MC458 — Projeto e Análise de Algoritmos I

C.C. de Souza   C.N. da Silva   O. Lee   P.J. de Rezende

1^o. Semestre de 2018

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 $A[1 \ldots j-1]$ ordenado e inserimos o elemento $A[j]$ neste subvetor.
- Repetimos o processo para $j = 2, \ldots, n$ 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
para $j \leftarrow 2$ **até** n **faça**	?
$chave \leftarrow A[j]$	?
▷ Insere $A[j]$ em $A[1 \ldots j-1]$	?
$i \leftarrow j - 1$	?
enquanto $i \geq 1$ **e** $A[i] > chave$ **faça**	?
$A[i+1] \leftarrow A[i]$	?
$i \leftarrow i - 1$	?
$A[i+1] \leftarrow chave$	?

Consumo de tempo no pior caso: ?

Complexidade de tempo de Insertion sort

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

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

SELECTION-SORT(A, n)

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

Invariantes:

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

Complexidade de Selection sort

	SELECTION-SORT(A, n)	Tempo
1	**para** $i \leftarrow 1$ **até** $n - 1$ **faça**	?
2	$min \leftarrow i$	?
3	**para** $j \leftarrow i + 1$ **até** n **faça**	?
4	**se** $A[j] < A[min]$ **então** $min \leftarrow j$	?
5	$A[i] \leftrightarrow A[min]$	?

Consumo de tempo no pior caso: ?

Complexidade de Selection sort

	SELECTION-SORT(A, n)	Tempo
1	**para** $i \leftarrow 1$ **até** $n - 1$ **faça**	$\Theta(n)$
2	$min \leftarrow i$	$\Theta(n)$
3	**para** $j \leftarrow i + 1$ **até** n **faça**	$\Theta(n^2)$
4	**se** $A[j] < A[min]$ **então** $min \leftarrow j$	$\Theta(n^2)$
5	$A[i] \leftrightarrow A[min]$	$\Theta(n)$

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

O 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 $i \leftarrow p$ **até** q **faça**
    $B[i] \leftarrow A[i]$
para $j \leftarrow q + 1$ **até** r **faça**
    $B[r + q + 1 - j] \leftarrow A[j]$
$i \leftarrow p$
$j \leftarrow r$
para $k \leftarrow p$ **até** r **faça**
    se $B[i] \leq B[j]$
        então $A[k] \leftarrow B[i]$
            $i \leftarrow i + 1$
        senão $A[k] \leftarrow B[j]$
            $j \leftarrow j - 1$

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. $A[p \ldots k-1]$ está ordenado,
2. $A[p \ldots k-1]$ contém todos os elementos de $B[p \ldots i-1]$ e de $B[j+1 \ldots r]$,
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 ← ⌊(p + r)/2⌋
           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

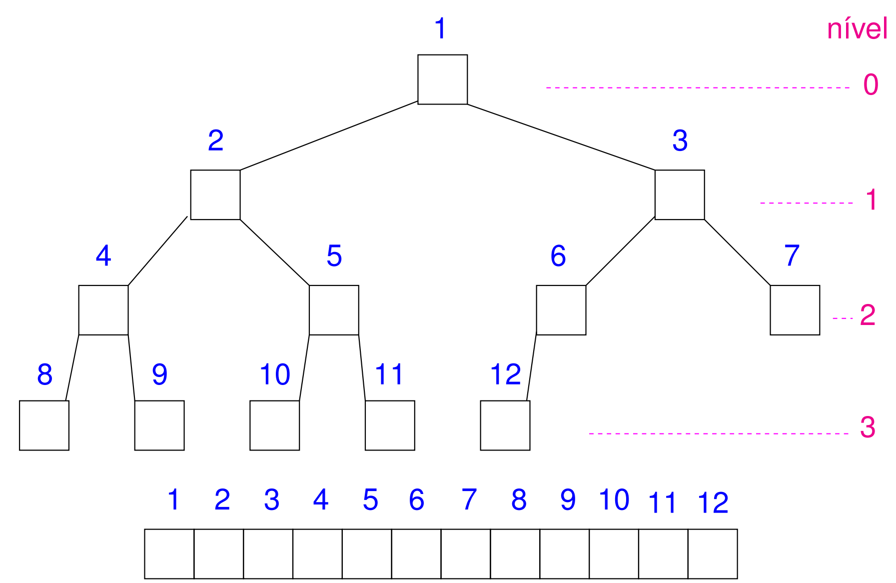
  O pior caso e o melhor caso têm a mesma complexidade.
- Complexidade de espaço/consumo espaço: $\Theta(n)$

  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

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

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,

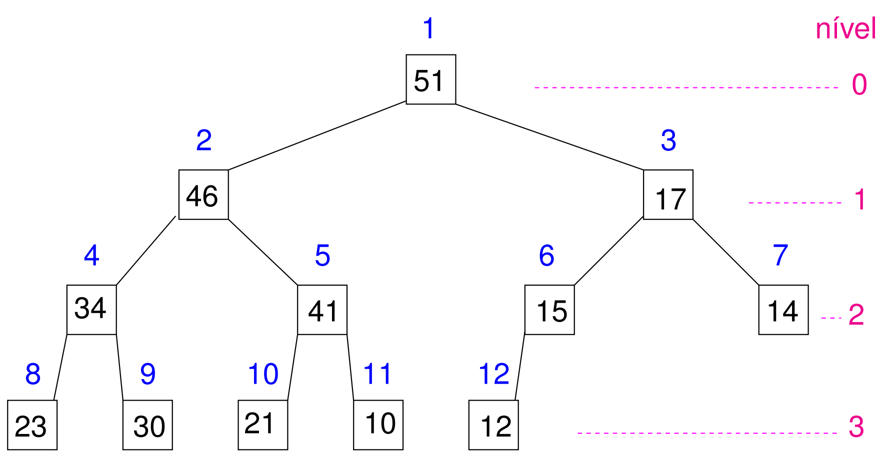
$$\begin{array}{rcccl} 2^h\,i & \leq & n & < & 2^{h+1}i \Rightarrow \\ 2^h & \leq & n/i & < & 2^{h+1} \Rightarrow \\ h & \leq & \lg(n/i) & < & h+1 \end{array}$$

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

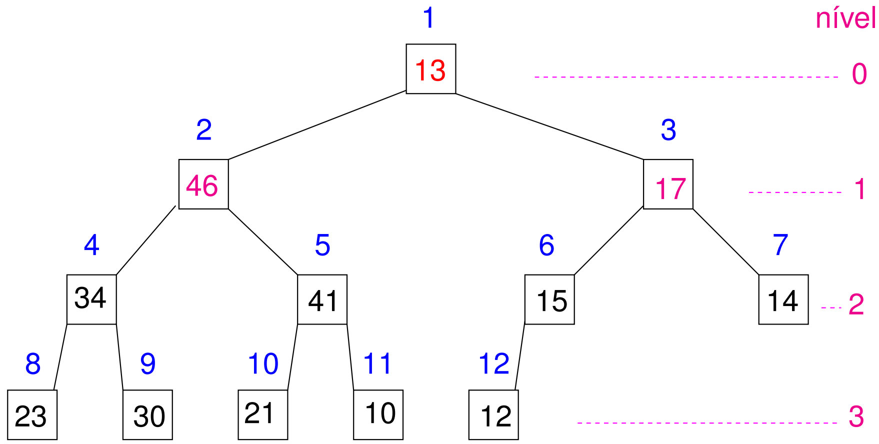


1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Min-heaps

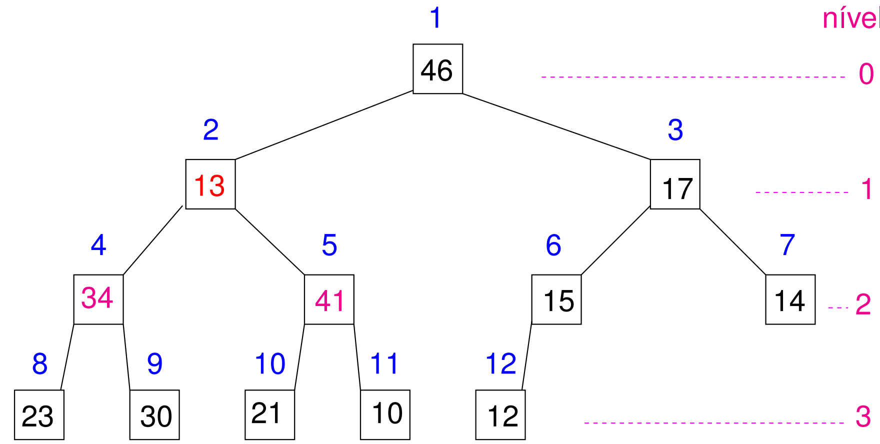
- Um nó i satisfaz a propriedade de (min-)heap se $A[\lfloor i/2 \rfloor] \leq A[i]$ (ou seja, pai $\leq$ filho).
- Uma árvore binária completa é um *min-heap* se todo nó distinto da raiz satisfaz a propriedade de min-heap.
- Vamos nos concentrar apenas em max-heaps.
- Os algoritmos que veremos podem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

Manipulação de max-heap



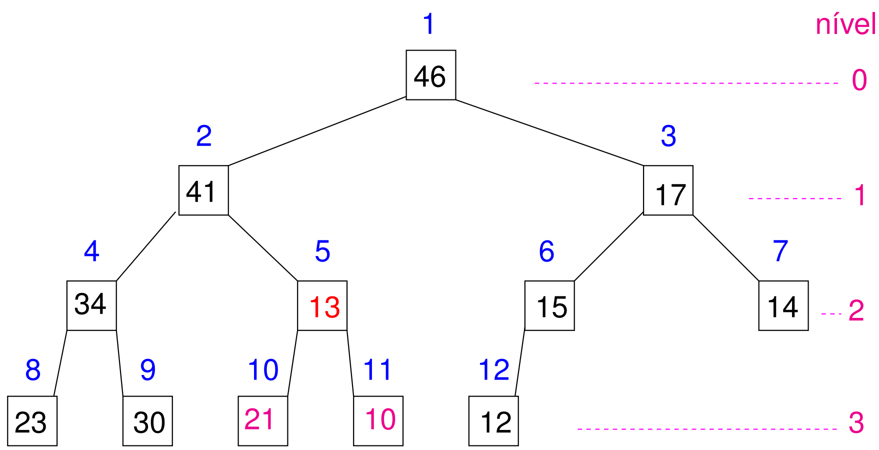
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



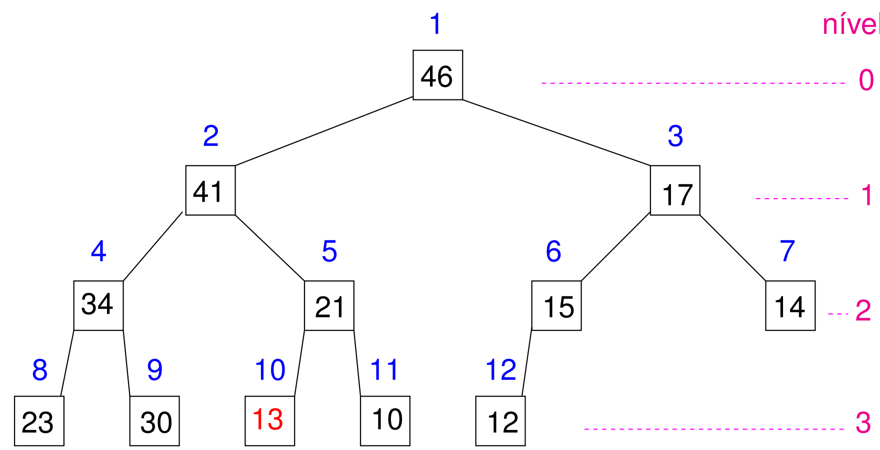
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



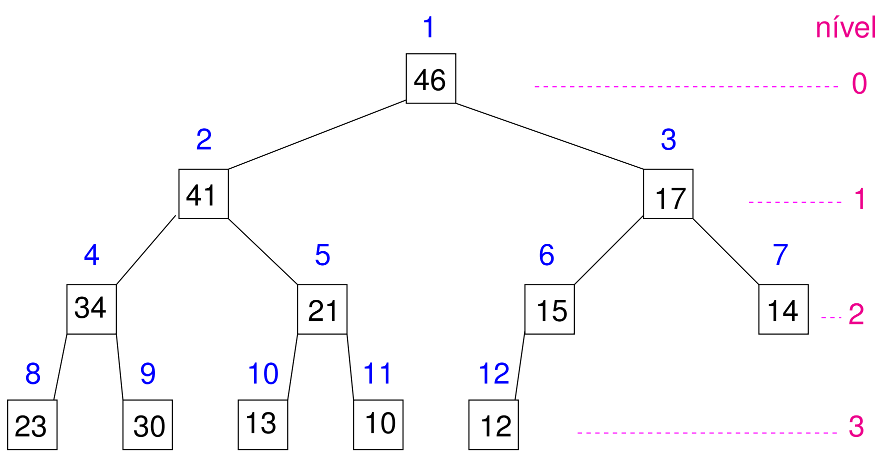
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

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 **se** $e \leq n$ **e** $A[e] > A[i]$
4 **então** $\text{maior} \leftarrow e$
5 **senão** $\text{maior} \leftarrow i$
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 $h = 1$, 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

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

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

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

Complexidade de MaxHeapify

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

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

$T(h) \leq T(h-1) + \Theta(5) + O(4)$.

Complexidade de MaxHeapify

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

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

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

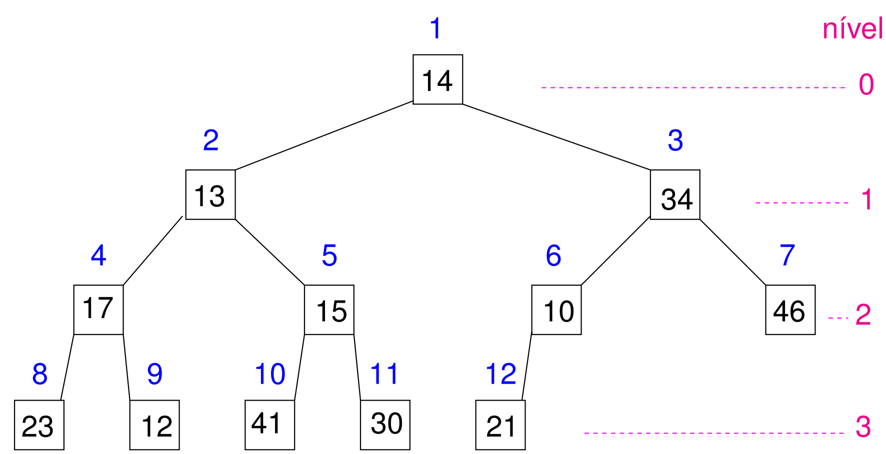
Solução assintótica: $T(n)$ é ???.

Solução assintótica: $T(n)$ é $O(h)$.

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

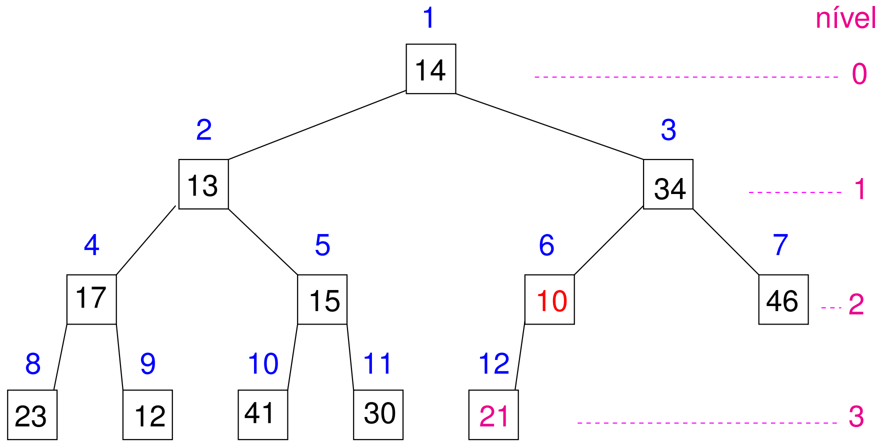
O consumo de tempo do algoritmo Max-Heapify é $O(\lg n)$ (ou melhor ainda, $O(\lg \frac{n}{i})$).

Construção de um max-heap



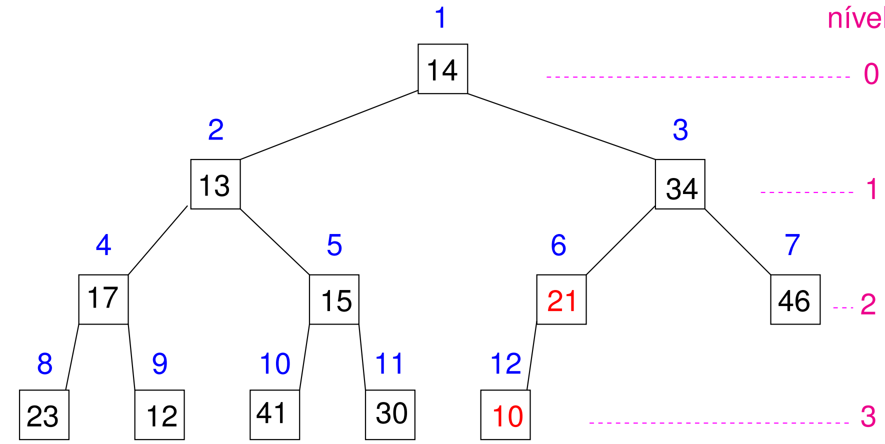
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



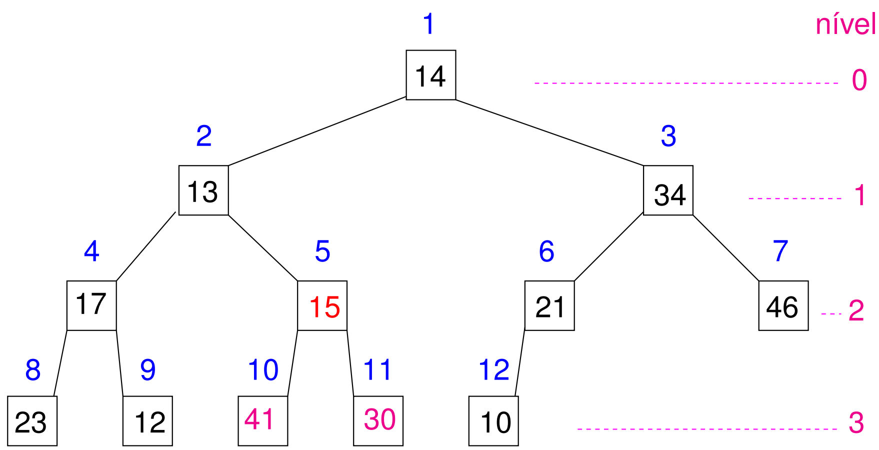
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



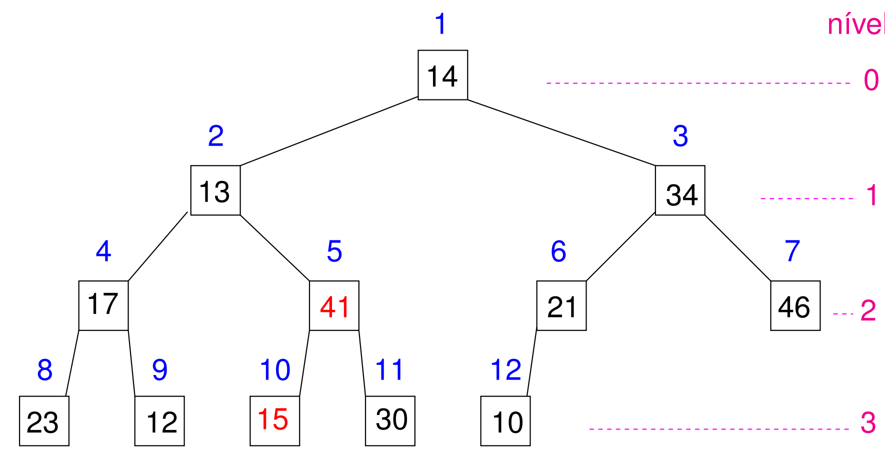
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



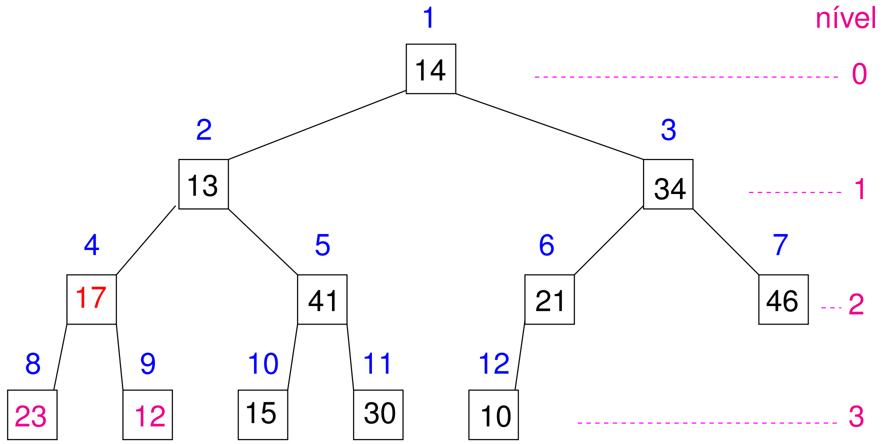
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	21	46	23	12	41	30	10

Construção de um max-heap



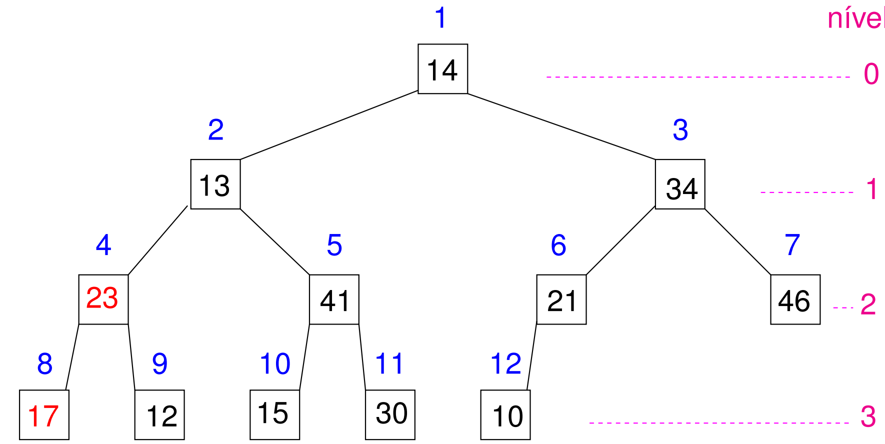
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



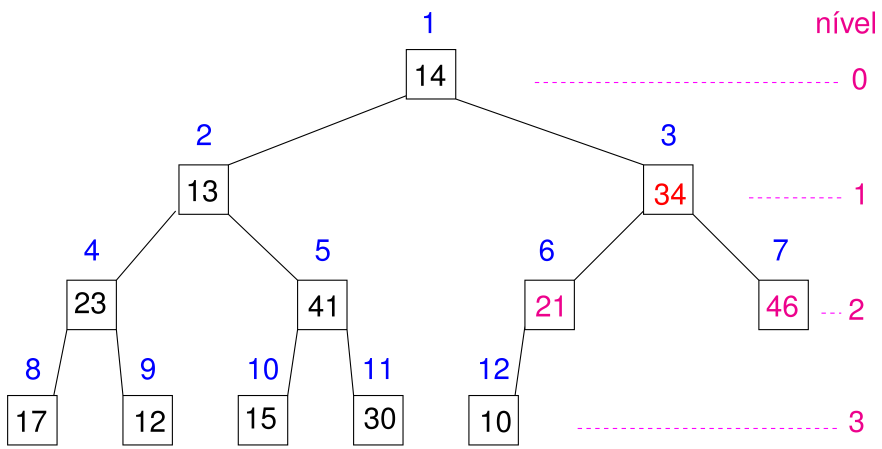
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	41	21	46	23	12	15	30	10

Construção de um max-heap



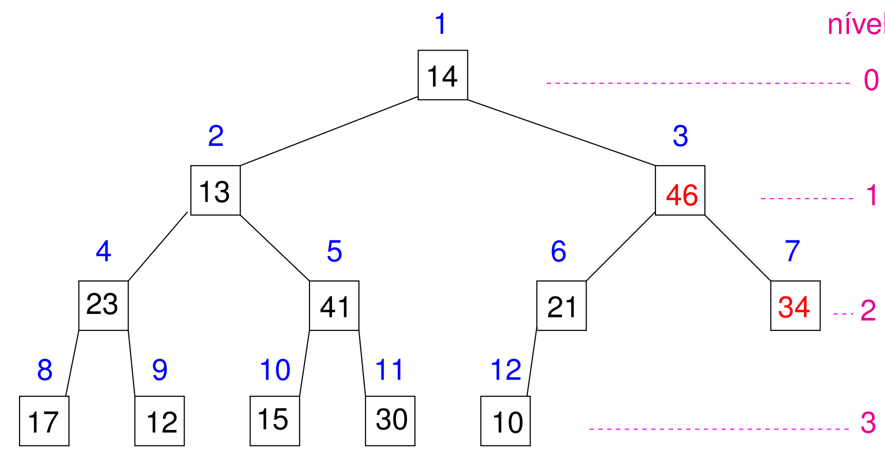
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



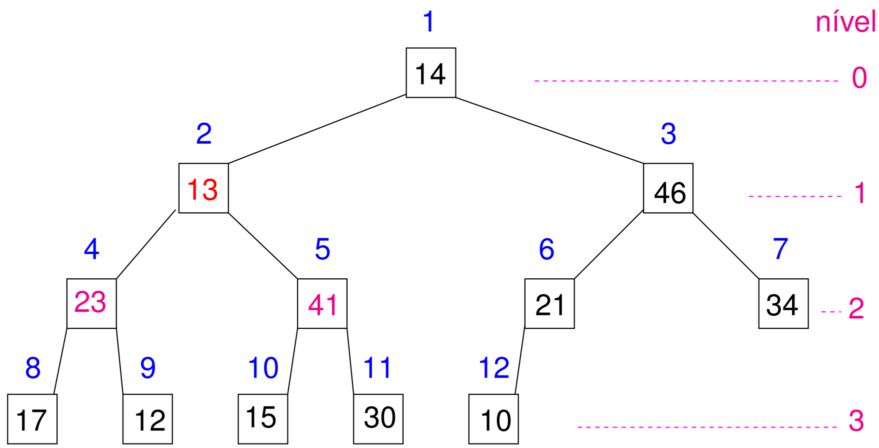
1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	23	41	21	46	17	12	15	30	10

Construção de um max-heap



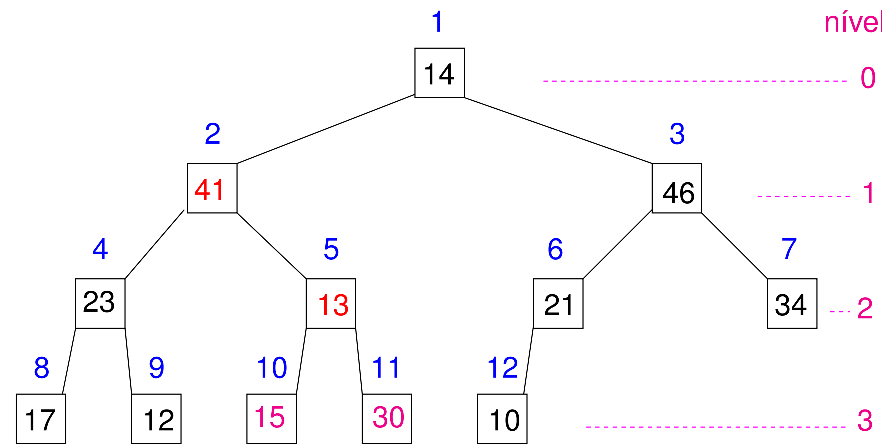
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



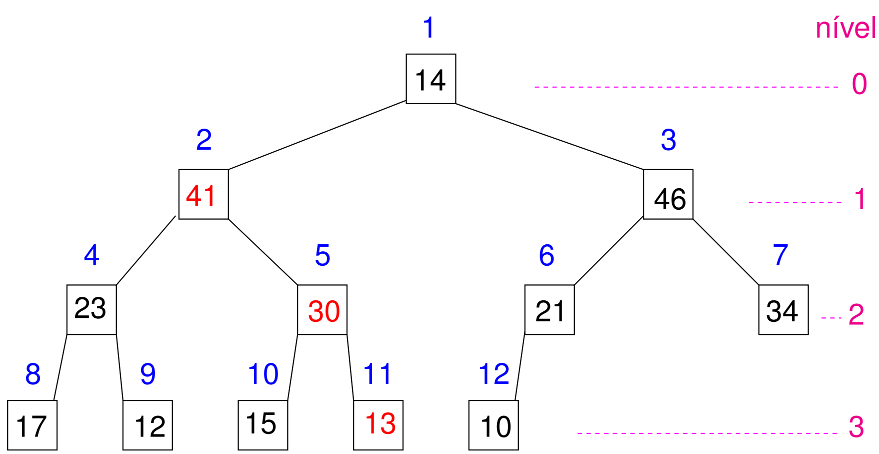
1	2	3	4	5	6	7	8	9	10	11	12
14	13	46	23	41	21	34	17	12	15	30	10

Construção de um max-heap



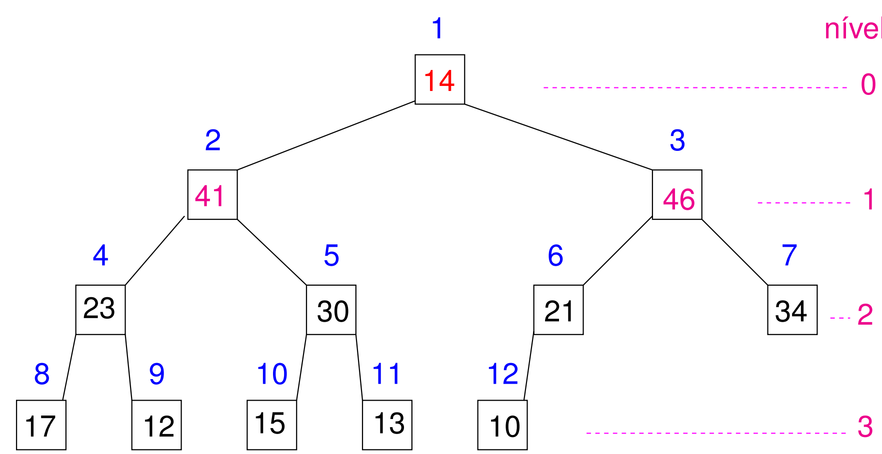
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	13	21	34	17	12	15	30	10

Construção de um max-heap



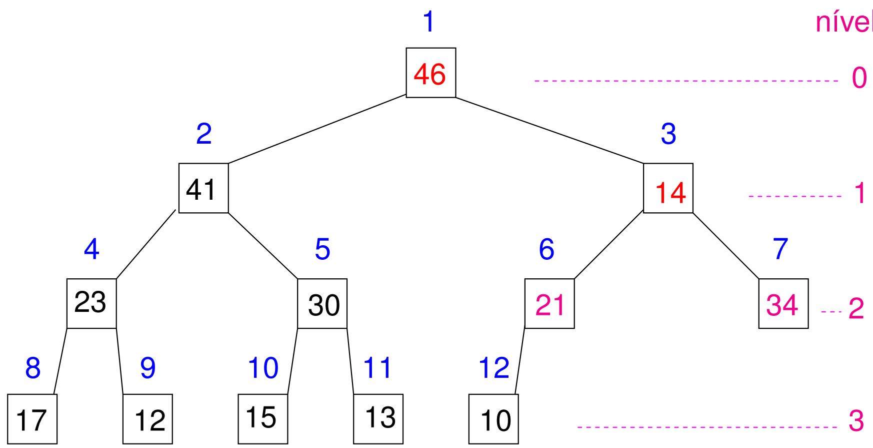
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



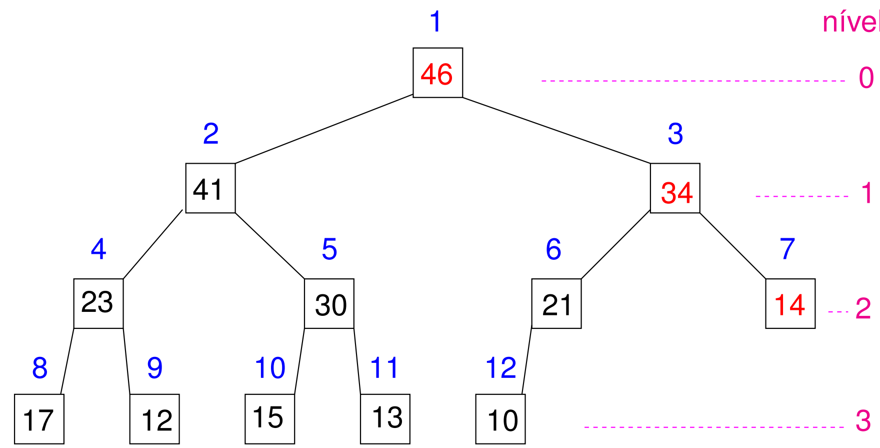
1	2	3	4	5	6	7	8	9	10	11	12
14	41	46	23	30	21	34	17	12	15	13	10

Construção de um max-heap



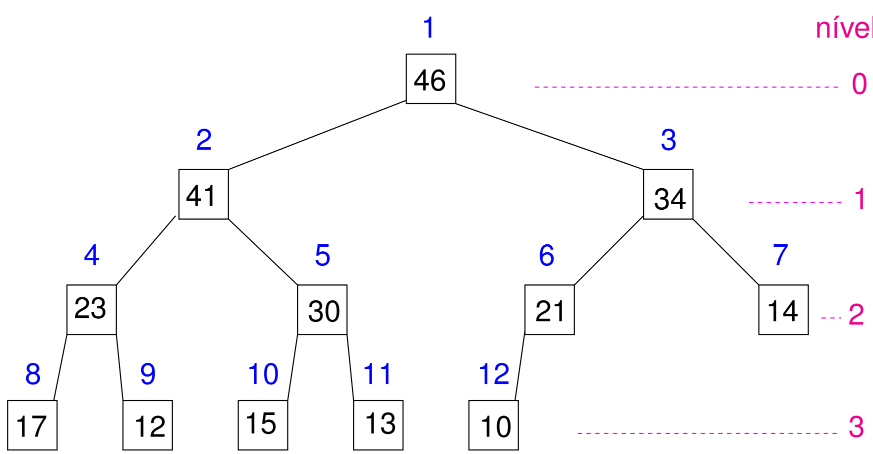
1	2	3	4	5	6	7	8	9	10	11	12
46	41	14	23	30	21	34	17	12	15	13	10

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)
1 **para** $i \leftarrow \lfloor n/2 \rfloor$ **decrescendo até** 1 **faça**
2 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 $\lfloor \lg n \rfloor + 1$.

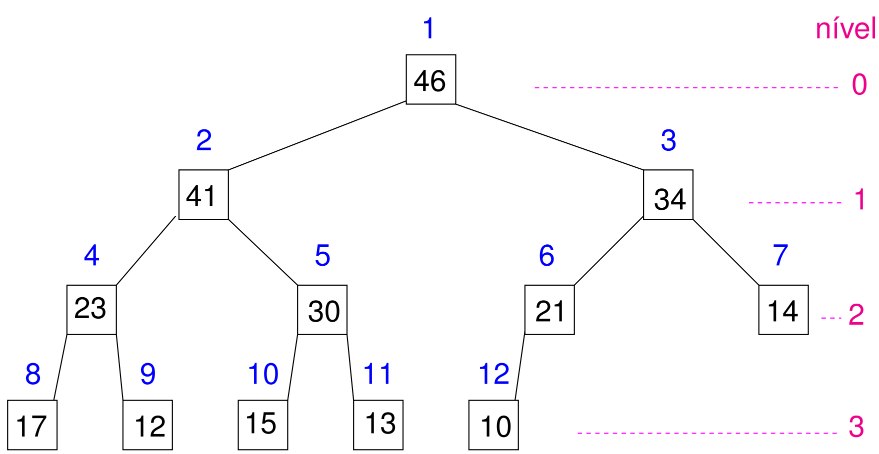
  A complexidade de BUILDMAXHEAP é $T(n) = O(S(\lg n))$.

Construção de um max-heap

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

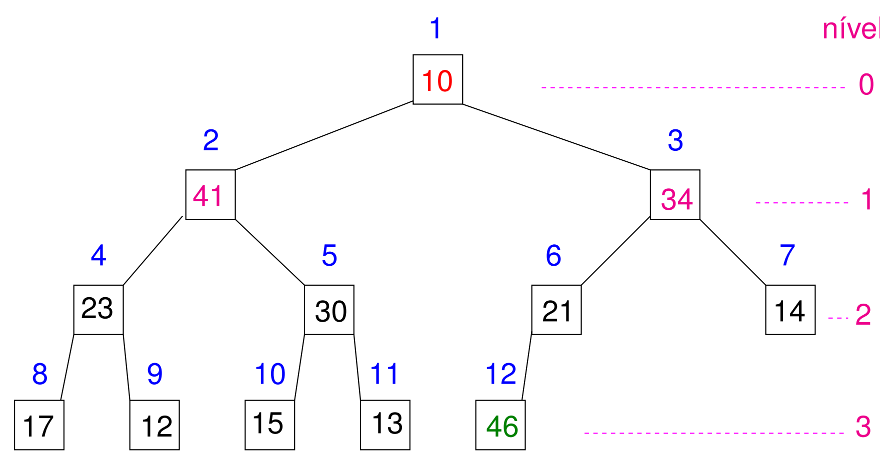
  Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)
- Veja no CLRS uma prova diferente deste fato.

HeapSort



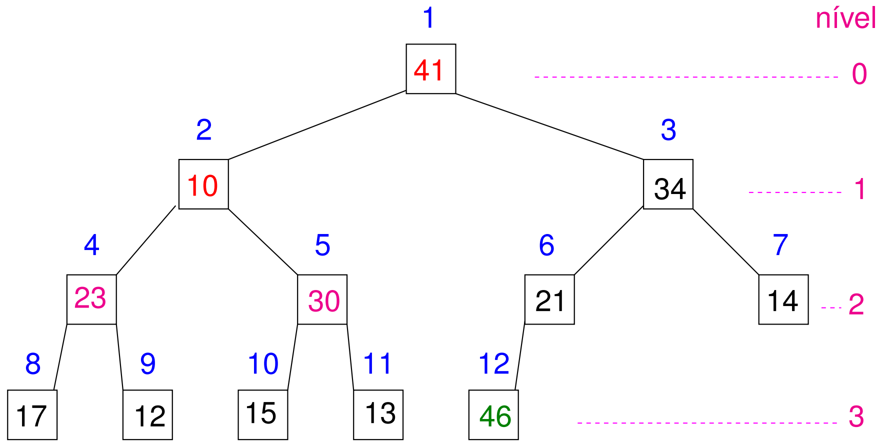
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

HeapSort



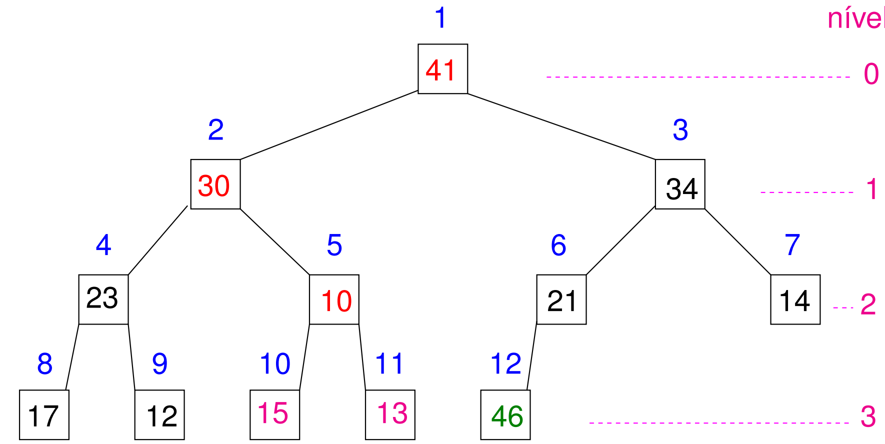
1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46

HeapSort



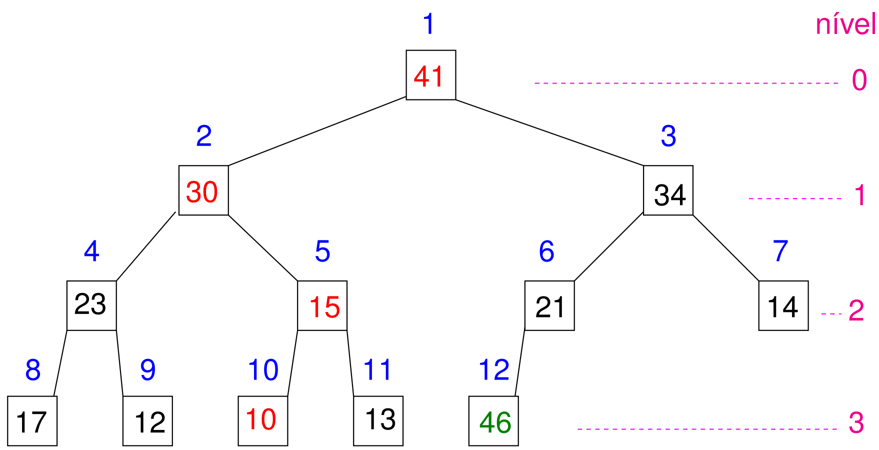
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

HeapSort



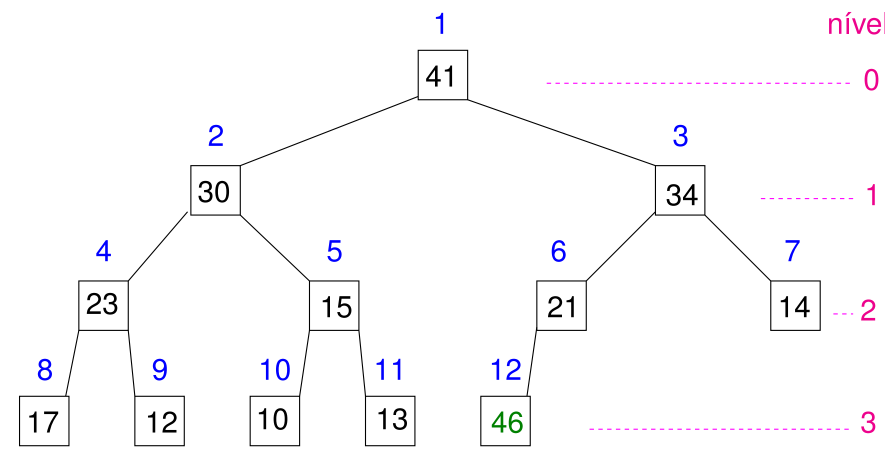
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

HeapSort



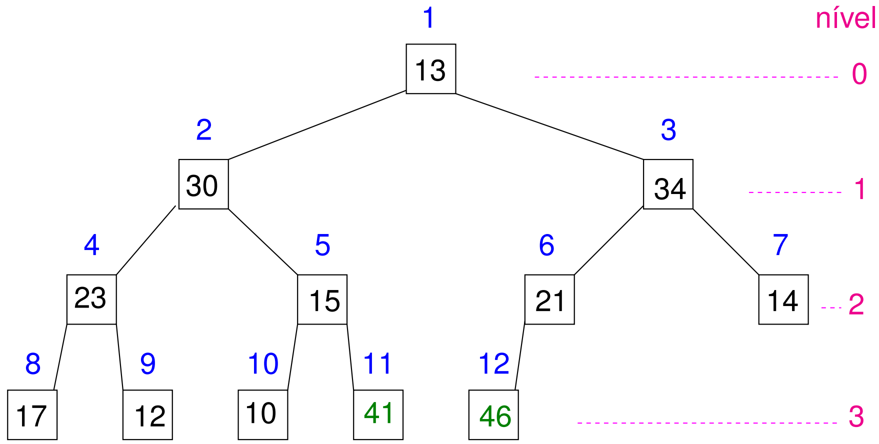
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



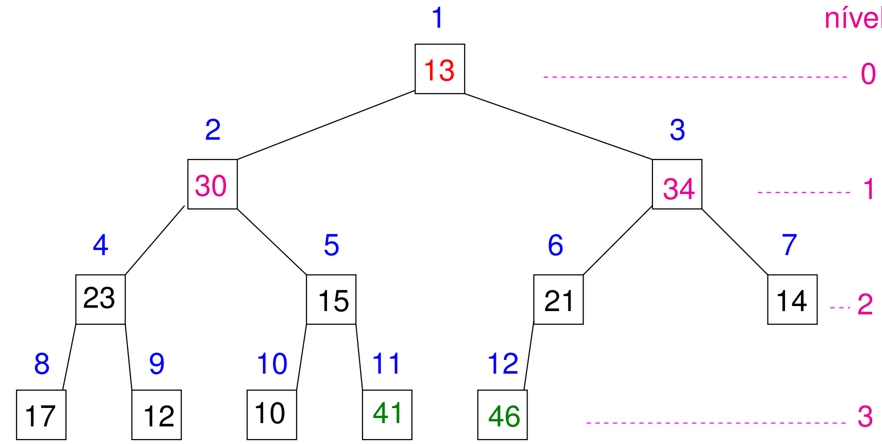
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



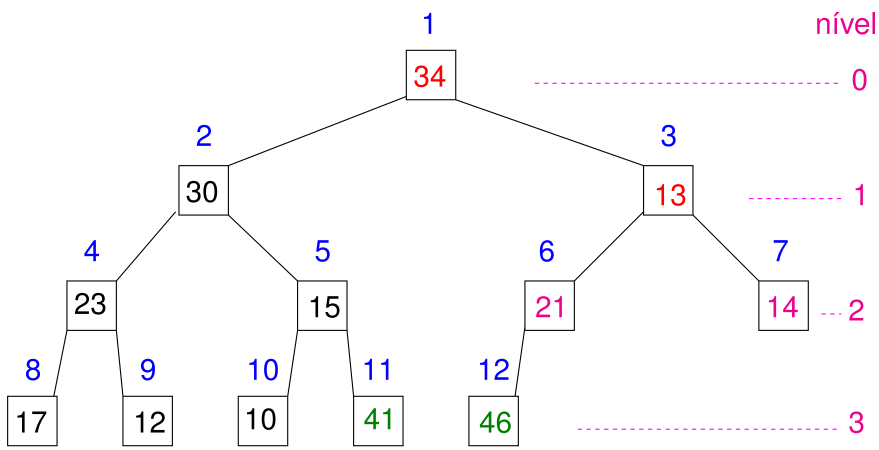
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



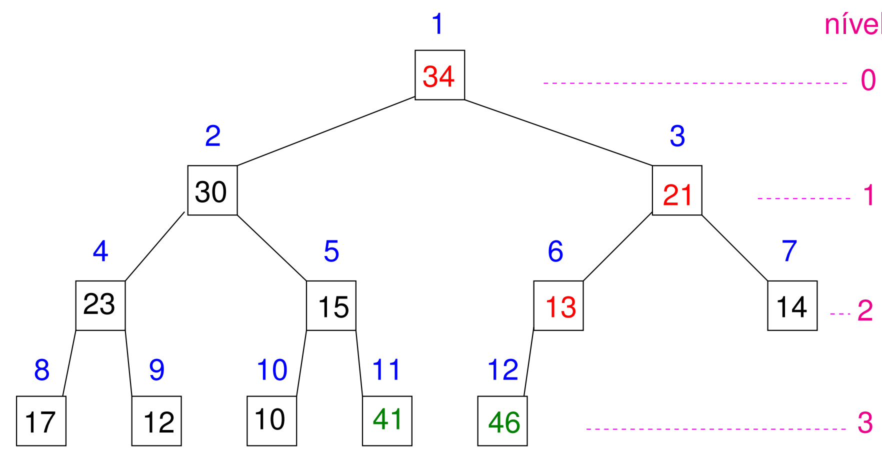
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



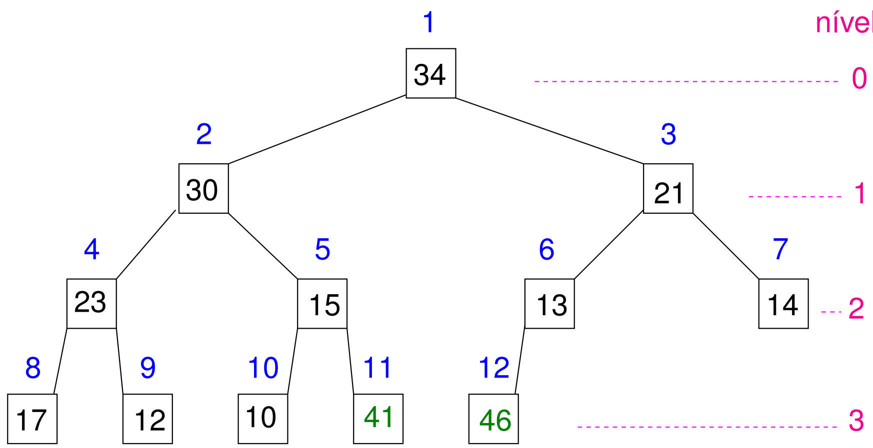
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

HeapSort



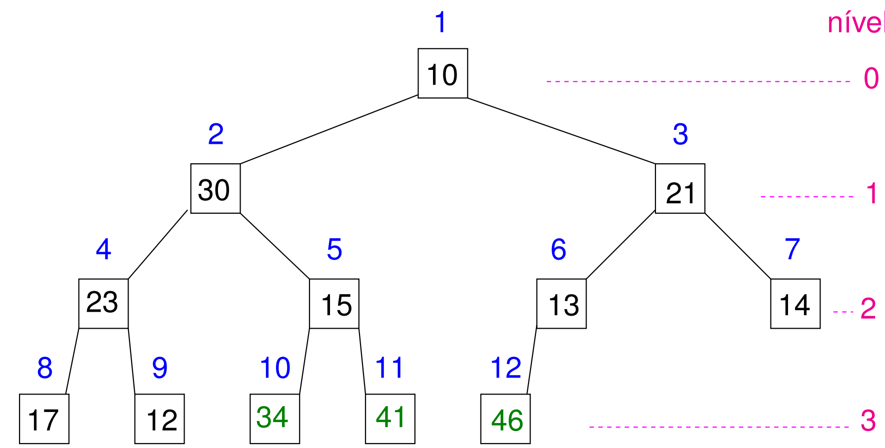
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



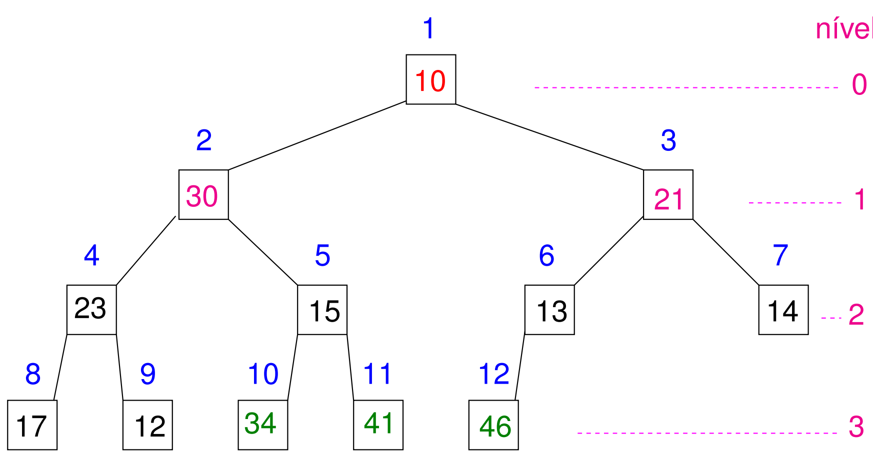
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



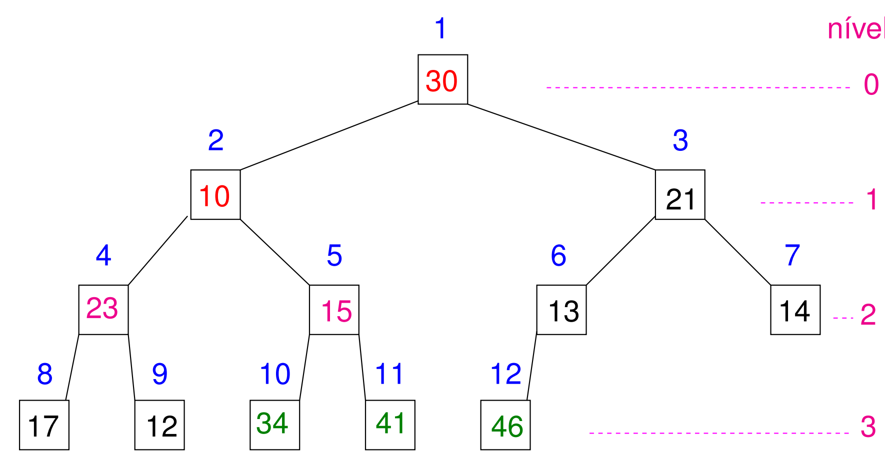
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



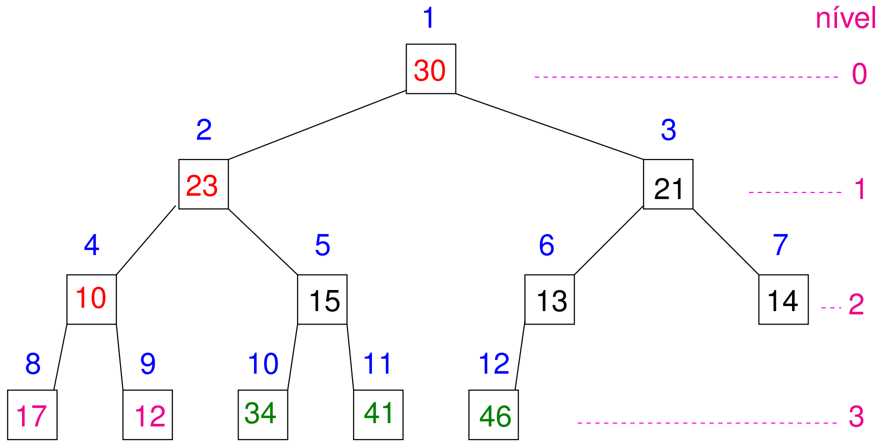
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



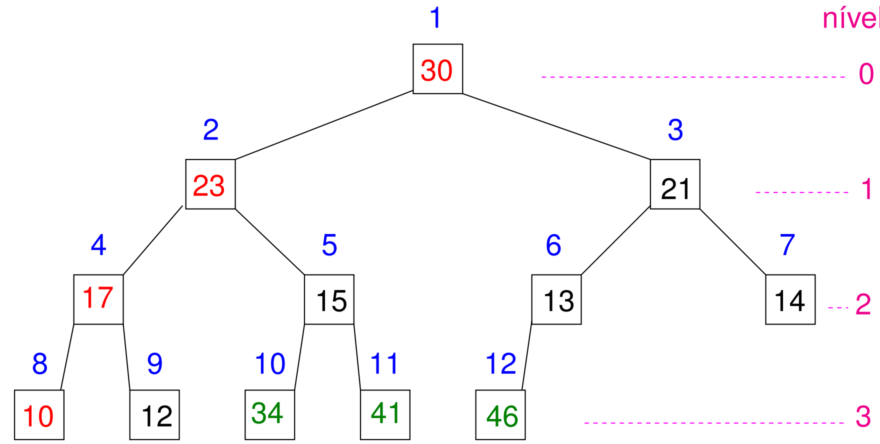
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

HeapSort



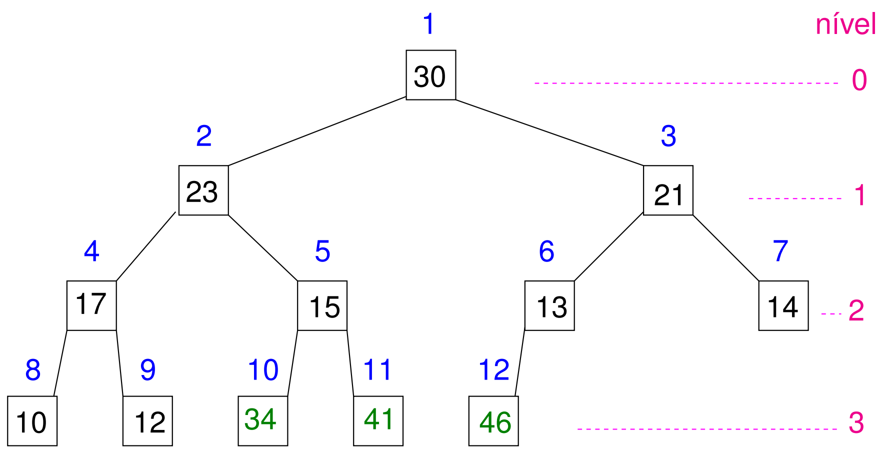
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

HeapSort



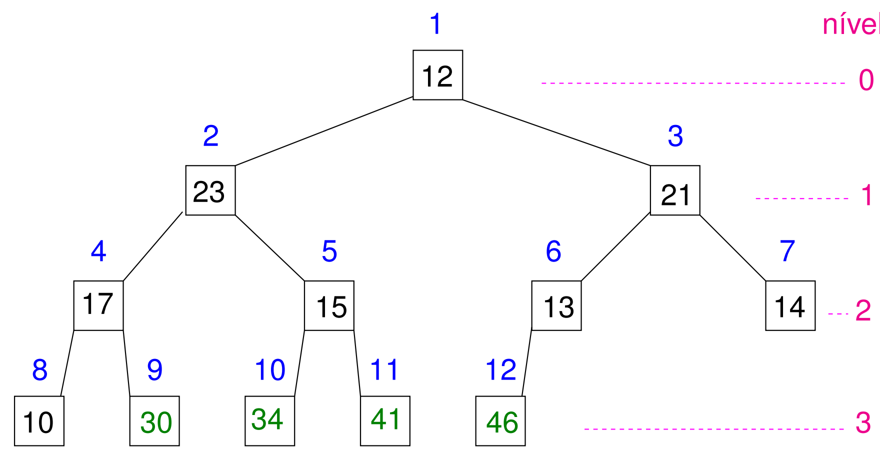
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

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 $O(n \lg n)$.

Qual a complexidade de tempo quando o vetor já está ordenado?

Qual é a complexidade de tempo de melhor caso?

Em algum caso o HeapSort é linear (como o InsertionSort)?

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 $\leftarrow A[1]$
$A[1] \leftarrow A[n]$
`cor` $\leftarrow n - 1$
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: $O(\lg n)$.

QuickSort

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

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

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

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

Conquista: ordene os dois subvetores recursivamente usando o QUICKSORT;

Combinação: nada a fazer, o vetor está ordenado.

Partição

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

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

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

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 $x \leftarrow A[r]$ $\triangleright$ 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 $x \leftarrow A[r]$ $\triangleright$ 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 $A[p \ldots r]$ 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 $A[p \ldots r]$ 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,

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

QuickSort

Rearranja um vetor $A[p \ldots r]$ 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 $n = 2, 3, 4, \ldots$

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

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

Demonstração – $T(n) = O(n^2)$

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

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

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

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

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

Mostre que, para $n \ge 1$,

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

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.

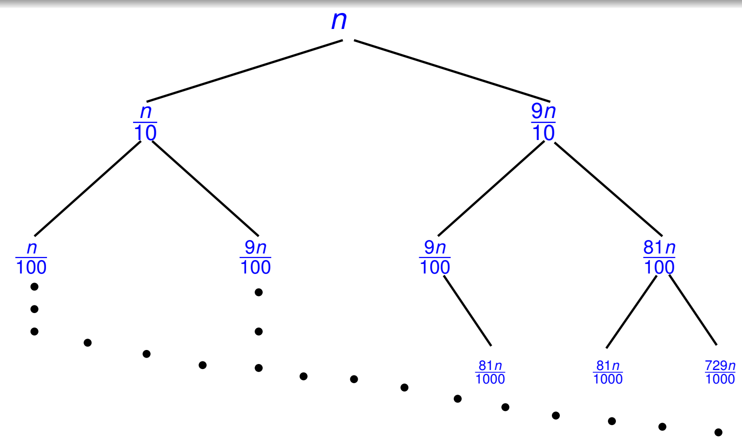
Por quê??

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)
$i \leftarrow$ RANDOM(p, r)
$A[i] \leftrightarrow A[r]$
devolva PARTICIONE(A, p, r)

QUICKSORT-ALEATÓRIO(A, p, r)
se $p < r$
  então $q \leftarrow$ 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 $A[p \ldots q-1] \leq A[q]$ e $A[q+1 \ldots r] > A[q]$.

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

$T(n)$ 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:

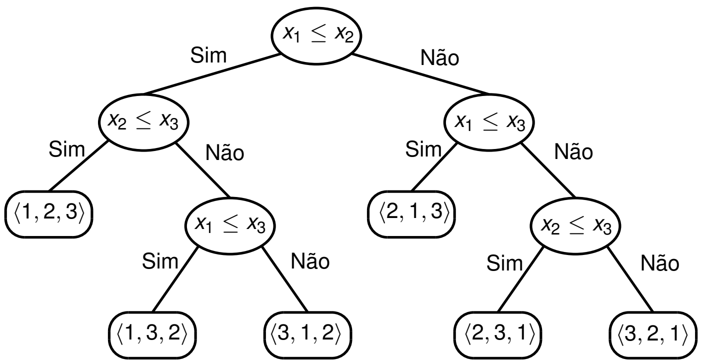
Problema da Ordenação:

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 $A[p \ldots r]$ 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)
$\mathcal{A} \leftarrow \emptyset$
enquanto $\mathcal{A}$ "não possui sorvedouro" **faça**
    Escolha índice i e j em $\{1, \dots, n\}$
    se $A[i] < A[j]$
        então $\mathcal{A} \leftarrow \mathcal{A} \cup (i, j)$
        senão $\mathcal{A} \leftarrow \mathcal{A} \cup (j, i)$
devolva $\mathcal{A}$

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.

Ordenação em Tempo Linear

Algoritmos lineares para ordenação

Os seguintes algoritmos de ordenação têm complexidade $O(n)$:

- Counting Sort: Elementos são números inteiros "pequenos"; mais precisamente, inteiros $x \in O(n)$.
- Radix Sort: Elementos são números inteiros de comprimento máximo constante, isto é, independente de n.
- Bucket Sort: Elementos são números reais uniformemente distribuídos no intervalo $[0..1)$.

Counting Sort

- Considere o problema de ordenar um vetor $A[1 \ldots n]$ de inteiros quando se sabe que todos os inteiros estão no intervalo entre 0 e k.
- Podemos ordenar o vetor simplesmente contando, para cada inteiro i no vetor, quantos elementos do vetor são menores que i.
- É exatamente o que faz o algoritmo *Counting Sort*.

Counting Sort

Counting-Sort(A, B, n, k)
para $i \leftarrow 0$ **até** k **faça**
    $C[i] \leftarrow 0$

para $j \leftarrow 1$ **até** n **faça**
    $C[A[j]] \leftarrow C[A[j]] + 1$
$\triangleright$ $C[i]$ é o número de js tais que $A[j] = i$

para $i \leftarrow 1$ **até** k **faça**
    $C[i] \leftarrow C[i] + C[i-1]$
$\triangleright$ $C[i]$ é o número de js tais que $A[j] \leq i$

para $j \leftarrow n$ **decrescendo até** 1 **faça**
    $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort - Complexidade

- Qual a complexidade do algoritmo COUNTING-SORT?
- O algoritmo não faz comparações entre elementos de A!
- Sua complexidade deve ser medida em função do número das outras operações, aritméticas, atribuições, etc.
- Claramente, a complexidade de COUNTING-SORT é $O(n+k)$. Quando $k \in O(n)$, ele tem complexidade $O(n)$.

Há algo de errado com o limite inferior de $\Omega(n \log n)$ para ordenação?

Algoritmos *in-place*

- Algoritmos de ordenação podem ser ou não *in-place*.
- Um algoritmo de ordenação é in-place se o uso de espaço (memória) adicional requerida é $O(1)$.
- Há duas formas de se computar o espaço adicional utilizado: considerando-se apenas o que é alocado explicitamente ou considerando-se as alocações implícitas.
- Nós consideraremos a segunda forma, na qual a memória utilizada para a pilha (implícita) do sistema (usada ao longo das recursões) é computada.
- Exemplos: SELECTION-SORT, INSERTION-SORT e HEAPSORT são métodos de ordenação *in-place*, já MERGESORT,QUICKSORT e COUNTING-SORT não são.

Algoritmos *estáveis*

- Algoritmos de ordenação podem ser ou não *estáveis*.
- Um método de ordenação é estável se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- Exemplos: COUNTING-SORT, INSERTION-SORT e QUICKSORT são exemplos de métodos estáveis (desde que certos cuidados sejam tomados na implementação). HEAPSORT não é.

Radix Sort

- Considere agora o problema de ordenar um vetor $A[1 \ldots n]$ inteiros quando se sabe que todos os inteiros podem ser representados com apenas d dígitos, onde d é uma constante.
- Por exemplo, os elementos de A podem ser CEPs, ou seja, inteiros de 8 dígitos.

Radix Sort

- Poderíamos ordenar os elementos do vetor dígito a dígito da seguinte forma:
  - Separamos os elementos do vetor em grupos que compartilham o mesmo dígito mais significativo.
  - Em seguida, ordenamos os elementos em cada grupo pelo mesmo método, levando em consideração apenas os $d-1$ dígitos menos significativos.
- Esse método funciona, mas requer o uso de bastante memória adicional para a organização dos grupos e subgrupos.

Radix Sort

- Podemos evitar o uso excessivo de memória adicional começando pelo dígito menos significativo.
- É isso o que faz o algoritmo Radix Sort.
- Para que Radix Sort funcione corretamente, ele deve usar um método de ordenação *estável*.
- Por exemplo, o COUNTING-SORT.

Radix Sort

Suponha que os elementos do vetor A a ser ordenado sejam números inteiros de até d dígitos. O *Radix Sort* é simplesmente:

RADIX-SORT(A, n, d)
1 **para** $i \leftarrow 1$ até d **faça**
2 Ordene $A[1 \dots n]$ pelo i-ésimo dígito usando um método estável

Radix Sort - Exemplo

329		720		720		329
457		355		329		355
657		436		436		436
839		457		839		457
436	$\rightarrow$	657	$\rightarrow$	355	$\rightarrow$	657
720		329		457		720
355		839		657		839
		$\uparrow$		$\uparrow$		$\uparrow$

Radix Sort - Corretude

O seguinte argumento indutivo garante a corretude do algoritmo:

- Hipótese de indução: os números estão ordenados com relação aos $i-1$ dígitos menos significativos.
- O que acontece ao ordenarmos pelo i-ésimo dígito?
- Se dois números têm i-ésimo dígitos distintos, o de menor i-ésimo dígito aparece antes do de maior i-ésimo dígito.
- Se ambos possuem o mesmo i-ésimo dígito, então a ordem dos dois também estará correta pois o método de ordenação é estável e, pela HI, os dois elementos já estavam ordenados segundo os $i-1$ dígitos menos significativos.

Radix Sort - Complexidade

- Qual é a complexidade de RADIX-SORT?
- Depende da complexidade do algoritmo estável usado para ordenar cada dígito.
- Se essa complexidade for $\Theta(f(n))$, obtemos uma complexidade total de $\Theta(d\ f(n))$.
- Como d é constante, a complexidade é então $\Theta(f(n))$.
- Se o algoritmo estável for, por exemplo, o COUNTING-SORT, obtemos a complexidade $\Theta(n+k)$.
- Se $k \in O(n)$, isto resulta em uma complexidade linear em n.

E o limite inferior de $\Omega(n \log n)$ para ordenação?

Radix Sort - Complexidade

- Em contraste, um algoritmo por comparação como o MERGESORT teria complexidade $\Theta(n \lg n)$.
- Assim, RADIX-SORT é mais vantajoso que MERGESORT quando $d < \lg n$, ou seja, o número de dígitos for menor que $\lg n$.
- Se n for um limite superior para o maior valor a ser ordenado, então $O(\log n)$ é uma estimativa para a quantidade de dígitos dos números.
- Isso significa que não há diferença significativa entre o desempenho do MERGESORT e do RADIX-SORT?

Radix Sort - Complexidade

- O nome *Radix Sort* vem da **base** (em inglês *radix*) em que intepretamos os dígitos.
- A vantagem de se usar RADIX-SORT fica evidente quando interpretamos os dígitos de forma mais geral que simplesmente $0..9$.
- Tomemos o seguinte exemplo: suponha que desejemos ordenar um conjunto de $n = 2^{20}$ números de 64 bits. Então, MERGESORT faria cerca de $n \lg n = 20 \times 2^{20}$ comparações e usaria um vetor auxiliar de tamanho 2^{20}.

Radix Sort - Complexidade

- Agora suponha que interpretamos cada número do como tendo $d = 4$ dígitos em base $k = 2^{16}$, e usarmos RADIX-SORT com o *Counting Sort* como método estável.

  Então a complexidade de tempo seria da ordem de $d(n + k) = 4(2^{20} + 2^{16})$ operações, bem menor que 20×2^{20} do MERGESORT. Mas, note que utilizamos dois vetores auxiliares, de tamanhos 2^{16} e 2^{20}.
- Se o uso de memória auxiliar for muito limitado, então o melhor mesmo é usar um algoritmo de ordenação por comparação *in-place*.
- Note que é possível usar o *Radix Sort* para ordenar outros tipos de elementos, como datas, palavras em ordem lexicográfica e qualquer outro tipo que possa ser visto como uma d-upla ordenada de itens comparáveis.

Bucket Sort

- Supõe que os n elementos da entrada estão distribuídos uniformemente no intervalo $[0, 1)$.
- A idéia é dividir o intervalo $[0, 1)$ em n segmentos de mesmo tamanho (*buckets*) e distribuir os n elementos nos seus respectivos segmentos. Como os elementos estão distribuídos uniformemente, espera-se que o número de elementos seja aproximadamente o mesmo em todos os segmentos.
- Em seguida, os elementos de cada segmento são ordenados por um método qualquer. Finalmente, os segmentos ordenados são concatenados em ordem crescente.

Bucket Sort - Pseudocódigo

BUCKETSORT(A, n)
para $i \leftarrow 1$ **até** n **faça**
    insira $A[i]$ na lista ligada $B[\lfloor n\,A[i] \rfloor]$
para $i \leftarrow 0$ **até** $n - 1$ **faça**
    ordene a lista $B[i]$ com INSERTION-SORT
Concatene as listas $B[0], B[1], \ldots, B[n-1]$

Bucket Sort - Exemplo

$$A = \begin{array}{r|l} 1 & .78 \\ 2 & .17 \\ 3 & .39 \\ 4 & .26 \\ 5 & .72 \\ 6 & .94 \\ 7 & .21 \\ 8 & .12 \\ 9 & .23 \\ 10 & .68 \end{array} \qquad B = \begin{array}{r|l} 0 & \\ 1 & .12, .17 \\ 2 & .21, .23, .26 \\ 3 & .39 \\ 4 & \\ 5 & \\ 6 & .68 \\ 7 & .72, .78 \\ 8 & \\ 9 & .94 \end{array}$$

Bucket Sort - Corretude

- Dois elementos x e y de A, $x < y$, ou terminam na mesma lista ou são colocados em listas diferentes $B[i]$ e $B[j]$.
- A primeira possibilidade implica que x aparecerá antes de y na concatenação final, já que cada lista é ordenada.
- No segundo caso, como $x < y$, segue que $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$. Como $i \neq j$, temos $i < j$. Assim, x aparecerá antes de y na lista final.

Bucket Sort - Complexidade

- É claro que o pior caso do *Bucket Sort* é quadrático, supondo-se que as ordenações das listas seja feita com ordenação por inserção.
- Entretanto, o tempo esperado é linear.
  Intuitivamente, a idéia da demonstração é que, como os n elementos estão distribuídos uniformemente no intervalo $[0, 1)$, então o tamanho esperado das listas é pequeno.
- Portanto, as ordenações das n listas $B[i]$ leva tempo total esperado $\Theta(n)$.
- Os detalhes podem ser vistos no livro de CLRS.

Estatísticas de Ordem

Estatísticas de Ordem (Problema da Seleção)

- Estamos interessados em resolver o

  Problema da Seleção:
  Dado um conjunto A de n números reais e um inteiro i, determinar o i-ésimo menor elemento de A.
- Casos particulares importantes:

  Mínimo : $i = 1$

  Máximo : $i = n$

  Mediana : $i = \lfloor \frac{n+1}{2} \rfloor$ (mediana inferior)

  Mediana : $i = \lceil \frac{n+1}{2} \rceil$ (mediana superior)

Mínimo

Recebe um vetor $A[1 \dots n]$ e devolve o mínimo do vetor.

MÍNIMO(A, n)
$\text{mín} \leftarrow A[1]$
para $j \leftarrow 2$ **até** n **faça**
    se $\text{mín} > A[j]$
        então $\text{mín} \leftarrow A[j]$
devolva mín

Número de comparações: $n - 1 = \Theta(n)$

Aula passada: não é possível fazer com menos comparações.

Mínimo e máximo

Recebe um vetor $A[1 \dots n]$ e devolve o mínimo e o máximo do vetor.

MINMAX(A, n)
$\text{mín} \leftarrow \text{máx} \leftarrow A[1]$
para $j \leftarrow 2$ **até** n **faça**
    se $A[j] < \text{mín}$
        então $\text{mín} \leftarrow A[j]$
    se $A[j] > \text{máx}$
        então $\text{máx} \leftarrow A[j]$
devolva $(\text{mín}, \text{máx})$

Número de comparações: $2(n-1) = 2n - 2 = \Theta(n)$

É possível fazer melhor!

Mínimo e máximo

- Processe os elementos em pares. Para cada par, compare o menor com o mínimo atual e o maior com o máximo atual. Isto resulta em 3 comparações para cada 2 elementos.
- Se n for ímpar, inicialize o mínimo e o máximo como sendo o primeiro elemento.
- Se n for par, inicialize o mínimo e o máximo comparando os dois primeiros elementos.

Número de comparações:

$3\lfloor n/2 \rfloor$	se n for ímpar
$3\lfloor n/2 \rfloor - 2$	se n for par

Pode-se mostrar que isto é o melhor possível.
(Exercício $\star$ do CLRS)

Problema da Seleção – primeira solução

Recebe $A[1 \dots n]$ e i tal que $1 \leq i \leq n$
e devolve valor do i-ésimo menor elemento de $A[1 \dots n]$

SELECT-ORD(A, n, i)
ORDENE(A, n)
devolva $A[i]$

ORDENE pode ser *MergeSort* ou *HeapSort*.

A complexidade de tempo de SELECT-ORD é $O(n \lg n)$.

Será que não dá para fazer melhor que isso?

Afinal, consigo achar o mínimo e/ou máximo em tempo $O(n)$.

Relembrando – Partição

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

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

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

Relembrando – 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)
  x ← A[r]   ▷ x é o "pivô"
  i ← p−1
  para j ← p até r − 1 faça
      se A[j] ≤ x
          então i ← i + 1
                A[i] ↔ A[j]
  A[i+1] ↔ A[r]
  devolva i + 1
```


Problema da Seleção – segunda solução

Suponha que queremos achar o i-ésimo menor de $A[1 \dots n]$.

- Executamos PARTICIONE e este rearranja o vetor e devolve um índice k tal que

$$A[1 \dots k-1] \leq A[k] < A[k+1 \dots n].$$

- Eis a idéia do algoritmo:
  - Se $i = k$, então o pivô $A[k]$ é o i-ésimo menor! (Yeesss!)
  - Se $i < k$, então o i-ésimo menor está em $A[1 \dots k-1]$;
  - Se $i > k$, então o i-ésimo menor está em $A[k+1 \dots n]$.

Problema da Seleção – segunda solução

Recebe $A[p \dots r]$ e i tal que $1 \leq i \leq r-p+1$ e devolve o i-ésimo menor elemento de $A[p \dots r]$.

```
SELECT-NL(A, p, r, i)
  se p = r
      então devolva A[p]
  q ← PARTICIONE(A, p, r)
  k ← q − p + 1
  se i = k   ▷ pivô é o i-ésimo menor!
      então devolva A[q]
      senão se i < k
          então devolva SELECT-NL(A, p, q − 1, i)
          senão devolva SELECT-NL(A, q + 1, r, i − k)
```

Segunda solução – complexidade

SELECT-NL(A, p, r, i)	Tempo
se $p = r$	?
então devolva $A[p]$	?
$q \leftarrow$ PARTICIONE(A, p, r)	?
$k \leftarrow q - p + 1$	?
se $i = k$	?
então devolva $A[q]$	?
senão se $i < k$	?
então devolva SELECT-NL$(A, p, q-1, i)$	?
senão devolva SELECT-NL$(A, q+1, r, i-k)$	?

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

Segunda solução – complexidade

SELECT-NL(A, p, r, i)	Tempo
se $p = r$	$\Theta(1)$
então devolva $A[p]$	$O(1)$
$q \leftarrow$ PARTICIONE(A, p, r)	$\Theta(n)$
$k \leftarrow q - p + 1$	$\Theta(1)$
se $i = k$	$\Theta(1)$
então devolva $A[q]$	$O(1)$
senão se $i < k$	$O(1)$
então devolva SELECT-NL$(A, p, q-1, i)$	$T(k-1)$
senão devolva SELECT-NL$(A, q+1, r, i-k)$	$T(n-k)$

$T(n) = \max\{T(k-1), T(n-k)\} + \Theta(n)$

$T(n) \in \Theta(n^2)$ (Exercício)

Segunda solução – complexidade

- A complexidade de SELECT-NL no pior caso é $\Theta(n^2)$.
- Então é melhor usar SELECT-ORD?
- Não, SELECT-NL é muito eficiente na prática.
- Vamos mostrar que no caso médio SELECT-NL tem complexidade $O(n)$.

SELECT aleatorizado

O pior caso do SELECT-NL ocorre devido a uma escolha infeliz do pivô.
Um modo de evitar isso é usar aleatoriedade (como no QUICKSORT-ALEATÓRIO).

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

Algoritmo SELECT-ALEAT

Recebe $A[p \dots r]$ e i tal que $1 \le i \le r-p+1$
e devolve o i-ésimo menor elemento de $A[p \dots r]$

SELECT-ALEAT(A, p, r, i)
se $p = r$
    então devolva $A[p]$
$q \leftarrow$ PARTICIONE-ALEATÓRIO(A, p, r)
$k \leftarrow q - p + 1$
se $i = k$ ▷ pivô é o i-ésimo menor
    então devolva $A[q]$
    senão se $i < k$
        então devolva SELECT-ALEAT$(A, p, q-1, i)$
        senão devolva SELECT-ALEAT$(A, q+1, r, i-k)$

Análise do caso médio

Recorrência para o caso médio de SELECT-ALEAT.

$T(n)$ = complexidade de tempo médio de SELECT-ALEAT.

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) \le \frac{1}{n}\sum_{k=1}^{n} T(\max\{k-1, n-k\}) + \Theta(n).$$

$T(n)$ é $\Theta(???)$.

C.C. de Souza, C.N. da Silva, O. Lee, P.J. de Rezende MC458 — Projeto e Análise de Algoritmos I v. 2.2

Análise do caso médio

$$\begin{aligned} T(n) &\le \frac{1}{n}\sum_{k=1}^{n} T(\max\{k-1, n-k\}) + an \\ &\le \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} T(k) + an \end{aligned}$$

pois

$$\max\{k-1, n-k\} = \begin{cases} k-1 & \text{se } k > \lceil n/2\rceil, \\ n-k & \text{se } k \le \lceil n/2\rceil. \end{cases}$$

Se n é par, cada termo de $T(\lceil n/2\rceil)$ a $T(n-1)$ aparece exatamente duas vezes na somatória.
Se n é ímpar, esses termos aparecem duas vezes e $T(\lfloor n/2\rfloor)$ aparece uma vez.

C.C. de Souza, C.N. da Silva, O. Lee, P.J. de Rezende MC458 — Projeto e Análise de Algoritmos I v. 2.2

Demonstração: $T(n) \le cn$

$$\begin{aligned} T(n) &\le \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} T(k) + an \\ &\overset{\text{hi}}{\le} \frac{2}{n}\sum_{k=\lfloor n/2\rfloor}^{n-1} ck + an \\ &= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2\rfloor-1} k\right) + an \\ &= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2\rfloor-1)\lfloor n/2\rfloor}{2}\right) + an \end{aligned}$$

C.C. de Souza, C.N. da Silva, O. Lee, P.J. de Rezende MC458 — Projeto e Análise de Algoritmos I v. 2.2

Demonstração: $T(n) \leq cn$

$$
\begin{aligned}
T(n) &= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2\rfloor - 1)\lfloor n/2\rfloor}{2}\right) + an \\
&\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an \\
&= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an \\
&\leq \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) \leq cn.
\end{aligned}
$$

Isto funciona se $c > 4a$ e $n \geq 2c/(c-4a)$.
Logo, $T(n) = O(n)$.

Conclusão

A complexidade de tempo de SELECT-ALEAT no caso médio é $O(n)$.

Na verdade,

A complexidade de tempo de SELECT-ALEAT no caso médio é $\Theta(n)$.

Veremos depois:
Algoritmo que resolve o Problema da Seleção em tempo linear (no pior caso).

Problema da Seleção – terceira solução

Problema da Seleção:
Dado um conjunto A de n números reais e um inteiro i, determinar o i-ésimo menor elemento de A.

Veremos um algoritmo linear para o Problema da Seleção.

BFPRT = Blum, Floyd, Pratt, Rivest e Tarjan

Para simplificar a exposição, vamos supor que os elementos em A são distintos.

Problema da Seleção – terceira solução

1. Divida os n elementos em $\left\lfloor \frac{n}{5} \right\rfloor$ subconjuntos de 5 elementos e um subconjunto de $n \bmod 5$ elementos.

$$
\begin{array}{c|cccccccccc|c}
1 & \bullet & \bullet & \dots & \bullet & \bullet & \bullet & \dots & \bullet & \bullet & \\
2 & \bullet & \bullet & \dots & \bullet & \bullet & \bullet & \dots & \bullet & \bullet & \\
 & \bullet & \bullet & \dots & \bullet & \bullet & \bullet & \dots & \bullet & \bullet & n \\
 & \bullet & \bullet & \dots & \bullet & \bullet & \bullet & \dots & \bullet & & \\
 & \bullet & \bullet & \dots & \bullet & \bullet & \bullet & \dots & \bullet & &
\end{array}
$$

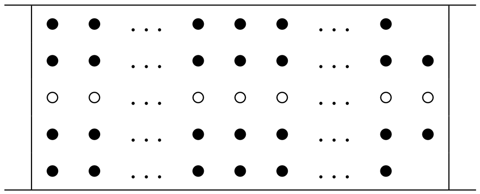
2. Encontre a mediana de cada um dos $\left\lceil \frac{n}{5} \right\rceil$ subconjuntos.

$$
\begin{array}{c|cccccccccc|c}
1 & \bullet & \bullet & \dots & \bullet & \bullet & \bullet & \dots & \bullet & & \\
2 & \bullet & \bullet & \dots & \bullet & \bullet & \bullet & \dots & \bullet & \bullet & \\
 & \circ & \circ & \dots & \circ & \circ & \circ & \dots & \circ & \circ & \\
 & \bullet & \bullet & \dots & \bullet & \bullet & \bullet & \dots & \bullet & \bullet & n \\
 & \bullet & \bullet & \dots & \bullet & \bullet & \bullet & \dots & \bullet & &
\end{array}
$$

Na figura acima, cada subconjunto está em ordem crescente, de cima para baixo.

Problema da Seleção – terceira solução

3. Determine, recursivamente, a mediana x das medianas dos subconjuntos de no máximo 5 elementos.



A figura acima é a mesma que a anterior, com as colunas ordenadas pela mediana de cada grupo. A ordem dos elementos em cada coluna permanece inalterada.
Por simplicidade de exposição, supomos que a última coluna permanece no mesmo lugar.
Note que o algoritmo não ordena as medianas!

Problema da Seleção - terceira solução

4. Usando x como pivô, particione o conjunto original A criando dois subconjuntos $A_<$ e $A_>$, onde
   - $A_<$ contém os elementos $< x$ e
   - $A_>$ contém os elementos $> x$.

   Se a posição final de x após o particionamento é k, então $|A_<| = k - 1$ e $|A_>| = n - k$.

Problema da Seleção - terceira solução

5. Finalmente, para encontrar o i-ésimo menor elemento do conjunto, compare i com a posição k de x após o particionamento:
   - Se $i = k$, x é o elemento procurado;
   - Se $i < k$, então determine recursivamente o i-ésimo menor elemento do subconjunto $A_<$;
   - Senão, determine recursivamente o $(i - k)$-ésimo menor elemento do subconjunto $A_>$.

Note que esta parte é idêntica ao feito em SELECT-NL e em SELECT-ALEAT. O que diferencia este algoritmo dos outros é a escolha do pivô. Escolhendo-se a mediana das medianas, vamos poder garantir que nenhum dos lados é muito "grande".

Terceira solução – complexidade

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

1. Divisão em subconjuntos de 5 elementos. $\Theta(n)$
2. Encontrar a mediana de cada subconjunto. $\Theta(n)$
3. Encontrar x, a mediana das medianas. $T(\lceil n/5 \rceil)$
4. Particionamento com pivô x. $O(n)$
5. Encontrar o i-ésimo menor de $A_<$ $T(k-1)$
   OU encontrar o $i - k$-ésimo menor de $A_>$. $T(n-k)$

Temos então a recorrência
$T(n) = T(\lceil n/5 \rceil) + T(\max\{k - 1, n - k\}) + \Theta(n)$

Terceira Solução - Complexidade

O diagrama abaixo classifica os elementos da última figura.

$$
\begin{array}{|cccccccccc|ccc}
\hline
\square & \square & \dots & \square & \square & \bullet & \dots & \bullet & & & & & \\
\square & \square & \dots & \square & \square & \bullet & \dots & \bullet & \bullet & & \square & = & < x \\
\square & \square & \dots & \square & x & \triangle & \dots & \triangle & \triangle & & \triangle & = & > x \\
\bullet & \bullet & \dots & \bullet & \triangle & \triangle & \dots & \triangle & \triangle & & \bullet & = & ? \\
\bullet & \bullet & \dots & \bullet & \triangle & \triangle & \dots & \triangle & & & & & \\
\hline
\end{array}
$$

Veja que o número de elementos $> x$, isto é $\triangle$s, é no mínimo $\frac{3n}{10} - 6$.

Isto porque no mínimo $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$ grupos contribuem com 3 elementos $> x$, exceto possivelmente o último e aquele que contém x. Portanto, $3\left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2\right) \geq \frac{3n}{10} - 6$.

Terceira Solução - Complexidade

Da mesma forma, o número de elementos $< x$, isto é $\square$s, é no mínimo $\frac{3n}{10} - 6$.
Assim, no passo 5 do algoritmo,

$$\max\{k-1, n-k\} \leq n - \left(\frac{3n}{10} - 6\right) \leq \frac{7n}{10} + 6.$$

A recorrência $T(n)$ está agora completa:

$$T(n) \leq \begin{cases} \Theta(1), & n \leq 140 \\ T(\lceil n/5 \rceil) + T(\lfloor 7n/10 \rfloor + 6) + \Theta(n), & n > 140, \end{cases}$$

140 é um "número mágico" que faz as contas funcionarem...

A solução é $T(n) \in \Theta(n)$

Solução da recorrência: $T(n) \leq cn$

$$
\begin{aligned}
T(n) &\leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 \rfloor + 6) + an \\
&\overset{\text{hi}}{\leq} c\lceil n/5 \rceil + c(\lfloor 7n/10 \rfloor + 6) + an \\
&\leq c(n/5 + 1) + c(7n/10 + 6) + an \\
&= 9cn/10 + 7c + an \\
&= cn + (-cn/10 + 7c + an) \\
&\leq cn,
\end{aligned}
$$

Quero que $(-cn/10 + 7c + an) \leq 0$.

Isto equivale a $c \geq 10a(n/(n-70))$ quando $n > 70$. Como $n > 140$, temos $n/(n-70) \leq 2$ e assim basta escolher $c \geq 20a$.

Algoritmo SELECT

Recebe $A[p \dots r]$ e i tal que $1 \leq i \leq r-p+1$ e devolve um índice q tal que $A[q]$ é o i-ésimo menor elemento de $A[p \dots r]$.

SELECT(A, p, r, i)
1 **se** $p = r$
2   **então devolva** p $\triangleright$ p e não $A[p]$
3 $q \leftarrow$ PARTICIONE-BFPRT(A, p, r)
4 $k \leftarrow q - p + 1$
5 **se** $i = k$
6   **então devolva** q $\triangleright$ q e não $A[q]$
7   **senão se** $i < k$
8   **então devolva** SELECT($A, p, q-1, i$)
9   **senão devolva** SELECT($A, q+1, r, i-k$)

Particione-BFPRT

Rearranja $A[p \dots r]$ e devolve um índice q, $p \leq q \leq r$, tal que $A[p \dots q-1] \leq A[q] < A[q+1 \dots r]$ e

$$\max\{k-1, n-k\} \leq \left\lceil \frac{7n}{10} \right\rceil + 6,$$

onde $n = r - p + 1$ e $k = q - p + 1$.

Particione-BFPRT

- Divida o vetor em $\lfloor n/5 \rfloor$ grupos de tamanho 5 e um grupo ≤ 5,
- ordene cada grupo e determine a mediana de cada um deles,
- determine a mediana das medianas chamando Select (!!)
- e particione o vetor em torno desse valor.

Particione-BFPRT

Particione-BFPRT(A, p, r) $\triangleright\ n := r - p + 1$

para $j \leftarrow p, p{+}5, p{+}5 \cdot 2, \dots$ **até** $p{+}5(\lceil n/5 \rceil - 1)$ **faça**
    Ordene$(A, j, j{+}4)$
Ordene$(A, p{+}5\lfloor n/5 \rfloor, n)$

para $j \leftarrow 1$ **até** $\lceil n/5 \rceil - 1$ **faça**
    $A[j] \leftrightarrow A[p{+}5j{-}3]$
$A[\lceil n/5 \rceil] \leftrightarrow A[\lfloor (p{+}5\lfloor n/5 \rfloor{+}n)/2 \rfloor]$

$k \leftarrow$ Select$(A, p, p{+}\lceil n/5 \rceil{-}1, \lfloor (\lceil n/5 \rceil{+}1)/2 \rfloor)$

$A[k] \leftrightarrow A[r]$
devolva Particione(A, p, r)

Exercícios

Exercício 1 Mostre como modificar QuickSort de modo que tenha complexidade de tempo $\Theta(n \lg n)$ no pior caso.

Exercício 2 Suponha que você tenha uma subrotina do tipo “caixa-preta” que determina a mediana em tempo linear (no pior caso). Descreva um algoritmo linear simples que resolve o problema da seleção para todo i.

Exercício 3 Dado um conjunto de n números, queremos imprimir em ordem crescente os i maiores elementos deste usando um algoritmo baseado em comparações.
Compare a complexidade dos seguinte métodos em função de n e i.

- Ordene o vetor e liste os i maiores elementos.
- Construa uma fila de prioridade (max-heap) e chame a rotina EXTRACT-MAX i vezes.
- Use um algoritmo de seleção para encontrar o i-ésimo maior elemento, particione o vetor em torno dele e ordene os i maiores elementos.