Projeto e Análise de Algoritmos

A. G. Silva e R. de Santiago

Baseado nos materiais de
Souza, Silva, Lee, Rezende, Miyazawa – Unicamp

26 de abril de 2019

Conteúdo programático

- Introdução (4 horas/aula)
- Notação Assintótica e Crescimento de Funções (4 horas/aula)
- Recorrências (4 horas/aula)
- Divisão e Conquista (12 horas/aula)
- Buscas (4 horas/aula)
- Grafos (4 horas/aula)
- Algoritmos Gulosos (8 horas aula)
- Programação Dinâmica (8 horas/aula)
- NP-Completude e Reduções (6 horas/aula)
- Algoritmos Aproximados e Busca Heurística (6 horas/aula)

Cronograma

- **15mar** – Apresentação da disciplina. Introdução.
- **22mar** – *Prova de proficiência/validação.*
- **29mar** – Notação assintótica. Recorrências.
- **05abr** – Recorrências. Divisão e conquista.
- **12abr** – Ordenação. Multiplicação de inteiros.
- **19abr** – *Dia não letivo.* Exercícios.
- **26abr** – Ordenação. Estatística de ordem.
- **03mai** – **Primeira avaliação.**
- **10mai** – Grafos. Buscas.
- **17mai** – Algoritmos gulosos.
- **24mai** – Algoritmos gulosos. Programação dinâmica.
- **31mai** – Programação dinâmica.
- **07jun** – **Segunda avaliação.**
- **14jun** – *Semana Acadêmica.* NP-Completude e reduções.
- **21jun** – *Dia não letivo.* Exercícios.
- **28jun** – **Avaliação substitutiva** *(opcional)*

Ordenação – outros métodos importantes

Algoritmos de ordenação

Algoritmos de ordenação:

- Insertion sort $\checkmark$
- Selection sort $\checkmark$
- Mergesort $\checkmark$
- Quicksort $\checkmark$
- Heapsort

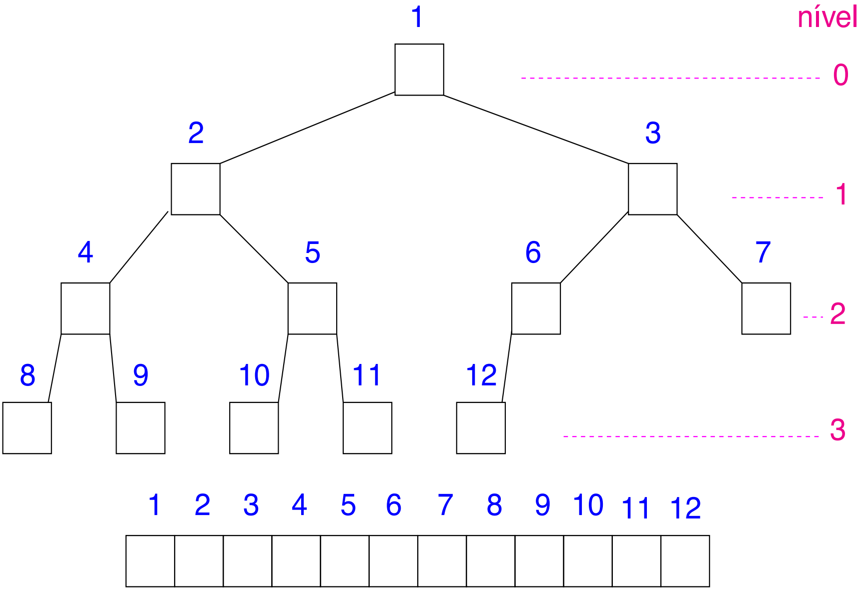
Algoritmos lineares:

- Counting sort
- Radix sort

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 $\lfloor n/2 \rfloor + 1, \ldots, n - 1, n$.

Níveis

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

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

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

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

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,

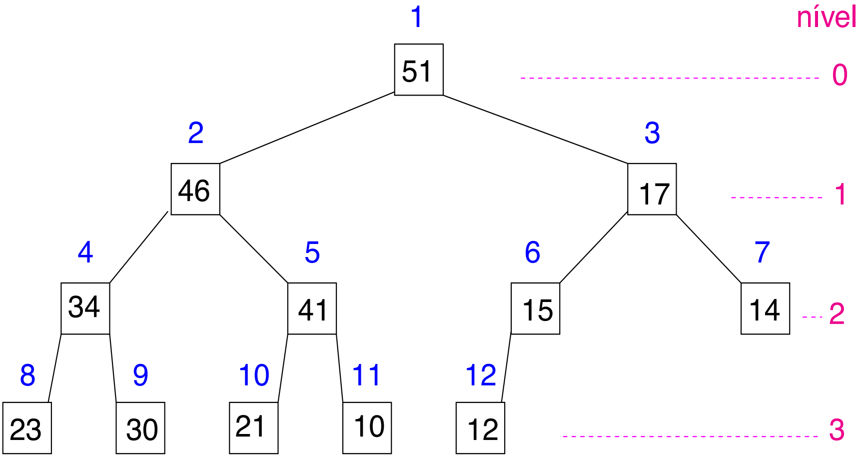
$$\begin{array}{rcccll} 2^h\,i & \leq & n & < & 2^{h+1}\,i & \Rightarrow \\ 2^h & \leq & n/i & < & 2^{h+1} & \Rightarrow \\ h & \leq & \lg(n/i) & < & h+1 & \end{array}$$

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

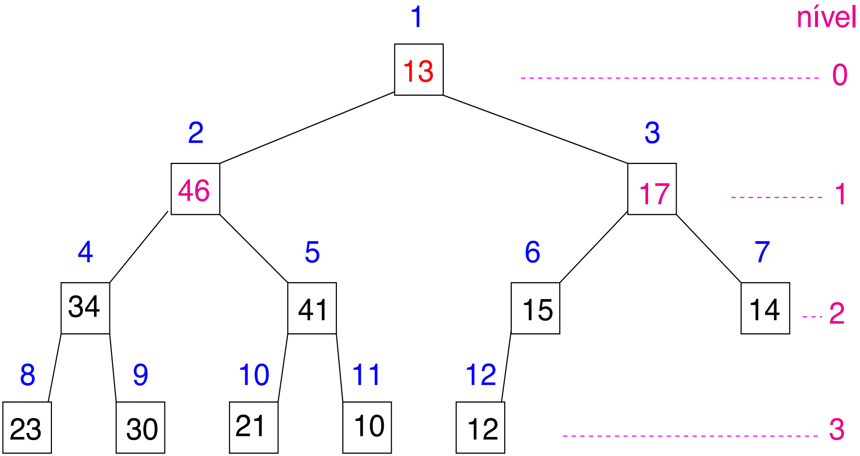


1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Min-heaps

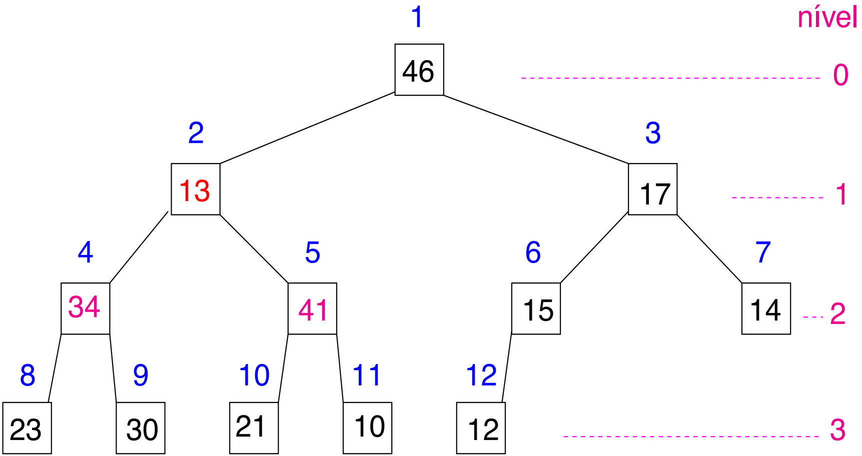
- Um nó i satisfaz a propriedade de (min-)heap se $A[\lfloor i/2 \rfloor] \leq A[i]$ (ou seja, pai $\leq$ filho).
- Uma árvore binária completa é um *min-heap* se todo nó distinto da raiz satisfaz a propriedade de min-heap.
- Vamos nos concentrar apenas em max-heaps.
- Os algoritmos que veremos podem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

Manipulação de max-heap



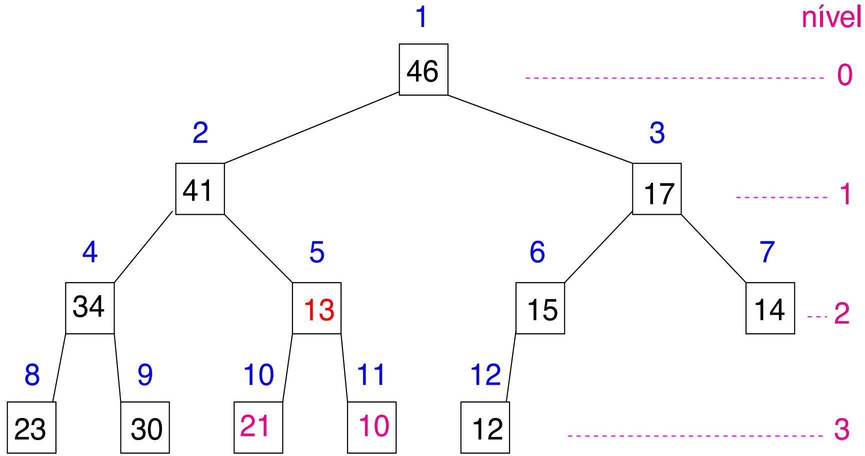
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



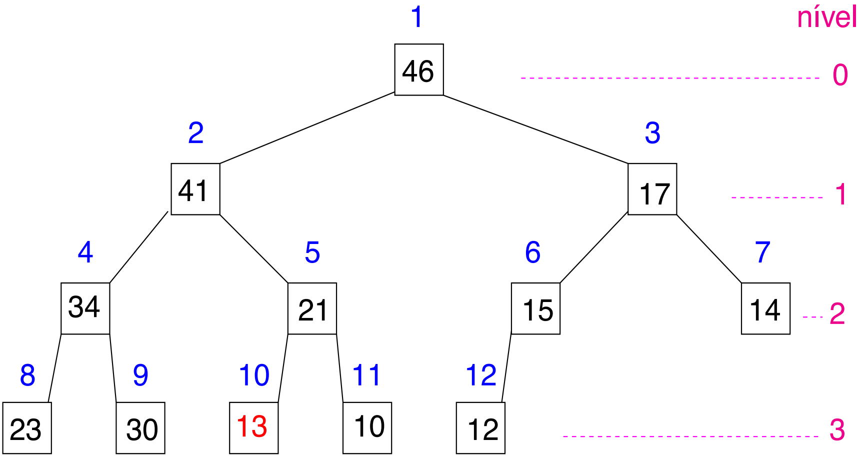
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



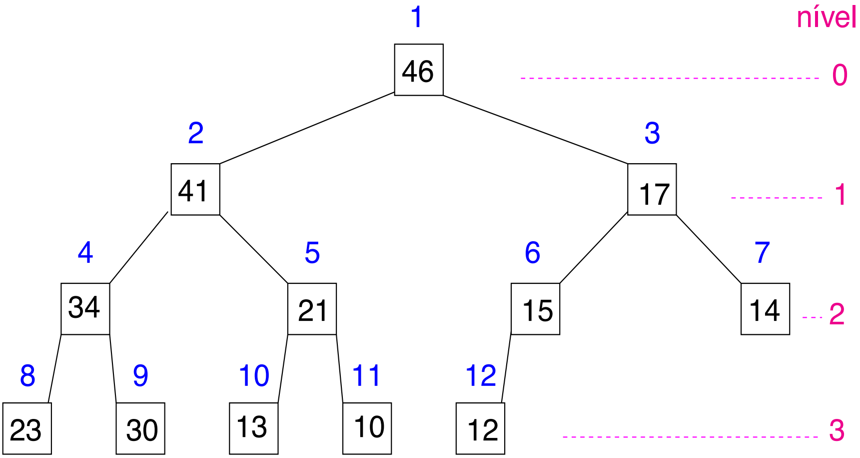
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

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)
e ← 2i
d ← 2i + 1
se e ≤ n  e A[e] > A[i]
      então maior ← e
      senão maior ← i
se d ≤ n e A[d] > A[maior]
      então maior ← d
se maior ≠ i
      então A[i] ↔ A[maior]
                  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

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 $A[i] \leftrightarrow A[\text{maior}]$	?
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
$e \leftarrow 2i$	$\Theta(1)$
$d \leftarrow 2i + 1$	$\Theta(1)$
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 $A[i] \leftrightarrow A[\text{maior}]$	$O(1)$
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

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

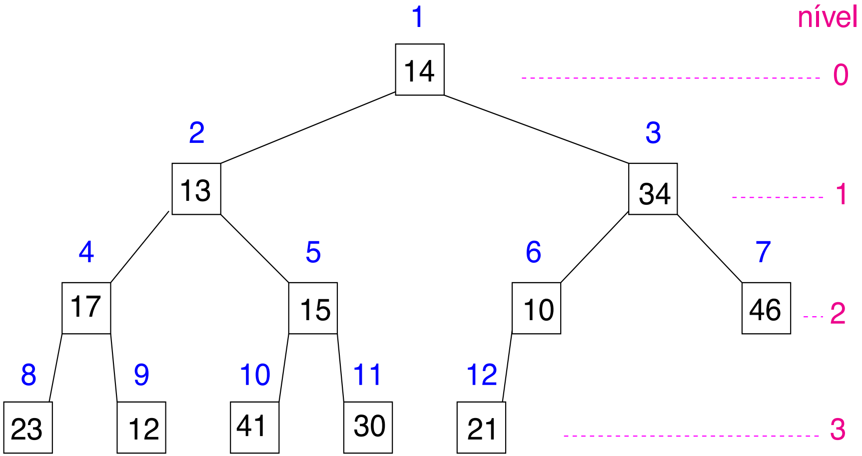
Solução assintótica: $T(n)$ é ???.

Solução assintótica: $T(n)$ é $O(h)$.

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

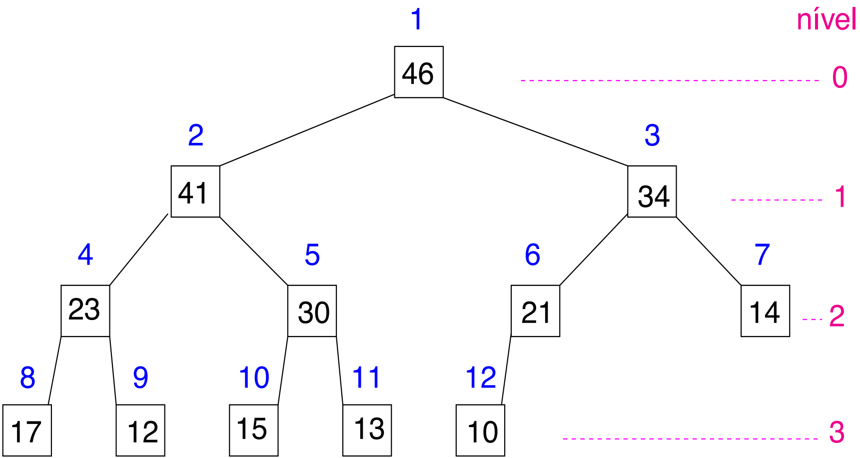
O consumo de tempo do algoritmo Max-Heapify é $O(\lg n)$ (ou melhor ainda, $O(\lg \frac{n}{i})$).

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
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 $\lfloor \lg n \rfloor + 1$.

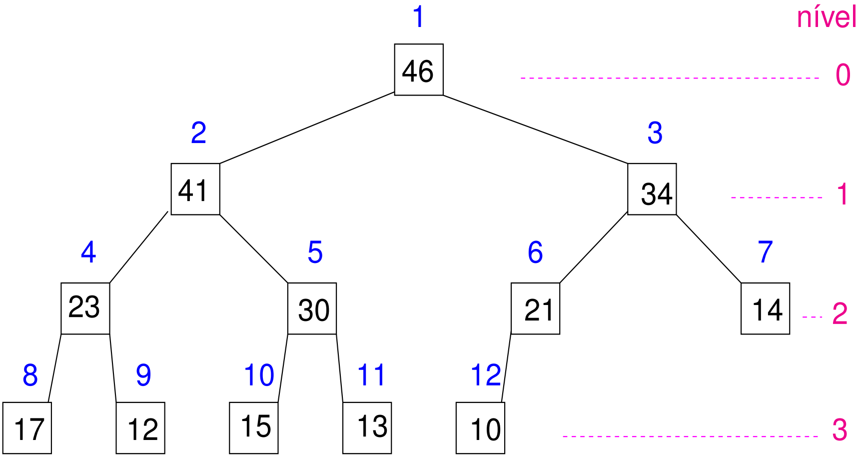
  A complexidade de BUILDMAXHEAP é $T(n) = O(S(\lg n))$.

Construção de um max-heap

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

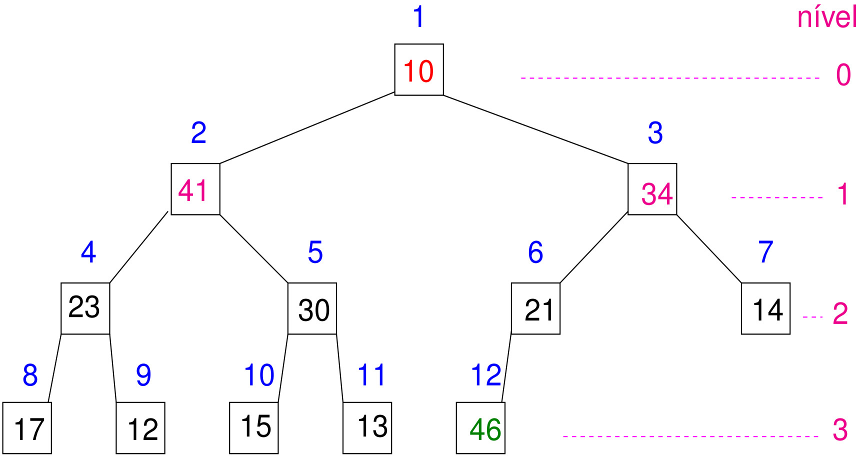
  Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)
- Veja no CLRS uma prova diferente deste fato.

HeapSort



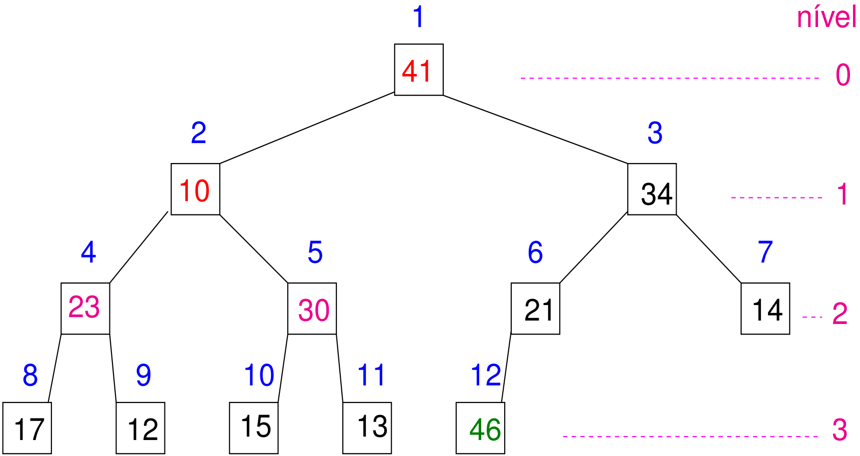
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

HeapSort



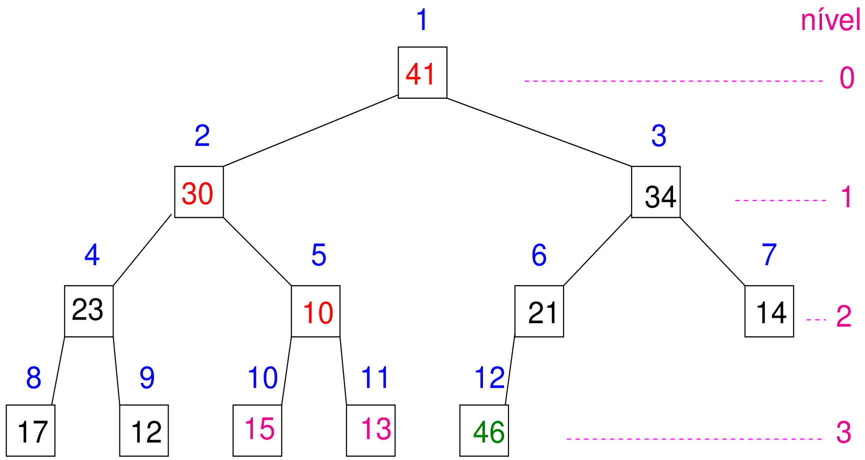
1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46

HeapSort



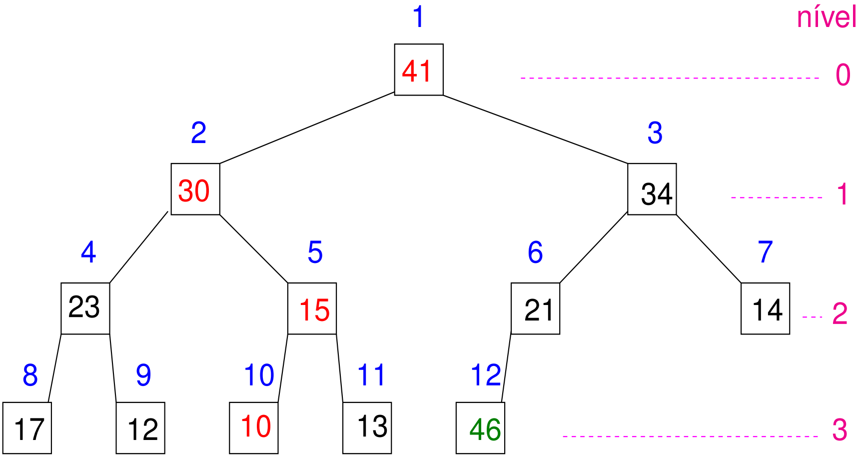
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

HeapSort



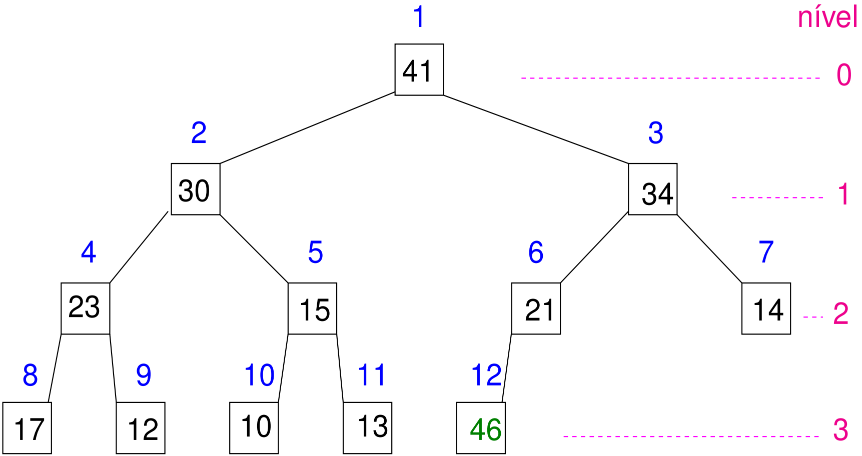
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

HeapSort



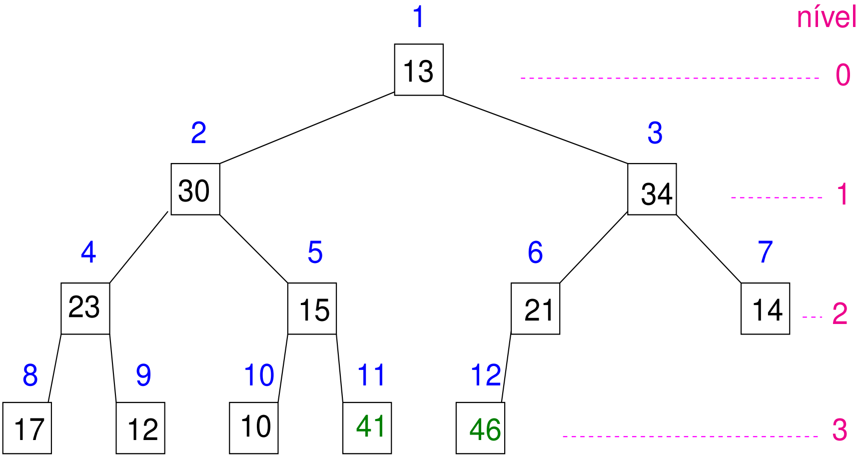
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort

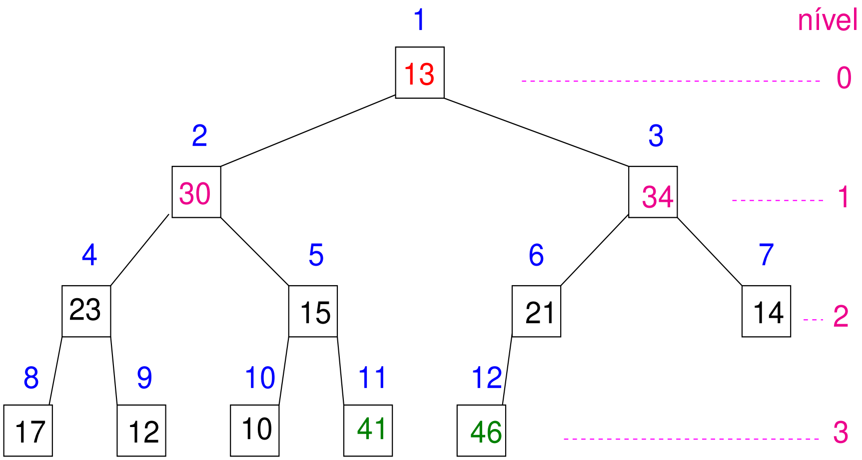


1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort

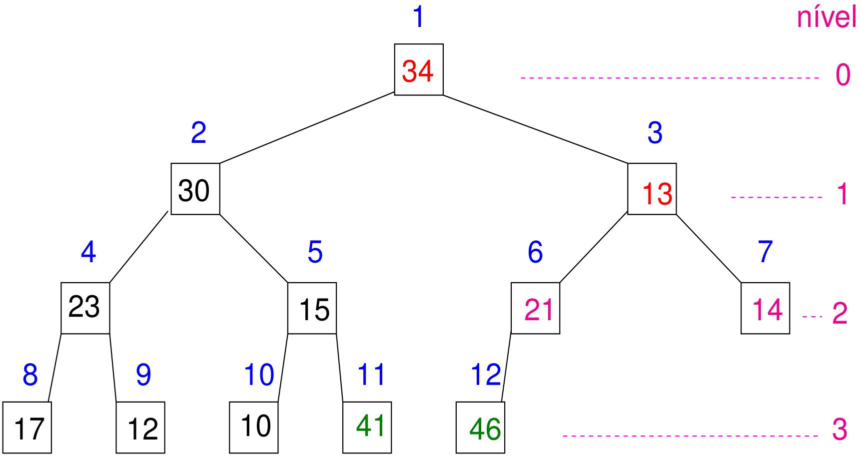


1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46



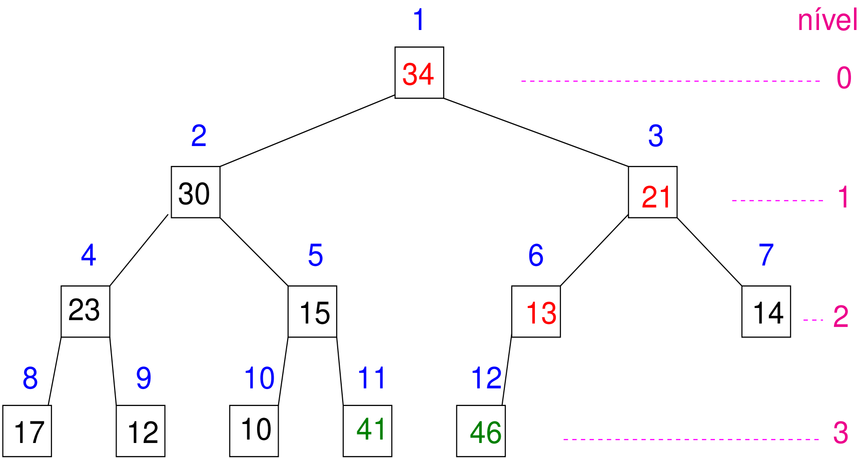
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



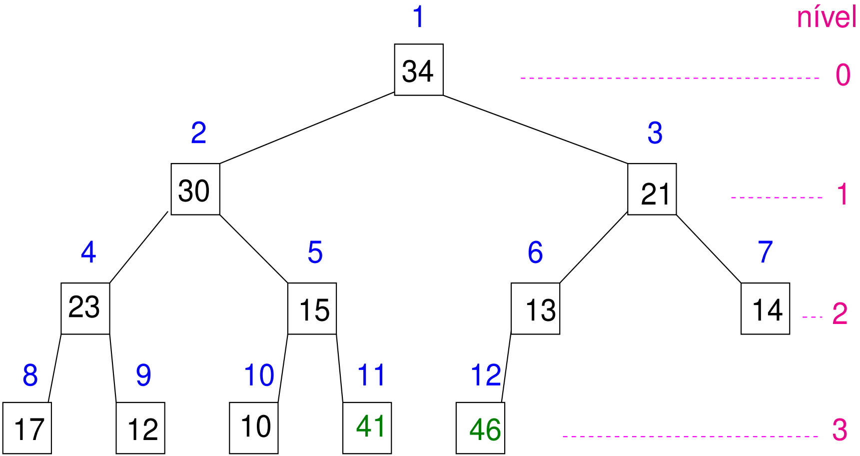
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

HeapSort



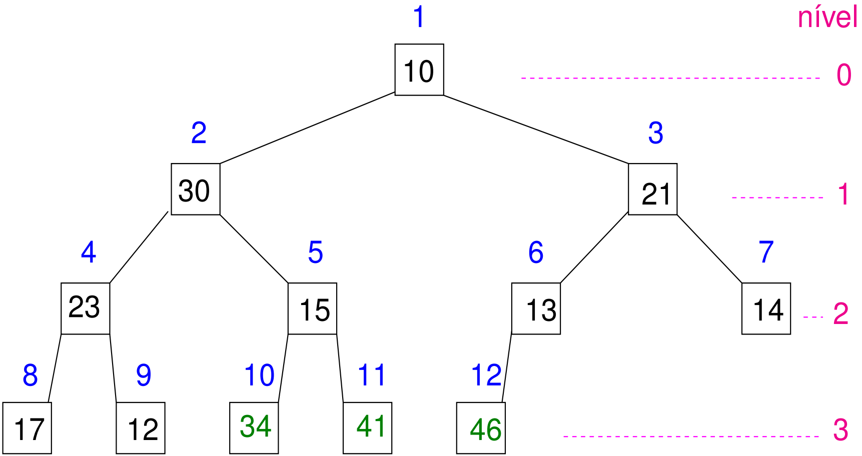
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



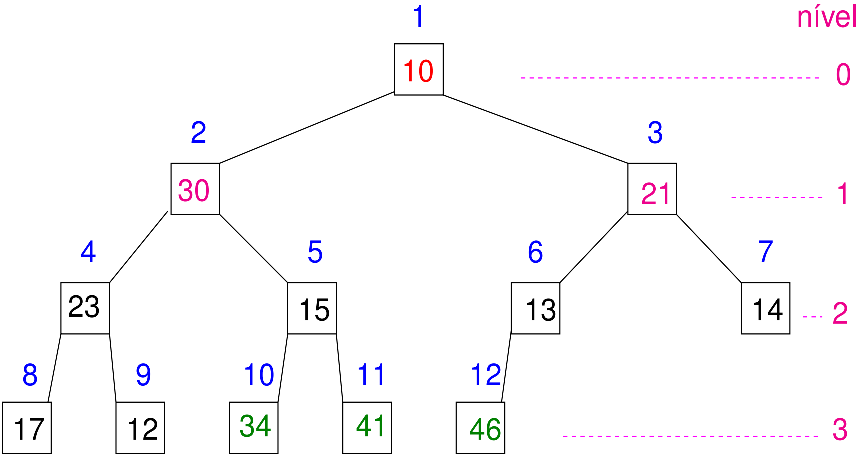
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



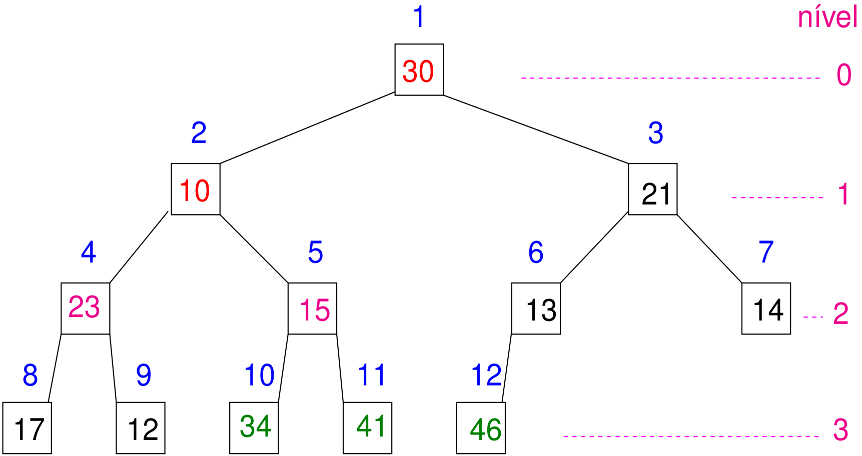
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



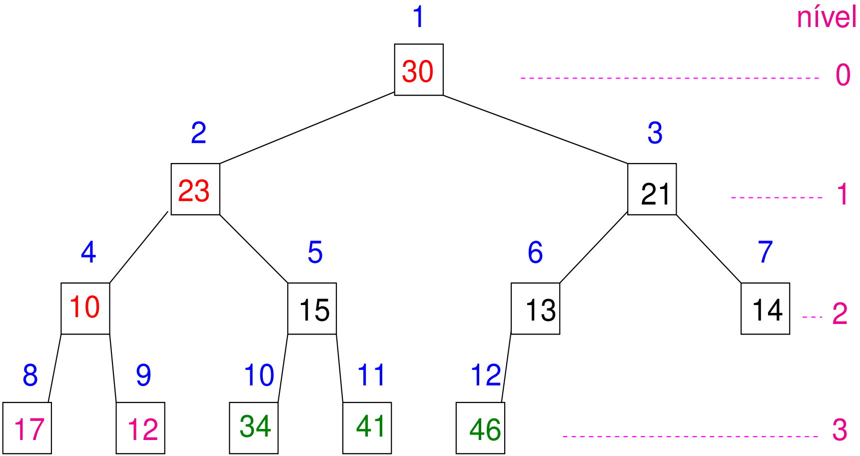
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



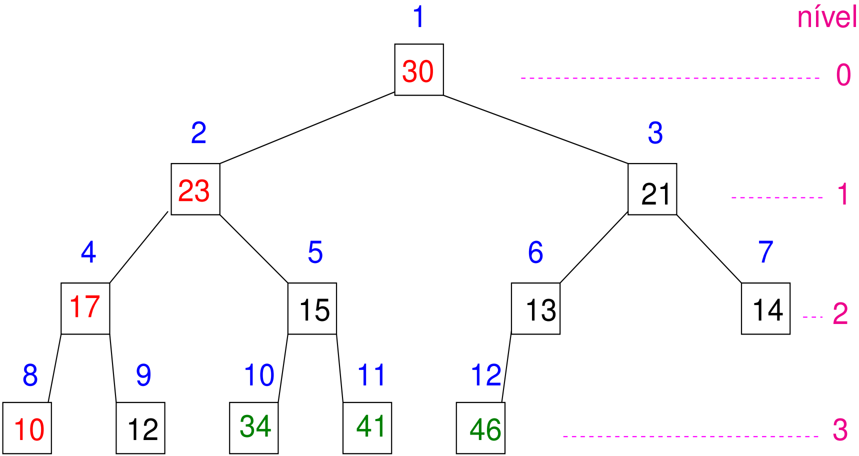
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

HeapSort



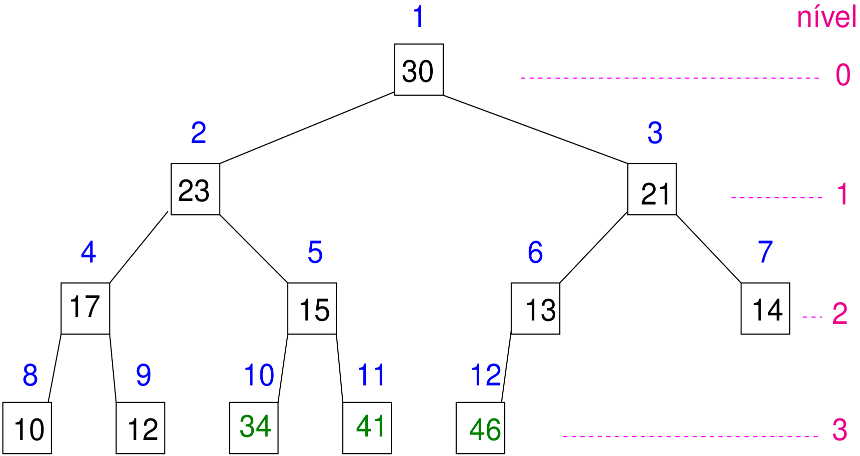
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

HeapSort



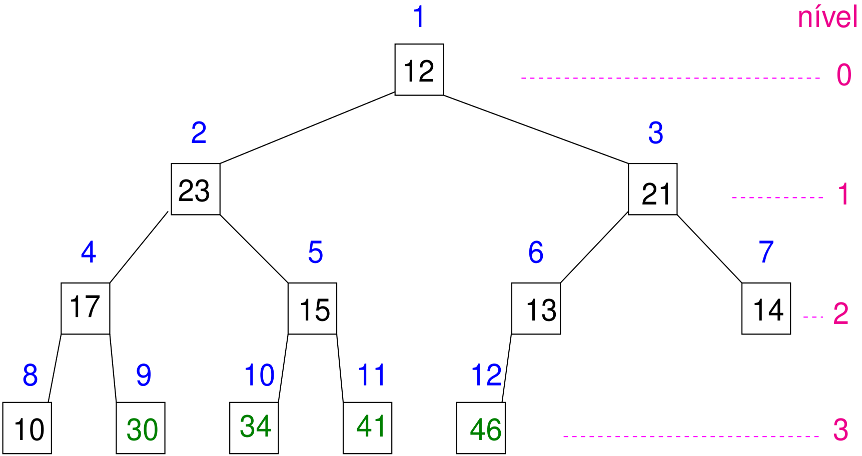
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

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

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

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)

```
▷ n ≥ 1
max ← A[1]
A[1] ← A[n]
n ← 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: $O(\lg n)$.

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 $A[1 \dots n]$ 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* e *estáveis*

- Algoritmos de ordenação podem ser ou não *in-place* ou *estáveis*.
- Um algoritmo de ordenação é in-place se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
- Exemplos: QUICKSORT e HEAPSORT são métodos de ordenação *in-place*, já MERGESORT e COUNTING-SORT não são.
- 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 e QUICKSORT são exemplos de métodos estáveis (desde que certos cuidados sejam tomados na implementação). HEAPSORT não é.

- 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

$$
\begin{array}{ccccccc}
329 & & 720 & & 720 & & 329 \\
457 & & 355 & & 329 & & 355 \\
657 & & 436 & & 436 & & 436 \\
839 & & 457 & & 839 & & 457 \\
436 & \rightarrow & 657 & \rightarrow & 355 & \rightarrow & 657 \\
720 & & 329 & & 457 & & 720 \\
355 & & 839 & & 657 & & 839 \\
 & & \uparrow & & \uparrow & & \uparrow
\end{array}
$$

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.

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)
    mín ← A[1]
    para j ← 2 até n faça
        se mín > A[j]
            então mín ← 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)
    mín ← máx ← A[1]
    para j ← 2 até n faça
        se A[j] < mín
            então mín ← A[j]
        se A[j] > máx
            então máx ← A[j]
    devolva (mín, 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:

$$\begin{array}{ll} 3\lfloor n/2 \rfloor & \text{se } n \text{ for ímpar} \\ 3\lfloor n/2 \rfloor - 2 & \text{se } n \text{ for par} \end{array}$$

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)
1 ORDENE(A, n)
2 **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 $A[p \ldots r]$ de modo que $p \leq q \leq r$ e $A[p \ldots q{-}1] \leq A[q] < A[q{+}1 \ldots r]$.

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

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

1 $j \leftarrow$ RANDOM(p, r)
2 $A[j] \leftrightarrow A[r]$
3 **devolva** PARTICIONE(A, p, r)

Algoritmo SELECT-ALEAT

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-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.

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

$T(n)$ é $\Theta(???)$.

Análise do caso médio

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

pois

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

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.

Demonstração: $T(n) \leq cn$

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

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.

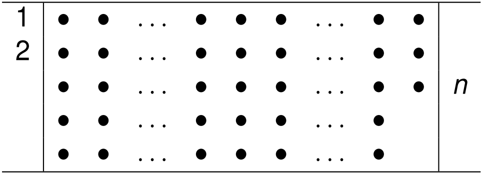
Veremos um algoritmo linear para o Problema da Seleção.

BFPRT = Blum, Floyd, Pratt, Rivest e Tarjan

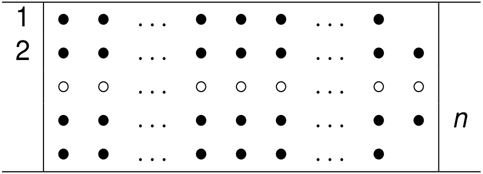
Para simplificar a exposição, vamos supor que os elementos em A são distintos.

Problema da Seleção – terceira solução

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



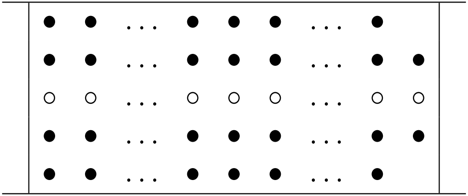
2. Encontre a mediana de cada um dos $\lceil \frac{n}{5} \rceil$ subconjuntos.



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.

$$
\boxed{\begin{array}{ccccccccc}
\square & \square & \dots & \square & \square & \bullet & \dots & \bullet & \\
\square & \square & \dots & \square & \square & \bullet & \dots & \bullet & \bullet \\
\square & \square & \dots & \square & x & \triangle & \dots & \triangle & \triangle \\
\bullet & \bullet & \dots & \bullet & \triangle & \triangle & \dots & \triangle & \triangle \\
\bullet & \bullet & \dots & \bullet & \triangle & \triangle & \dots & \triangle &
\end{array}}
\quad
\begin{array}{ccc}
\square & = & < x \\
\triangle & = & > x \\
\bullet & = & ?
\end{array}
$$

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 □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: $T(n) \le cn$

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

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

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

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)
se $p = r$
    então devolva p $\triangleright$ p e não $A[p]$
$q \leftarrow$ Particione-BFPRT(A, p, r)
$k \leftarrow q - p + 1$
se $i = k$
    então devolva q $\triangleright$ q e não $A[q]$
    senão se $i < k$
        então devolva Select$(A, p, q - 1, i)$
        senão devolva Select$(A, q + 1, r, i - k)$

PARTICIONE-BFPRT

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

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

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) $\quad \triangleright\ n := r - p + 1$

para $j \leftarrow p, p{+}5, p{+}5 \cdot 2, \ldots$ **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)