{0 \le k \le n-1} \left\{ ck^2 + c(n-k-1)^2 \right\} + 2c + bn \\
&= c \max_{0 \le k \le n-1} \left\{ k^2 + (n-k-1)^2 \right\} + 2c + bn \\
&= c(n-1)^2 + 2c + bn \\
&= cn^2 - 2cn + c + 2c + bn \\
&\le cn^2 + c \,,
\end{aligned}$$

se $c > b/2$ e $n \ge 2c/(2c-b)$.

Continuação – $T(n) = \Omega(n^2)$

Agora vou provar que $T(n) \ge dn^2$ para n grande.

$$\begin{aligned}
T(n) &= \max_{0 \le k \le n-1} \left\{ T(k) + T(n-k-1) \right\} + bn \\
&\ge \max_{0 \le k \le n-1} \left\{ dk^2 + d(n-k-1)^2 \right\} + bn \\
&= d \max_{0 \le k \le n-1} \left\{ k^2 + (n-k-1)^2 \right\} + bn \\
&= d(n-1)^2 + bn \\
&= dn^2 - 2dn + d + bn \\
&\ge dn^2 \,,
\end{aligned}$$

se $d < b/2$ e $n \ge d/(2d-b)$.

Conclusão

$T(n)$ é $\Theta(n^2)$.

A complexidade de tempo do QUICKSORT no pior caso é $\Theta(n^2)$.

A complexidade de tempo do QUICKSORT é $O(n^2)$.

QuickSort no melhor caso

$M(n) :=$ complexidade de tempo no melhor caso

$M(0) = \Theta(1)$
$M(1) = \Theta(1)$
$M(n) = \min_{0 \leq k \leq n-1} \{M(k) + M(n-k-1)\} + \Theta(n)$ para $n = 2, 3, 4, \ldots$

Mostre que, para $n \geq 1$,

$$M(n) \geq \frac{(n-1)}{2} \lg \frac{n-1}{2}.$$

Isto implica que no melhor caso o QuickSort é $\Omega(n \lg n)$.

Que é o mesmo que dizer que o QuickSort é $\Omega(n \lg n)$.

QuickSort no melhor caso

No melhor caso k é aproximadamente $(n-1)/2$.

$$R(n) = R(\left\lfloor \frac{n-1}{2} \right\rfloor) + R(\left\lceil \frac{n-1}{2} \right\rceil) + \Theta(n)$$

Solução: $R(n)$ é $\Theta(n \lg n)$.

Humm, lembra a recorrência do MergeSort...

Mais algumas conclusões

$M(n)$ é $\Theta(n \lg n)$.

O consumo de tempo do QuickSort no melhor caso é $\Omega(n \log n)$.

Mais precisamente, a complexidade de tempo do QuickSort no melhor caso é $\Theta(n \log n)$.

Caso médio

Apesar da complexidade de tempo do QuickSort no pior caso ser $\Theta(n^2)$, na prática ele é o algoritmo mais eficiente.

Mais precisamente, a complexidade de tempo do QuickSort no caso médio é mais próximo do melhor caso do que do pior caso.

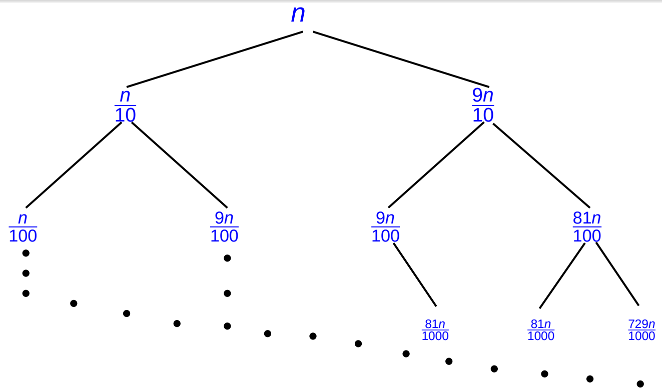
Por quê??

Suponha que (por sorte) o algoritmo Particione sempre divide o vetor na proporção $\frac{1}{10}$ para $\frac{9}{10}$. Então

$$T(n) = T(\left\lfloor \frac{n-1}{10} \right\rfloor) + T(\left\lceil \frac{9(n-1)}{10} \right\rceil) + \Theta(n)$$

Solução: $T(n)$ é $\Theta(n \lg n)$.

Árvore de recorrência



Número de níveis $\leq \log_{10/9} n$.

Em cada nível o custo é $\leq n$.

Custo total é $O(n \log n)$.

QuickSort Aleatório

O pior caso do QuickSort ocorre devido a uma escolha infeliz do pivô.

Um modo de minimizar este problema é usar aleatoriedade.

```
PARTICIONE-ALEATÓRIO(A, p, r)
    i ← RANDOM(p, r)
    A[i] ↔ A[r]
    devolva PARTICIONE(A, p, r)
```

```
QUICKSORT-ALEATÓRIO(A, p, r)
    se p < r
       então q ← PARTICIONE-ALEATÓRIO(A, p, r)
             QUICKSORT-ALEATÓRIO(A, p, q − 1)
             QUICKSORT-ALEATÓRIO(A, q + 1, r)
```


Análise do caso médio

Recorrência para o caso médio do algoritmo QuickSort-Aleatório.

$T(n) =$ consumo de tempo médio do algoritmo QuickSort-Aleatório.

Particione-Aleatório rearranja o vetor A e devolve um índice q tal que $A[p \dots q-1] \leq A[q]$ e $A[q+1 \dots r] > A[q]$.

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) = \frac{1}{n}\left(\sum_{k=0}^{n-1} (T(k) + T(n-1-k))\right) + \Theta(n).$$

$T(n)$ é $\Theta(???)$.

Análise do caso médio

$$\begin{aligned} T(n) &= \frac{1}{n}\left(\sum_{k=0}^{n-1} (T(k) + T(n-1-k))\right) + cn \\ &= \frac{2}{n}\sum_{k=0}^{n-1} T(k) + cn. \end{aligned}$$

Vou mostrar que $T(n)$ é $O(n \lg n)$.

Vou mostrar que $T(n) \leq an \lg n + b$ para $n \geq 1$ onde $a, b > 0$ são constantes.

Demonstração

$$
\begin{aligned}
T(n) &\le \frac{2}{n}\sum_{k=0}^{n-1} T(k) + cn \\
&\le \frac{2}{n}\sum_{k=0}^{n-1} (ak\lg k + b) + cn \\
&= \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + 2b + cn
\end{aligned}
$$

Lema

$$\sum_{k=1}^{n-1} k\lg k \le \frac{1}{2}n^2\lg n - \frac{1}{8}n^2.$$

Demonstração

$$
\begin{aligned}
T(n) &= \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + 2b + cn \\
&\le \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + 2b + cn \\
&= an\lg n - \frac{a}{4}n + 2b + cn \\
&= an\lg n + b + \left(cn + b - \frac{a}{4}n\right) \\
&\le an\lg n + b,
\end{aligned}
$$

escolhendo a de modo que $\frac{a}{4}n \ge cn + b$ para $n \ge 1$.

Prova do Lema

$$
\begin{aligned}
\sum_{k=1}^{n-1} k\lg k &= \sum_{k=1}^{\lceil n/2\rceil-1} k\lg k + \sum_{k=\lceil n/2\rceil}^{n-1} k\lg k \\
&\le (\lg n - 1)\sum_{k=1}^{\lceil n/2\rceil-1} k + \lg n\sum_{k=\lceil n/2\rceil}^{n-1} k \\
&= \lg n\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2\rceil-1} k \\
&\le \frac{1}{2}n(n-1)\lg n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2} \\
&\le \frac{1}{2}n^2\lg n - \frac{1}{8}n^2
\end{aligned}
$$

Conclusão

O consumo de tempo de QUICKSORT-ALEATÓRIO no caso médio é $O(n\lg n)$.

Exercício Mostre que $T(n) = \Omega(n\lg n)$.

Conclusão:

O consumo de tempo de QUICKSORT-ALEATÓRIO no caso médio é $\Theta(n\lg n)$.

O problema da ordenação - cota inferior

- Estudamos diversos algoritmos para o problema da ordenação.
- Todos eles têm algo em comum: usam somente comparações entre dois elementos do conjunto a ser ordenado para definir a posição relativa desses elementos.
- Isto é, o resultado da comparação de x_i com x_j, $i \neq j$, define se x_i será posicionado antes ou depois de x_j no conjunto ordenado.
- Todos os algoritmos dão uma cota superior para o número de comparações efetuadas por um algoritmo que resolva o problema da ordenação.
- A menor cota superior é dada pelos algoritmos MERGESORT e o HEAPSORT, que efetuam $\Theta(n \log n)$ comparações no pior caso.

O problema da ordenação - cota inferior

- Será que é possível projetar um algoritmo de ordenação baseado em comparações ainda mais eficiente?
- Veremos a seguir que não!
- É possível provar que qualquer algoritmo que ordena n elementos baseado apenas em comparações de elementos efetua **no mínimo** $\Omega(n \log n)$ comparações no pior caso.
- Para demonstrar esse fato, vamos representar os algoritmos de ordenação em um modelo computacional abstrato, denominado árvore (binária) de decisão.

Árvores de Decisão - Modelo Abstrato

- Os nós internos de uma árvore de decisão representam comparações feitas pelo algoritmo.
- As subárvores de cada nó interno representam possibilidades de continuidade das ações do algoritmo após a comparação.
- No caso das árvores binárias de decisão, cada nó possui apenas duas subárvores. Tipicamente, as duas subárvores representam os caminhos a serem seguidos conforme o resultado (verdadeiro ou falso) da comparação efetuada.
- As folhas são as respostas possíveis do algoritmo após as decisões tomadas ao longo dos caminhos da raiz até as folhas.

Árvores de decisão para o problema da ordenação

- Considere a seguinte definição alternativa do problema da ordenação:

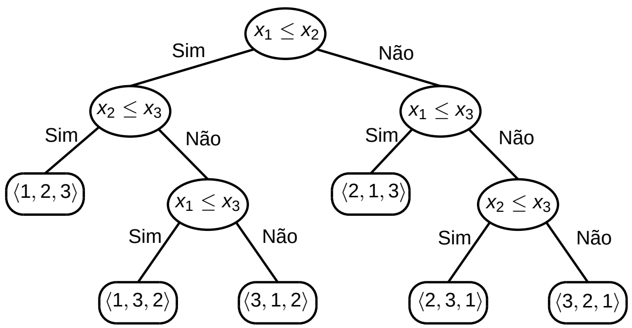
Problema da Ordenação:

Dado um conjunto de n inteiros $x_1, x_2, \ldots, x_n$, encontre uma permutação p dos índices $1 \leq i \leq n$ tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$.

- É possível representar um algoritmo para o problema da ordenação através de uma árvore de decisão da seguinte forma:
  - Os nós internos representam comparações entre dois elementos do conjunto, digamos $x_i \leq x_j$.
  - As ramificações representam os possíveis resultados da comparação: verdadeiro se $x_i \leq x_j$, ou falso se $x_i > x_j$.
  - As folhas representam possíveis soluções: as diferentes permutações dos n índices.

Árvores de Decisão para o Problema da Ordenação

Veja a árvore de decisão que representa o comportamento do *Insertion Sort* para um conjunto de 3 elementos:



Árvores de decisão para o problema da ordenação

- Ao representarmos um algoritmo de ordenação qualquer baseado em comparações por uma árvore binária de decisão, todas as permutações de n elementos devem ser possíveis soluções.
- Assim, a árvore binária de decisão deve ter pelo menos $n!$ folhas, podendo ter mais (nada impede que duas seqüências distintas de decisões terminem no mesmo resultado).
- O caminho mais longo da raiz a uma folha representa o pior caso de execução do algoritmo.
- A altura mínima de uma árvore binária de decisão com pelo menos $n!$ folhas fornece o número mínimo de comparações que o melhor algoritmo de ordenação baseado em comparações deve efetuar.

Cota inferior

- Qual a altura mínima, em função de n, de uma árvore binária de decisão com pelo menos $n!$ folhas?
- Uma árvore binária de decisão T com altura h tem, no máximo, 2^h folhas.
- Portanto, se T tem pelo menos $n!$ folhas, então $n! \leq 2^h$, ou seja, $h \geq \log_2 n!$.
- Mas,

$$\begin{aligned}
\log_2 n! &= \textstyle\sum_{i=1}^{n} \log i \\
&\geq \textstyle\sum_{i=\lceil n/2 \rceil}^{n} \log i \\
&\geq \textstyle\sum_{i=\lceil n/2 \rceil}^{n} \log n/2 \\
&\geq (n/2 - 1) \log n/2 \\
&= n/2 \log n - n/2 - \log n + 1 \\
&\geq n/4 \log n, \ \text{para } n \geq 16.
\end{aligned}$$

- Então, $h \in \Omega(n \log n)$.

Outro jeito

Devemos ter $n! \leq 2^h$, ou seja $\lg n! \leq h$.

Temos que

$$(n!)^2 = \prod_{i=0}^{n-1} (n-i)(i+1) \geq \prod_{i=1}^{n} n = n^n$$

Portanto,

$$h \geq \lg(n!) \geq \frac{1}{2} n \lg n.$$

Conclusão

- Provamos então que $\Omega(n \log n)$ é uma cota inferior para o problema da ordenação.
- Portanto, os algoritmos *Mergesort* e *Heapsort* são algoritmos ótimos.
- Veremos depois algoritmos lineares para ordenação, ou seja, que têm complexidade $O(n)$. (Como???)

Cotas inferiores de problemas

- Em geral é muito difícil provar cotas inferiores não triviais de um problema.

  Um problema com entrada de tamanho n tem como cota inferior trivial $\Omega(n)$.
- São pouquíssimos problemas para os quais se conhece uma cota inferior que coincide com a cota superior.
- Um deles é o problema da ordenação.
- Veremos mais dois exemplos: busca em um vetor ordenado e o problema de encontrar o máximo.

Busca em vetor ordenado

Dado um vetor crescente $A[p \dots r]$ e um elemento x, devolver um índice i tal que $A[i] = x$ ou -1 se tal índice não existe.

Busca-Binária(A, p, r, x)
se $p \leq r$
  então $q \leftarrow \lfloor (p+r)/2 \rfloor$
    se $A[q] > x$
      então devolva Busca-Binária$(A, p, q-1, x)$
    se $A[q] < x$
      então devolva Busca-Binária$(A, q+1, r, x)$
    devolva q $\triangleright A[q] = x$
  senão
    devolva -1

Número de comparações: $O(\lg n)$.

Busca em vetor ordenado

- É possível projetar um algoritmo mais rápido?
- Não, se o algoritmo baseia-se em comparações do tipo $A[i] < x$, $A[i] > x$ ou $A[i] = x$.
- A cota inferior do número de comparações para o problema da busca em vetor ordenado é $\Omega(\lg n)$.
- Pode-se provar isso usando o modelo de árvore de decisão.

Cota inferior

- Todo algoritmo para o problema da busca em vetor ordenado baseado em comparações pode ser representado através de uma árvore de decisão.
- Cada nó interno corresponde a uma comparação com o elemento procurado x.
- As ramificações correspondem ao resultado da comparação.
- As folhas correspondem às possíveis respostas do algoritmo. Então tal árvore deve ter pelo menos $n+1$ folhas.
- Logo, a altura da árvore é pelo menos $\Omega(\lg n)$.

Problema do Máximo

Problema:
Encontrar o maior elemento de um vetor $A[1\dots n]$.

- Existe um algoritmo que faz o serviço com $n-1$ comparações.
- Existe um algoritmo que faz menos comparações?
- Não, se o algoritmo é baseado em comparações.
- Considere um algoritmo genérico baseado em comparações que resolve o problema.
  Que "cara" ele tem?

Máximo

O algoritmo consiste, no fundo, na determinação de uma coleção $\mathcal{A}$ de pares ou *arcos* (i,j) de elementos distintos em $\{1,\dots,n\}$ tais que $A[i] < A[j]$ e existe um "sorvedouro".

Eis o paradigma de um algoritmo baseado em comparações:

```
MÁXIMO(A, n)
  𝒜 ← ∅
  enquanto 𝒜 "não possui sorvedouro" faça
      Escolha índice i e j em {1,...,n}
      se A[i] < A[j]
          então 𝒜 ← 𝒜 ∪ (i, j)
          senão 𝒜 ← 𝒜 ∪ (j, i)
  devolva 𝒜
```


Conclusão

Qualquer conjunto $\mathcal{A}$ devolvido pelo método contém uma "árvore enraizada" e portanto contém pelo menos $n-1$ arcos.

> Assim, qualquer algoritmo baseado em comparações que encontra o maior elemento de um vetor $A[1\dots n]$ faz **pelo menos** $n-1$ comparações.