

Projeto e Análise de Algoritmos

A. G. Silva e R. de Santiago

Baseado nos materiais de
Souza, Silva, Lee, Rezende, Miyazawa – Unicamp
Ribeiro – FCUP
Manber, Introduction to Algorithms (1989) – Livro

12 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** – 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)**

Teorema master

Teorema Master

- Veremos agora um resultado que descreve soluções para recorrências da forma

$$T(n) = aT(n/b) + f(n),$$

onde $a \geq 1$ e $b > 1$ são constantes.

- O **caso base** é omitido na definição e convenciona-se que é uma **constante** para valores pequenos.
- A expressão n/b pode indicar tanto $\lfloor n/b \rfloor$ quanto $\lceil n/b \rceil$.
- O Teorema Master **não** fornece a resposta para **todas** as recorrências da forma acima.

Teorema Master (Manber)

Teorema mestre (Manber)

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k$$

$$a \geq 1 \quad b > 2 \quad c > 0 \quad k \geq 0 \quad a, b \in \mathbb{N}$$

1 $T(n) \in \Theta(n^{\log_b a})$
se $a > b^k$

2 $T(n) \in \Theta(n^k \log n)$
se $a = b^k$

3 $T(n) \in \Theta(n^k)$
se $a < b^k$

Teorema Master (Cormen et al)

Teorema mestre (Cormen et al)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$a \geq 1 \quad b > 1$$

- ① $T(n) \in \Theta(n^{\log_b a})$
se $f(n) \in O(n^{(\log_b a)-\epsilon})$ para $\epsilon > 0$
- ② $T(n) \in \Theta(n^{\log_b a} \log n)$
se $f(n) \in \Theta(n^{\log_b a})$
- ③ $T(n) \in \Theta(f(n))$
se $f(n) \in \Omega(n^{(\log_b a)+\epsilon})$ para $\epsilon > 0$
e se $af\left(\frac{n}{b}\right) \leq cf(n)$ para $c < 1$
e n suficientemente grande

Exemplos de Recorrências

Exemplos onde o Teorema Master se aplica:

- Caso 1:

$$T(n) = 9T(n/3) + n$$

$$T(n) = 4T(n/2) + n \log n$$

- Caso 2:

$$T(n) = T(2n/3) + 1$$

$$T(n) = 2T(n/2) + (n + \log n)$$

- Caso 3:

$$T(n) = T(3n/4) + n \log n$$

Exemplos onde o Teorema Master **não se aplica**:

- $T(n) = T(n - 1) + n$
- $T(n) = T(n - a) + T(a) + n, (a \geq 1 \text{ inteiro})$
- $T(n) = T(\alpha n) + T((1 - \alpha)n) + n, (0 < \alpha < 1)$
- $T(n) = T(n - 1) + \log n$
- $T(n) = 2T(\frac{n}{2}) + n \log n$

Divisão e Conquista

- **Dividir para conquistar:** uma tática de guerra aplicada ao projeto de algoritmos.
- Um algoritmo de divisão e conquista é aquele que resolve o problema desejado combinando as soluções parciais de (um ou mais) subproblemas, obtidas recursivamente.
- É mais um paradigma de projeto de algoritmos baseado no princípio da indução.
- Informalmente, podemos dizer que o **paradigma incremental** representa o projeto de algoritmos por indução fraca, enquanto o **paradigma de divisão e conquista** representa o projeto por indução forte.
- É natural, portanto, demonstrar a corretude de algoritmos de divisão e conquista por indução.

DivisaoConquista(x)

- ▷ Entrada: A instância x
 - ▷ Saída: Solução y do problema em questão para x
1. se x é suficientemente pequeno então
 - ▷ $Solucao(x)$ algoritmo para pequenas instâncias
 2. retorne $Solucao(x)$
 3. senão
 - ▷ divisão
 4. decomponha x em instâncias menores x_1, x_2, \dots, x_k
 5. para i de 1 até k faça $y_i := DivisaoConquista(x_i)$
 - ▷ conquista
 6. combine as soluções y_i para obter a solução y de x .
 7. retorne(y)

Projeto por Divisão e Conquista - Exemplo 1

Exponenciação

Problema:

Calcular a^n , para todo real a e inteiro $n \geq 0$.

Primeira solução, por indução fraca:

- **Caso base:** $n = 0; a^0 = 1$.
- **Hipótese de indução:** Suponha que, para qualquer inteiro $n > 0$ e real a , sei calcular a^{n-1} .
- **Passo da indução:** Queremos provar que conseguimos calcular a^n , para $n > 0$. Por hipótese de indução, sei calcular a^{n-1} . Então, calculo a^n multiplicando a^{n-1} por a .

Exemplo 1 - Solução 1 - Algoritmo

Exponenciacao(a, n)

- ▷ **Entrada:** A base a e o expoente n .
 - ▷ **Saída:** O valor de a^n .
1. **se** $n = 0$ **então**
 2. **retorne**(1) {caso base}
 3. **senão**
 4. $an' := Exponenciacao(a, n - 1)$
 5. $an := an' * a$
 6. **retorne**(an)

Exemplo 1 - Solução 1 - Complexidade

Seja $T(n)$ o número de operações executadas pelo algoritmo para calcular a^n .

Então a relação de recorrência deste algoritmo é:

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(n - 1) + c_2, & n > 0, \end{cases}$$

onde c_1 e c_2 representam, respectivamente, o tempo (constante) executado na atribuição da base e multiplicação do passo.

Neste caso, não é difícil ver que

$$T(n) = c_1 + \sum_{i=1}^n c_2 = c_1 + nc_2 = \Theta(n).$$

Este algoritmo é linear no tamanho da entrada ?

Exemplo 1 - Solução 2 - Divisão e Conquista

Vamos agora projetar um algoritmo para o problema usando indução forte de forma a obter um algoritmo de divisão e conquista.

Segunda solução, por indução forte:

- **Caso base:** $n = 0; a^0 = 1$.
- **Hipótese de indução:** Suponha que, para qualquer inteiro $n > 0$ e real a , sei calcular a^k , para todo $k < n$.
- **Passo da indução:** Queremos provar que conseguimos calcular a^n , para $n > 0$. Por hipótese de indução sei calcular $a^{\lfloor \frac{n}{2} \rfloor}$. Então, calculo a^n da seguinte forma:

$$a^n = \begin{cases} \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2, & \text{se } n \text{ par;} \\ a \cdot \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2, & \text{se } n \text{ ímpar.} \end{cases}$$

Exemplo 1 - Solução 2 - Algoritmo

ExponenciacaoDC(a, n)

▷ Entrada: A base a e o expoente n .

▷ Saída: O valor de a^n .

1. **se** $n = 0$ **então**
2. **retorne**(1) {caso base}
3. **senão**
 - ▷ divisão
4. $an' := ExponenciacaoDC(a, n \text{ div } 2)$
 - ▷ conquista
5. $an := an' * an'$
6. **se** $(n \text{ mod } 2) = 1$
7. $an := an * a$
8. **retorne**(an)

Exemplo 1 - Solução 2 - Complexidade

- Seja $T(n)$ o número de operações executadas pelo algoritmo de divisão e conquista para calcular a^n .
- Então a relação de recorrência deste algoritmo é:

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + c_2, & n > 0, \end{cases}$$

- Não é difícil ver que $T(n) \in \Theta(\log n)$. Por quê?

Projeto por Divisão e Conquista - Exemplo 2

Busca Binária

Problema:

Dado um vetor ordenado A com n números reais e um real x , determinar a posição $1 \leq i \leq n$ tal que $A[i] = x$, ou que não existe tal i .

- O projeto de um algoritmo para este problema usando indução simples, nos leva a um algoritmo incremental de complexidade de pior caso $\Theta(n)$. **Pense em como seria a indução !**
- Se utilizarmos indução forte para projetar o algoritmo, podemos obter um algoritmo de divisão e conquista que nos leva ao algoritmo de busca binária. **Pense na indução !**
- Como o vetor está ordenado, conseguimos determinar, com apenas uma comparação, que *metade* das posições do vetor não pode conter o valor x .

Exemplo 2 - Algoritmo

BuscaBinaria(A, e, d, x)

- ▷ **Entrada:** Vetor A , delimitadores e e d do subvetor e x .
 - ▷ **Saída:** Índice $1 \leq i \leq n$ tal que $A[i] = x$ ou $i = 0$.
1. **se** $e = d$ **então se** $A[e] = x$ **então retorne**(e)
 2. **senão retorne**(0)
 3. **senão**
 4. $i := (e + d) \text{ div } 2$
 5. **se** $A[i] = x$ **retorne**(i)
 6. **senão se** $A[i] > x$
 7. $i := BuscaBinaria(A, e, i - 1, x)$
 8. **senão** $\{A[i] < x\}$
 9. $i := BuscaBinaria(A, i + 1, d, x)$
 10. **retorne**(i)

Exemplo 2 - Complexidade

- O número de operações $T(n)$ executadas na busca binária no pior caso é:

$$T(n) = \begin{cases} c_1, & n = 1 \\ T(\lceil \frac{n}{2} \rceil) + c_2, & n > 1, \end{cases}$$

- Não é difícil ver que $T(n) \in \Theta(\log n)$. **Por quê?**
- O algoritmo de busca binária (divisão e conquista) tem complexidade de pior caso $\Theta(\log n)$, que é assintoticamente melhor que o algoritmo de busca linear (incremental).
- E se o vetor não estivesse ordenado, qual paradigma nos levaria a um algoritmo assintoticamente melhor ?

Projeto por Divisão e Conquista - Exemplo 3 - Máximo e Mínimo

Problema:

Dado um conjunto S de $n \geq 2$ números reais, determinar o maior e o menor elemento de S .

- Um algoritmo incremental para esse problema faz $2n - 3$ comparações: fazemos uma comparação no caso base e duas no passo.
- Será que um algoritmo de divisão e conquista seria melhor ?
- Um possível algoritmo de divisão e conquista seria:
 - Divida S em dois subconjuntos de mesmo tamanho S_1 e S_2 e solucione os subproblemas.
 - O máximo de S é o máximo dos máximos de S_1 e S_2 e o mínimo de S é o mínimo dos mínimos de S_1 e S_2 .

Exemplo 3 - Complexidade

- Qual o número de comparações $T(n)$ efetuado por este algoritmo?

$$T(n) = \begin{cases} 1, & n = 2 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2, & n > 2, \end{cases}$$

- Supondo que n é uma potência de 2, temos:

$$T(n) = \begin{cases} 1, & n = 2 \\ 2T\left(\frac{n}{2}\right) + 2, & n > 2, \end{cases}$$

- Neste caso, podemos provar que $T(n) = \frac{3}{2}n - 2$ usando o método da substituição (indução!).

Exemplo 3 - Complexidade

- **Caso Base:** $T(2) = 1 = 3 - 2$.
- **Hipótese de Indução:** Suponha, para $n = 2^{k-1}$, $k \geq 2$, que $T(n) = \frac{3}{2}n - 2$.
- **Passo de Indução:** Queremos provar para $n = 2^k$, $k \geq 2$, que $T(n) = \frac{3}{2}n - 2$.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 2 \\ &= 2\left(\frac{3}{4}n - 2\right) + 2 \text{ (por h. i.)} \\ &= \frac{3}{2}n - 2. \end{aligned}$$

- É possível provar que $T(n) = \frac{3}{2}n - 2$ quando n não é potência de 2 ?

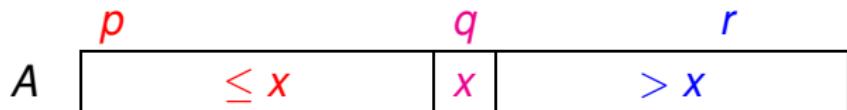
Exemplo 3 - Complexidade

- Assintoticamente, os dois algoritmos para este problema são equivalentes, ambos $\Theta(n)$.
- No entanto, o algoritmo de divisão e conquista permite que menos comparações sejam feitas. A estrutura hierárquica de comparações no retorno da recursão evita comparações desnecessárias.

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



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

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:

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

Saída:

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

Particione

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

Partizione

		<i>i</i>			<i>j</i>					<i>x</i>
A	33	11	55	77	99	22	88	66	33	44
		<i>i</i>			<i>j</i>					<i>x</i>
A	33	11	22	77	99	55	88	66	33	44
		<i>i</i>			<i>j</i>					<i>x</i>
A	33	11	22	77	99	55	88	66	33	44
		<i>i</i>			<i>j</i>					<i>x</i>
A	33	11	22	77	99	55	88	66	33	44
		<i>i</i>			<i>j</i>					<i>j</i>
A	33	11	22	33	99	55	88	66	77	44
		<i>p</i>			<i>q</i>					<i>r</i>
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** $i + 1$

Invariante:

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

QUICKSORT(A, p, r)

- ```

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

```

|          |    |    |    |    |    |    |    |    |    |    |
|----------|----|----|----|----|----|----|----|----|----|----|
| <i>A</i> | 99 | 33 | 55 | 77 | 11 | 22 | 88 | 66 | 33 | 44 |
|----------|----|----|----|----|----|----|----|----|----|----|

# QuickSort

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

**QUICKSORT**( $A, p, r$ )

1    **se**  $p < r$

2    **então**  $q \leftarrow \text{PARTICIONE}(A, p, r)$

---

3            **QUICKSORT**( $A, p, q - 1$ )

4            **QUICKSORT**( $A, q + 1, r$ )

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

No começo da linha 3,

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

# QuickSort

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

**QUICKSORT**( $A, p, r$ )

1    se  $p < r$

2    então  $q \leftarrow \text{PARTICIONE}(A, p, r)$

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**( $A, p, r$ )

1    **se**  $p < r$

2        **então**  $q \leftarrow \text{PARTICIONE}(A, p, r)$

3            **QUICKSORT**( $A, p, q - 1$ )

4            **QUICKSORT**( $A, q + 1, r$ )

---

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

# Complexidade de Quicksort

| Quicksort( $A, p, r$ )                            | Tempo |
|---------------------------------------------------|-------|
| 1 se $p < r$                                      | ?     |
| 2 então $q \leftarrow \text{PARTICIONE}(A, p, r)$ | ?     |
| 3 Quicksort( $A, p, q - 1$ )                      | ?     |
| 4 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          |
|---------------------------------------------------|----------------|
| 1 se $p < r$                                      | $\Theta(1)$    |
| 2 então $q \leftarrow \text{PARTICIONE}(A, p, r)$ | $\Theta(n)$    |
| 3           QUICKSORT( $A, p, q - 1$ )            | $T(k)$         |
| 4           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

$$T(0) = \Theta(1)$$

$$T(1) = \Theta(1)$$

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

Recorrência grosseira:

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

$T(n)$  é  $\Theta(\text{???})$ .

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**

$$T(0) = \Theta(1)$$

$$T(1) = \Theta(1)$$

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

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

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

## Demonstração – $T(n) = O(n^2)$

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

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

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

## Continuação – $T(n) = \Omega(n^2)$

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

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

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

# Conclusão

$T(n)$  é  $\Theta(n^2)$ .

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

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

# QuickSort no melhor caso

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

$$M(0) = \Theta(1)$$

$$M(1) = \Theta(1)$$

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

Mostre que, para  $n \geq 1$ ,

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

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

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

# QuickSort no melhor caso

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

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

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**.

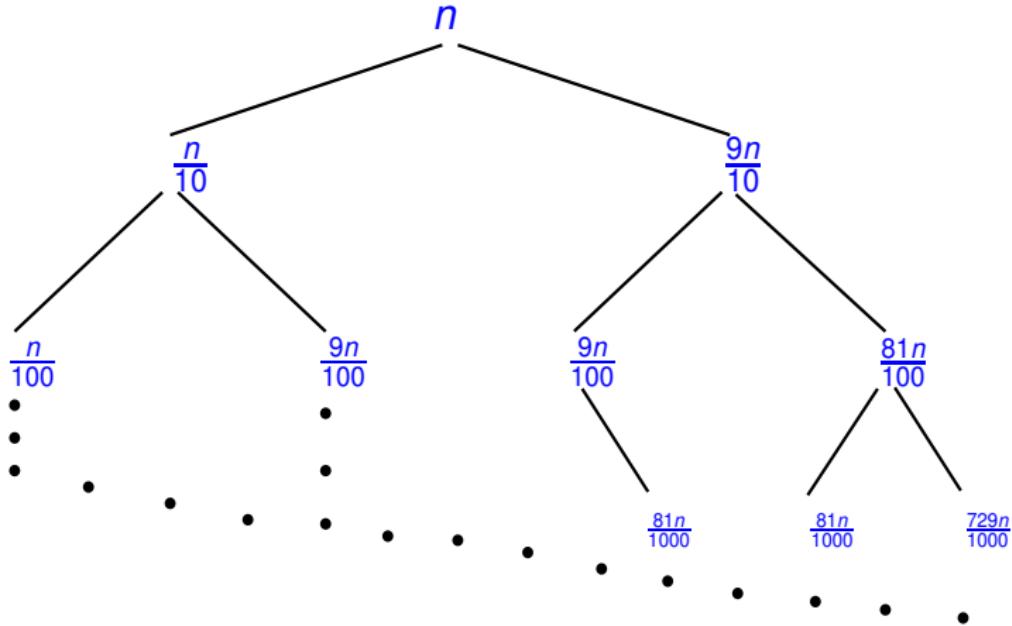
Por quê??

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

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

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**( $A, p, r$ )

- 1  $i \leftarrow \text{RANDOM}(p, r)$
- 2  $A[i] \leftrightarrow A[r]$
- 3 **devolva** **PARTICIONE**( $A, p, r$ )

**QUICKSORT-ALEATÓRIO**( $A, p, r$ )

- 1 **se**  $p < r$
- 2     **então**  $q \leftarrow \text{PARTICIONE-ALEATÓRIO}(A, p, r)$
- 3             **QUICKSORT-ALEATÓRIO**( $A, p, q - 1$ )
- 4             **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** rearanja o vetor  $A$  e devolve um índice  $q$  tal que  $A[p \dots q - 1] \leq A[q]$  e  $A[q + 1 \dots r] > A[q]$ .

$$T(0) = \Theta(1)$$

$$T(1) = \Theta(1)$$

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

$T(n)$  é  $\Theta(\text{???})$ .

# 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

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

Lema

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

## Demonstração

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

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

# Prova do Lema

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

# 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  $\Theta(n \lg n)$ .

## Ordenação – outros métodos importantes

# Algoritmos de ordenação

Algoritmos de ordenação:

- Insertion sort ✓
- Selection sort ✓
- Mergesort ✓
- Quicksort ✓
- Heapsort

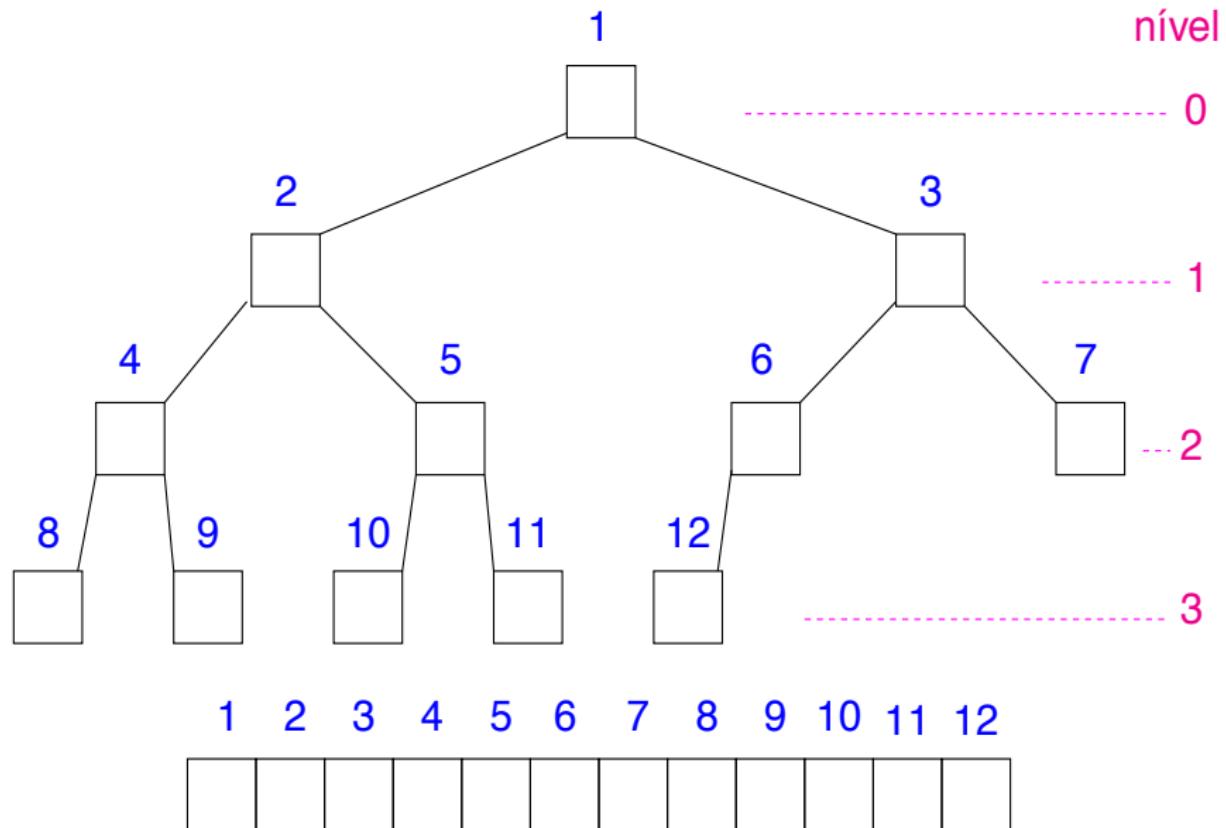
Algoritmos lineares:

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

## 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

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

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

$$2^h i, 2^{2h} i, 2^{3h} i, \dots, 2^h i$$

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

Assim,

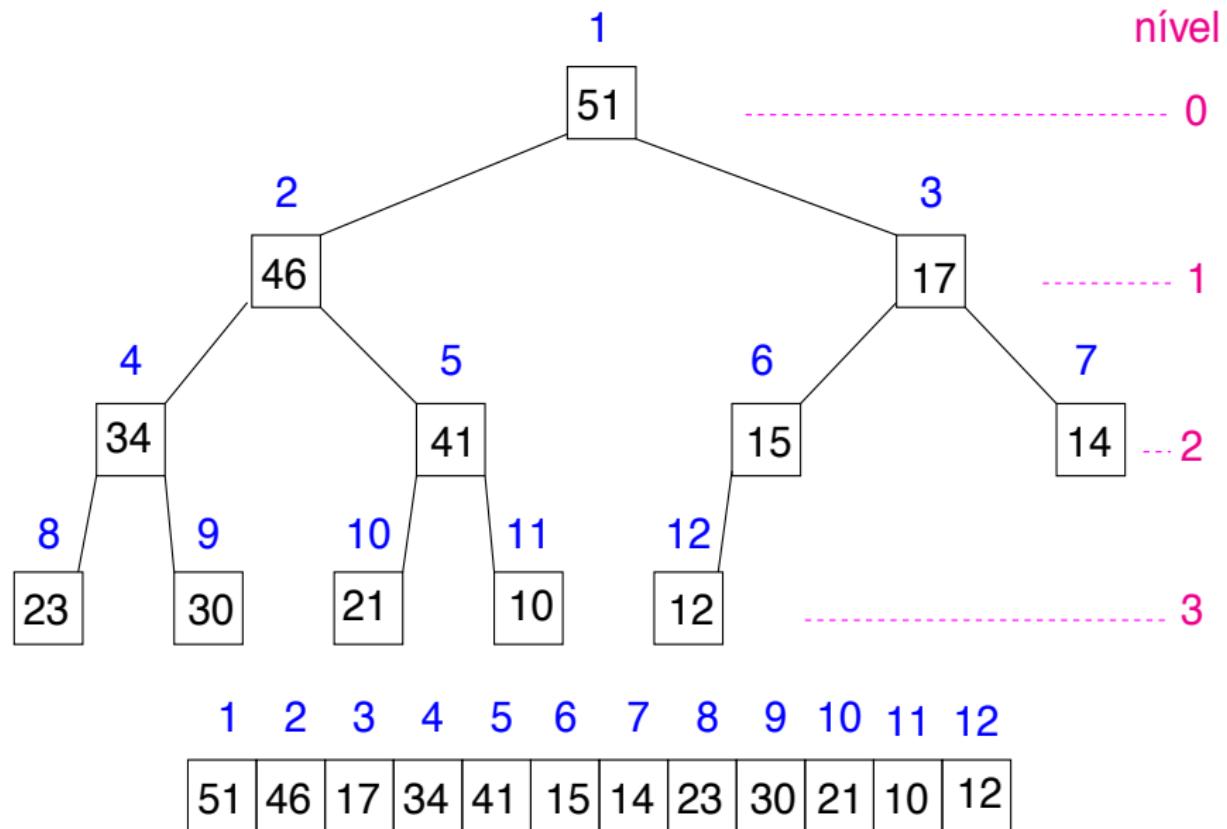
$$\begin{aligned} 2^h i &\leq n &< 2^{h+1} i &\Rightarrow \\ 2^h &\leq n/i &< 2^{h+1} &\Rightarrow \\ h &\leq \lg(n/i) &< h+1 \end{aligned}$$

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.

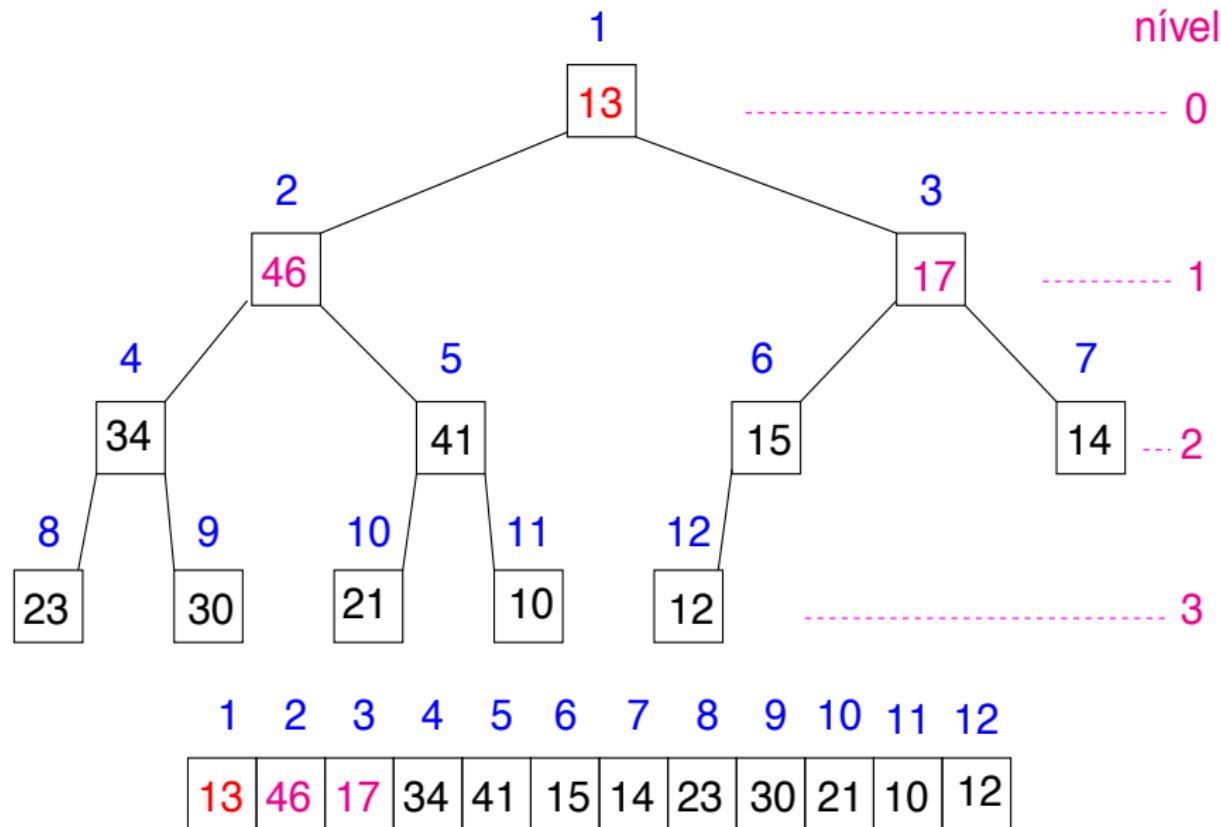
# Max-heap



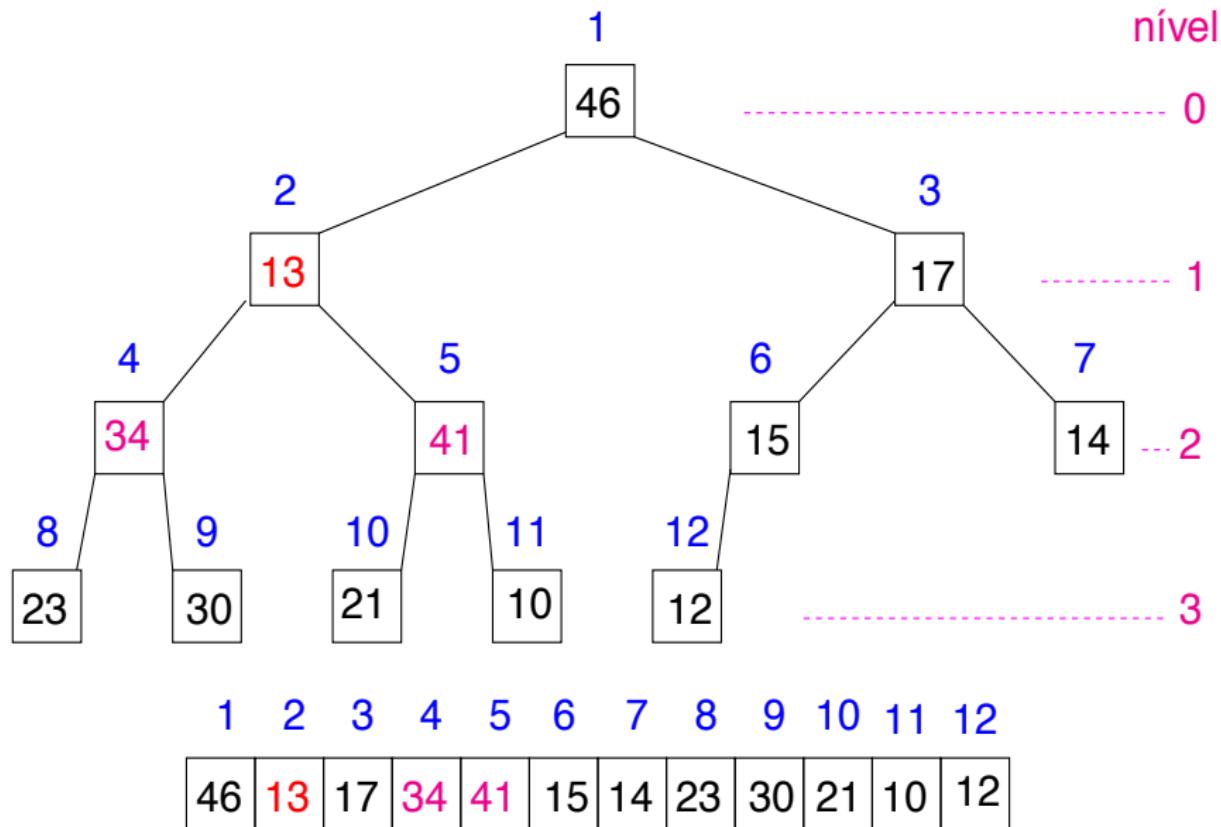
# Min-heaps

- 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**.

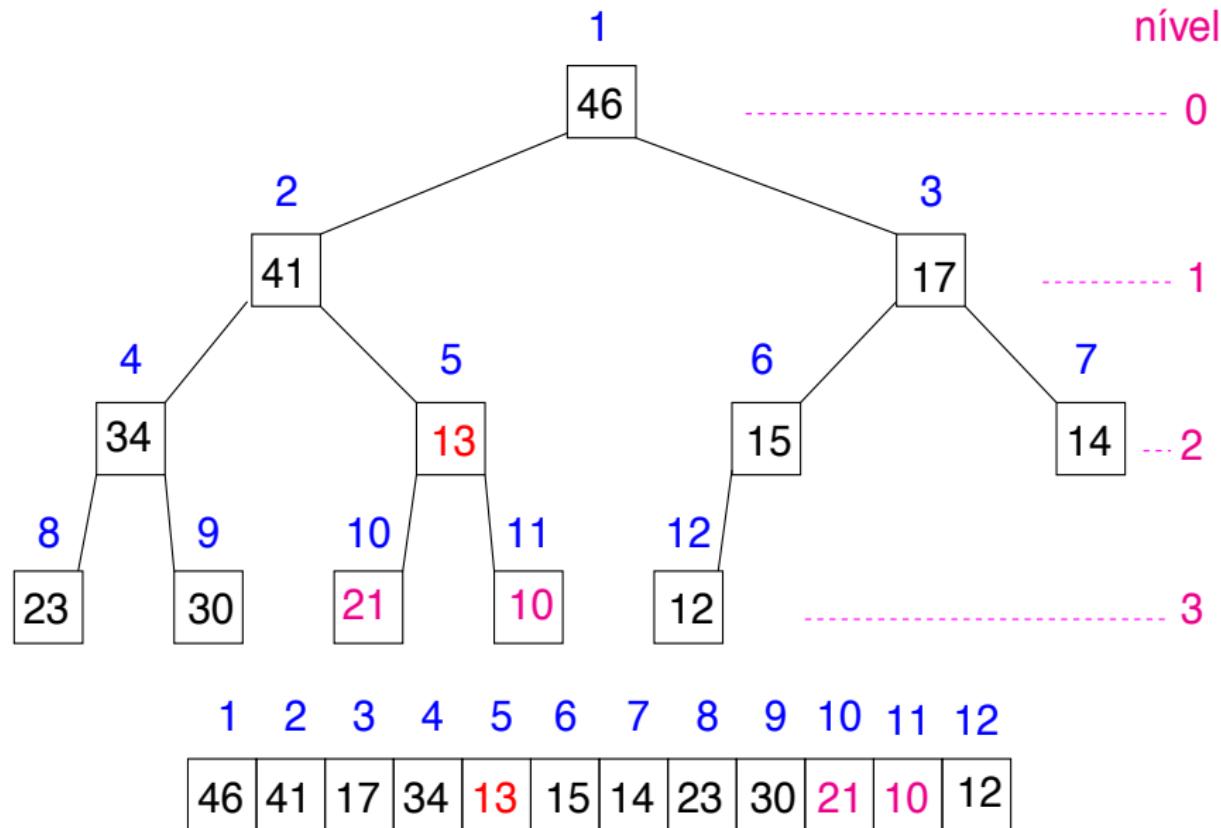
# Manipulação de max-heap



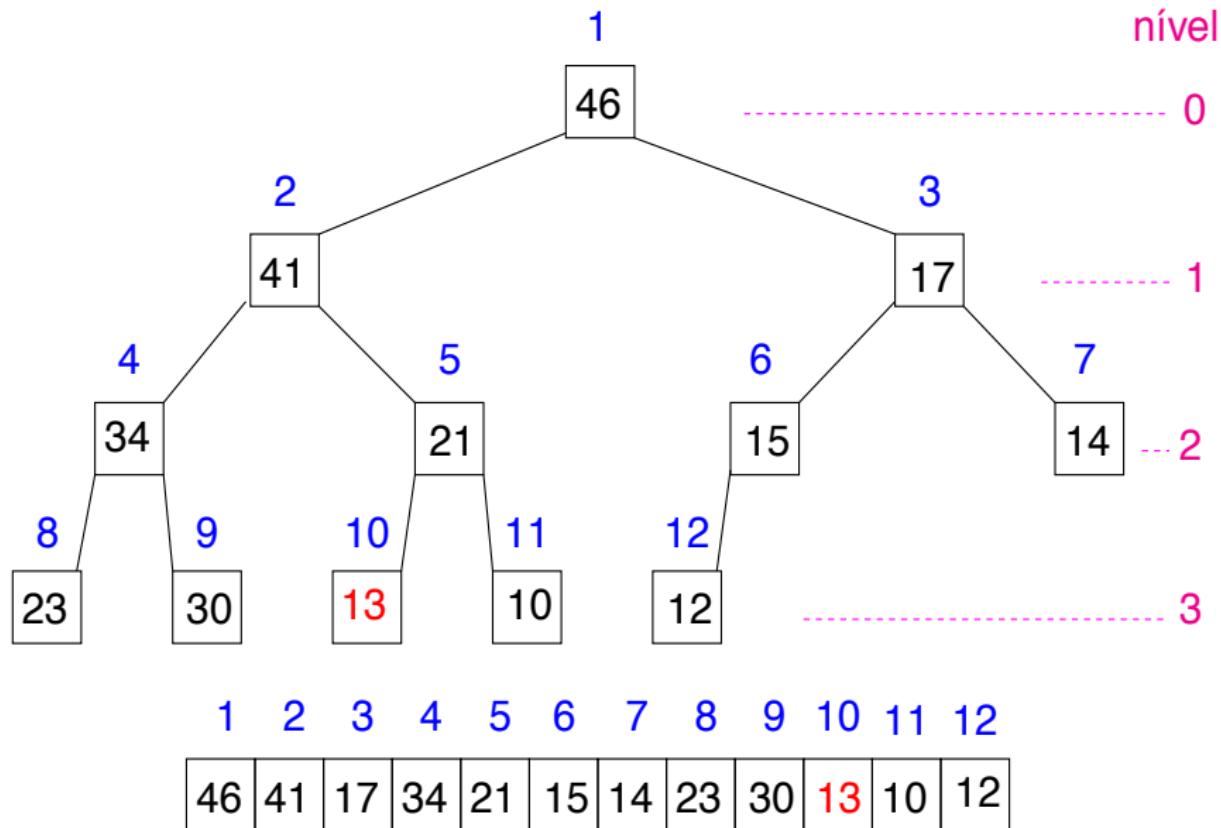
# Manipulação de max-heap



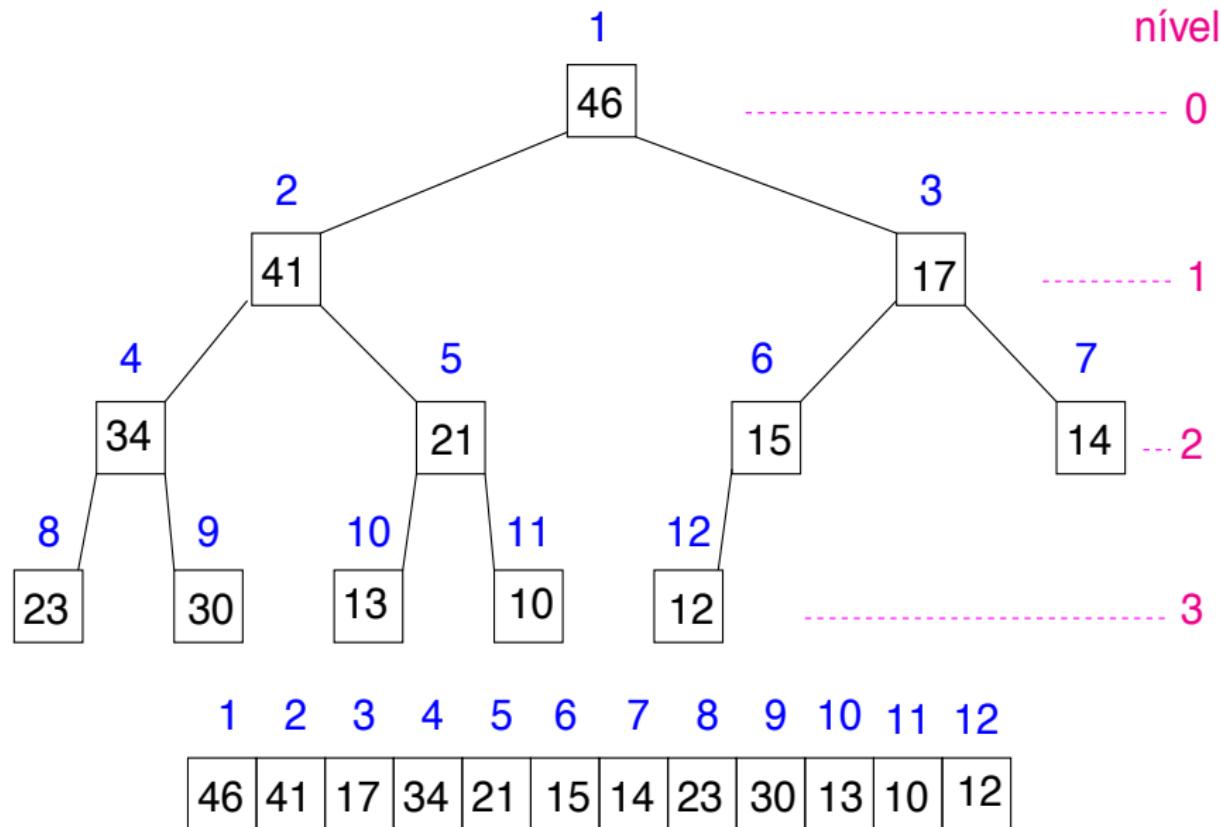
# Manipulação de max-heap



# Manipulação de max-heap



# Manipulação de max-heap



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

# 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

---

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

---

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

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

# Complexidade de MAXHEAPIFY

---

| <b>MAX-HEAPIFY</b> ( $A, n, i$ )                      | Tempo       |
|-------------------------------------------------------|-------------|
| 1 $e \leftarrow 2i$                                   | $\Theta(1)$ |
| 2 $d \leftarrow 2i + 1$                               | $\Theta(1)$ |
| 3 <b>se</b> $e \leq n$ e $A[e] > A[i]$                | $\Theta(1)$ |
| 4 <b>então</b> maior $\leftarrow e$                   | $O(1)$      |
| 5 <b>senão</b> maior $\leftarrow i$                   | $O(1)$      |
| 6 <b>se</b> $d \leq n$ e $A[d] > A[\text{maior}]$     | $\Theta(1)$ |
| 7 <b>então</b> maior $\leftarrow d$                   | $O(1)$      |
| 8 <b>se</b> maior $\neq i$                            | $\Theta(1)$ |
| 9 <b>então</b> $A[i] \leftrightarrow A[\text{maior}]$ | $O(1)$      |
| 10 <b>MAX-HEAPIFY</b> ( $A, n, \text{maior}$ )        | $T(h - 1)$  |

---

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

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

# Complexidade de MAXHEAPIFY

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

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

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

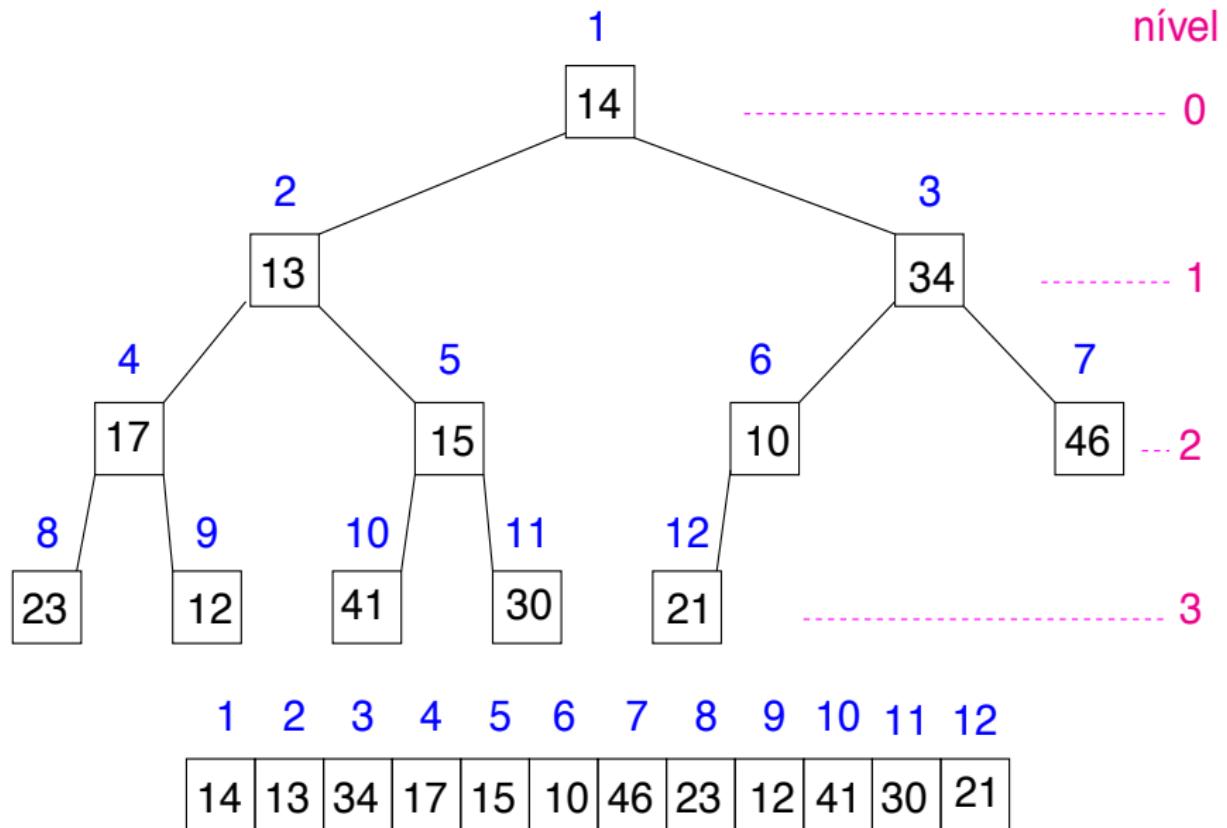
Solução assintótica:  $T(n)$  é ???.

Solução assintótica:  $T(n)$  é  $O(h)$ .

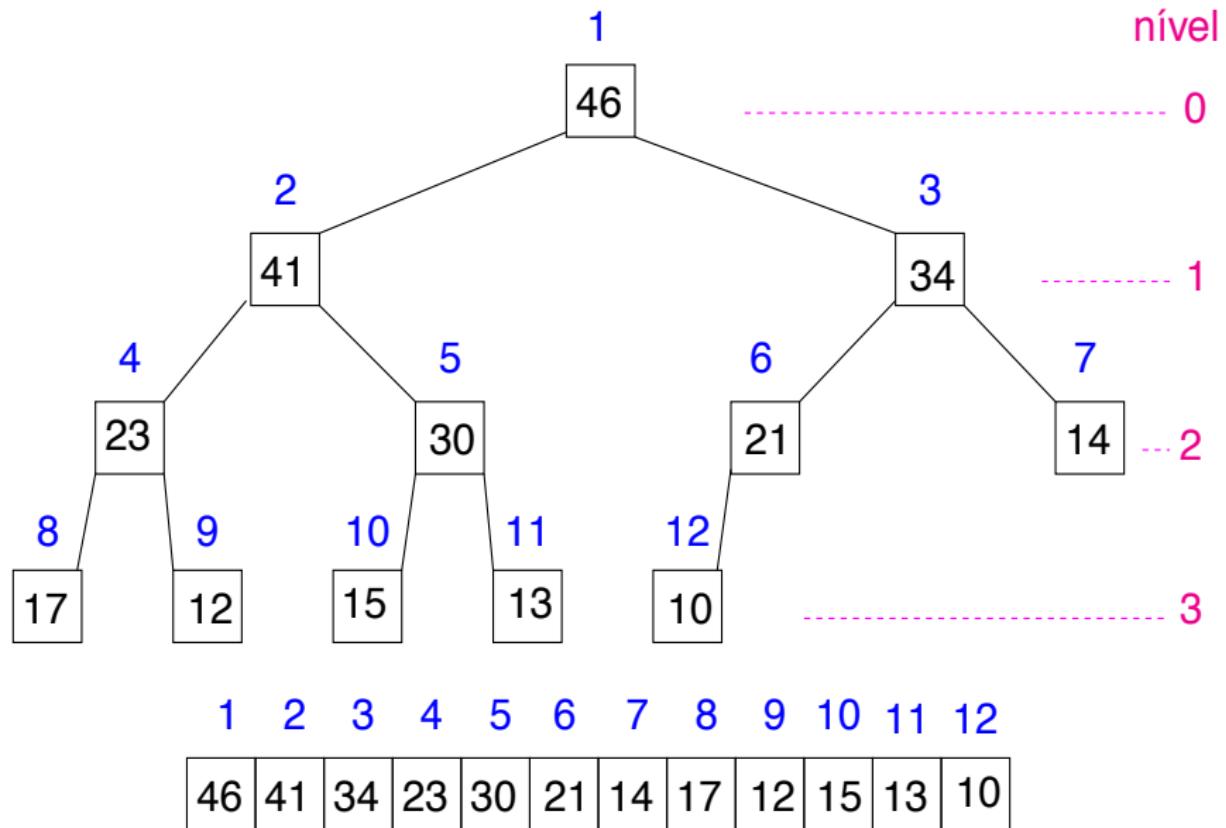
Como  $h \leq \lg n$ , podemos dizer que:

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



# Construção de um max-heap



# 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  $O(n)$ .

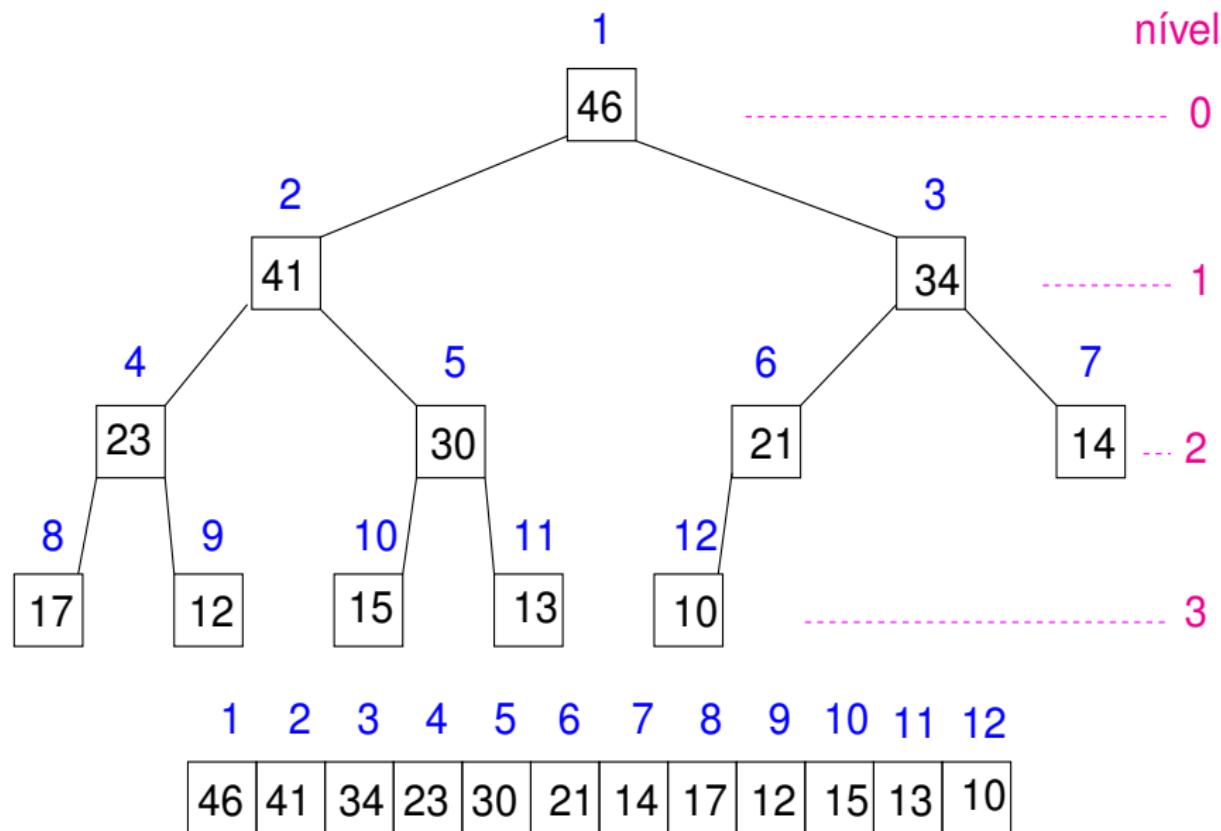
- Na iteração  $i$  são feitas  $O(h_i)$  comparações e trocas no pior caso, onde  $h_i$  é a altura da subárvore de raiz  $i$ .
- Seja  $S(h)$  a soma das alturas de todos os nós de uma árvore binária completa de altura  $h$ .
- A altura de um heap é  $\lfloor \lg n \rfloor + 1$ .

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

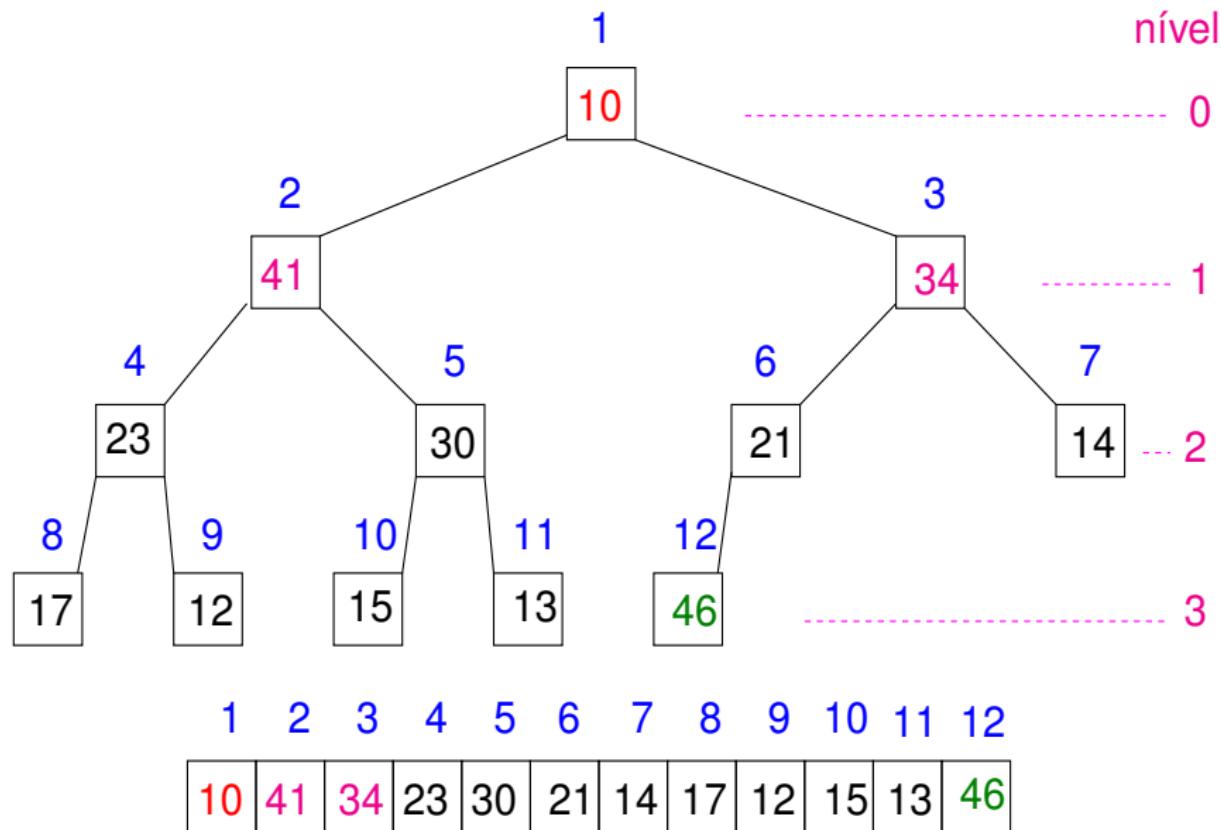
# Construção de um max-heap

- 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.

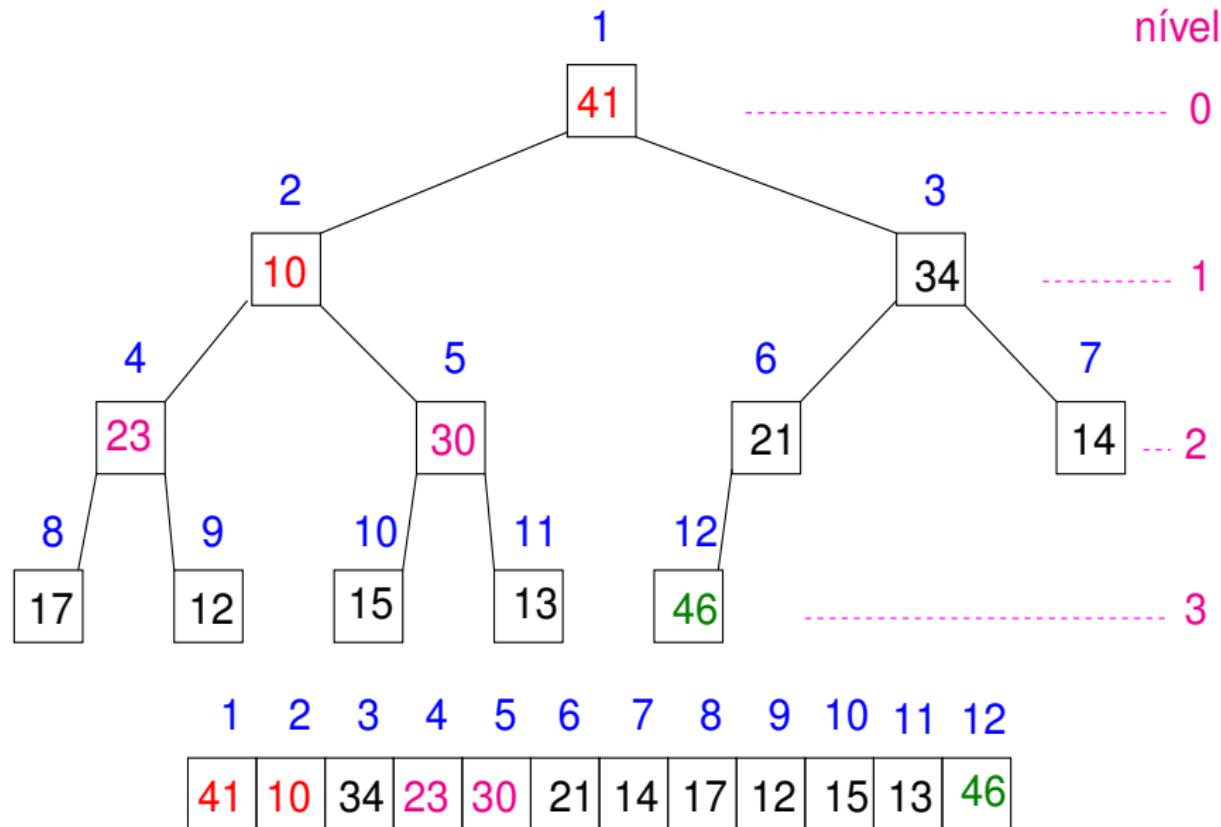
# HeapSort



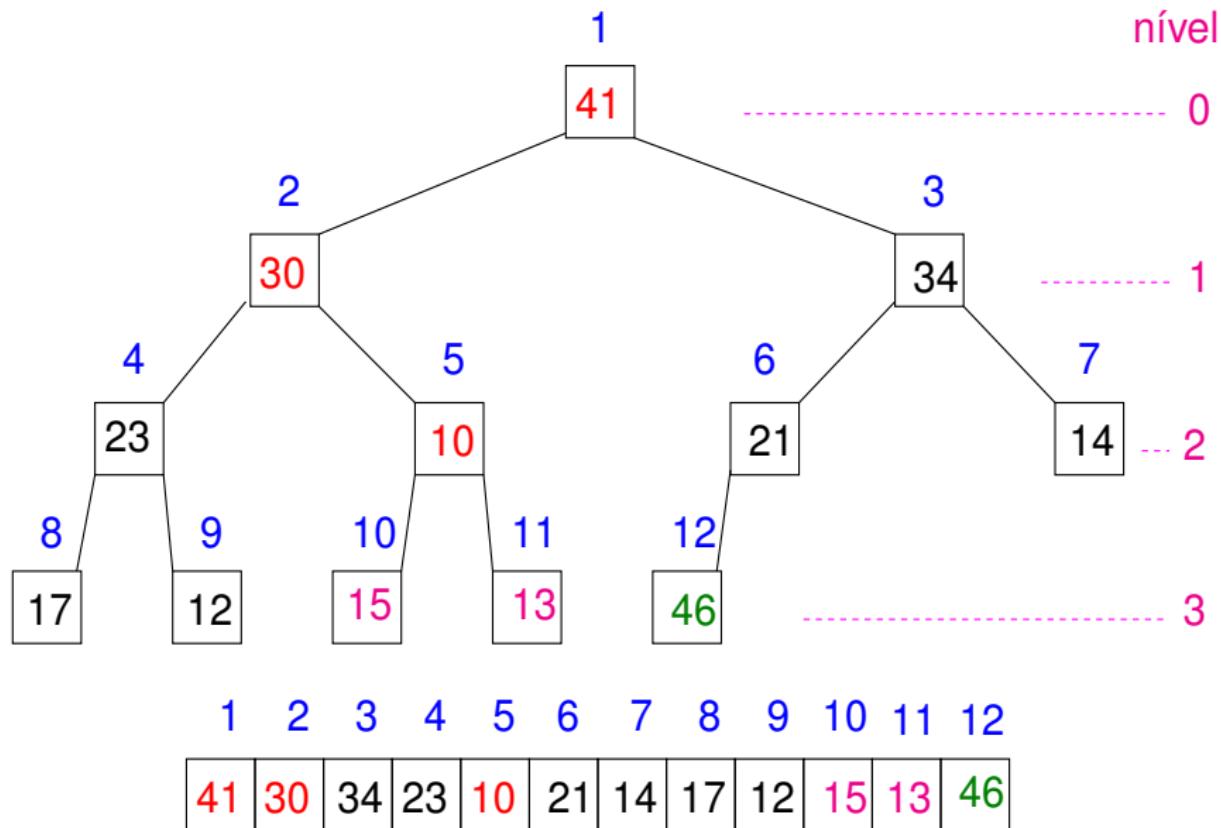
# HeapSort



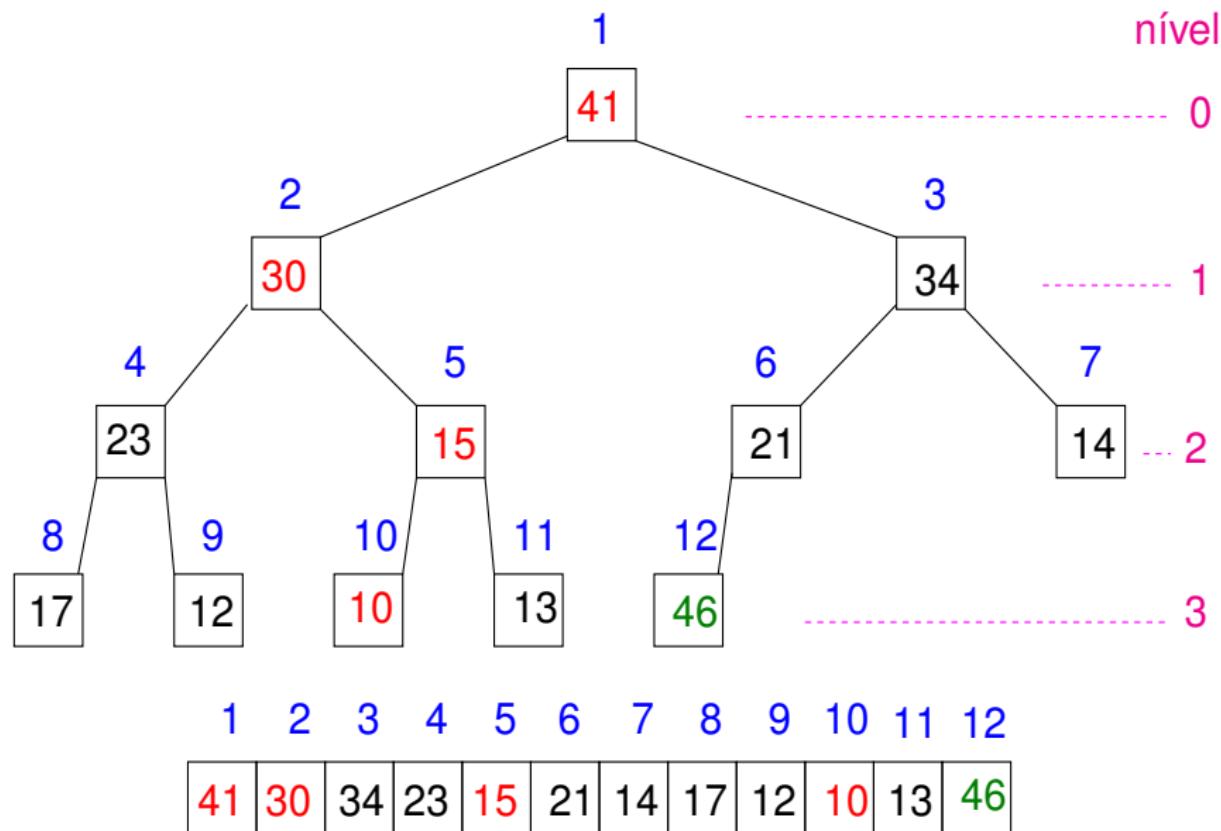
# HeapSort



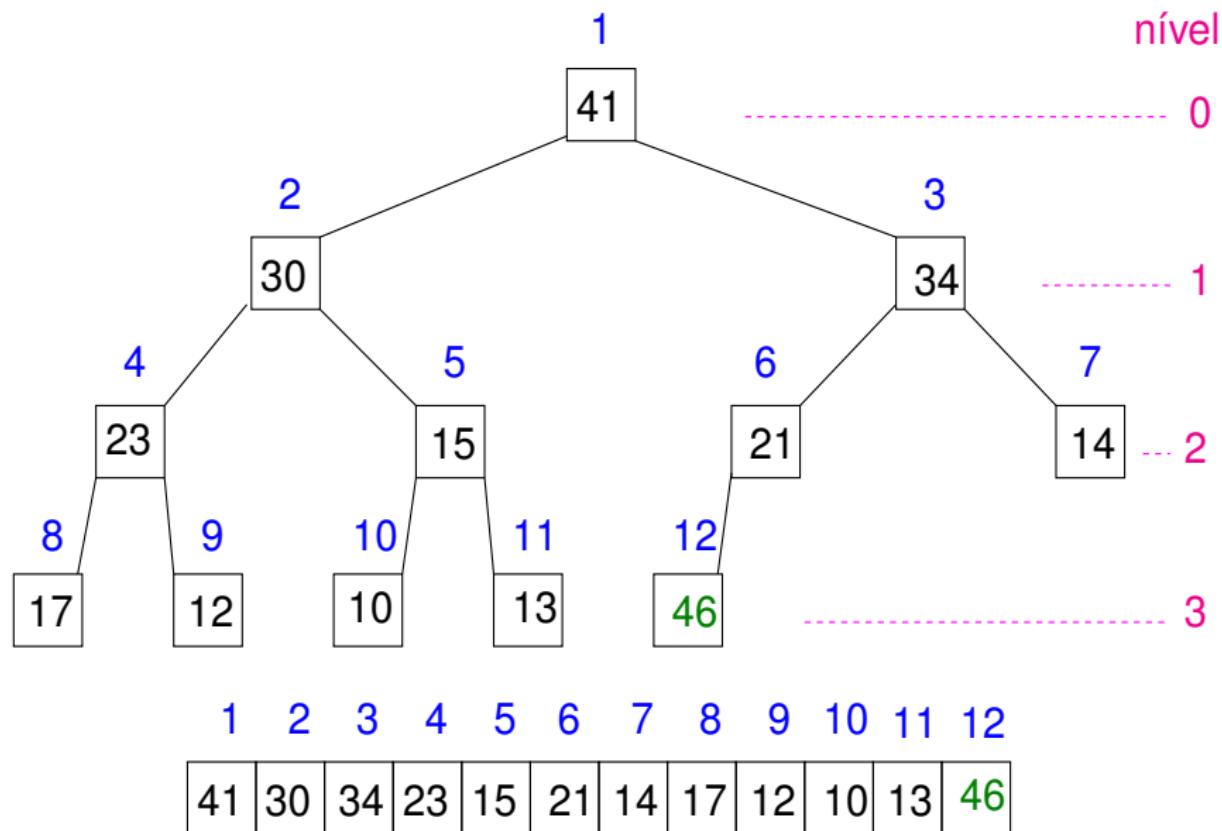
# HeapSort



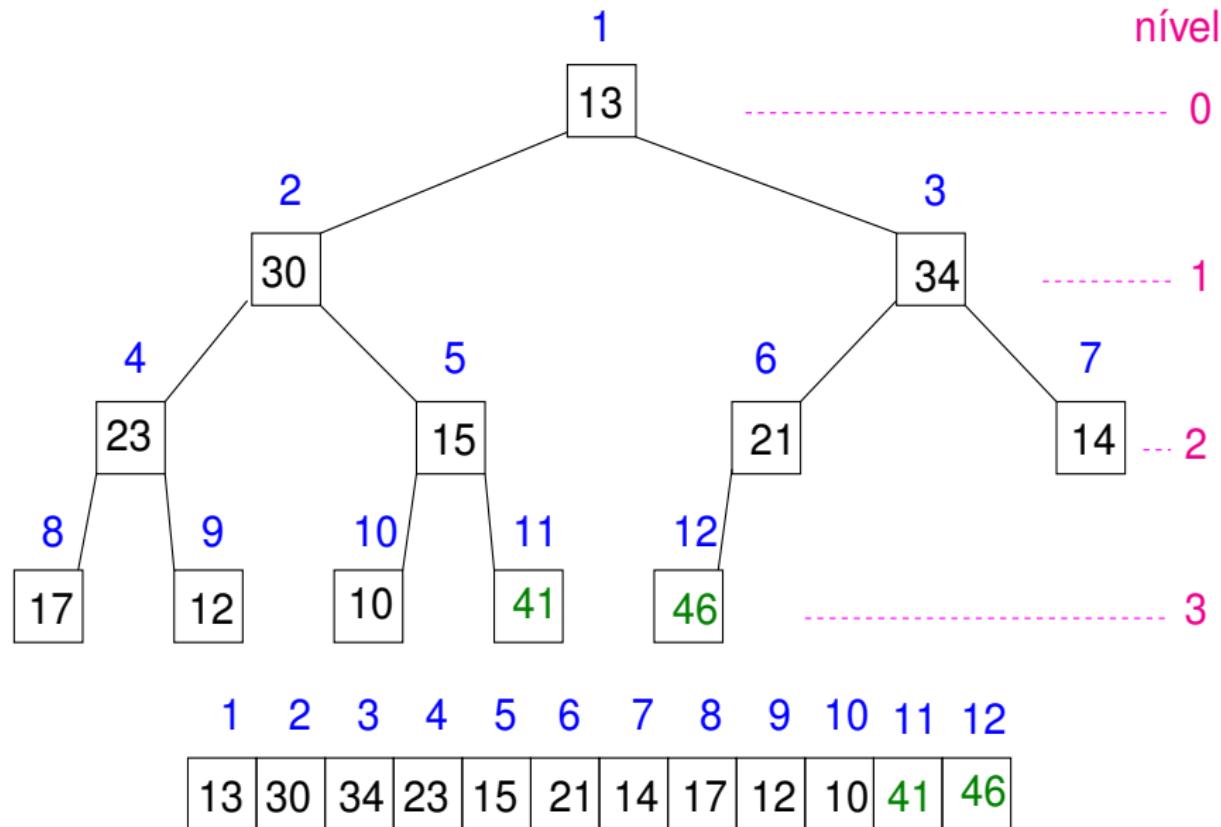
# HeapSort



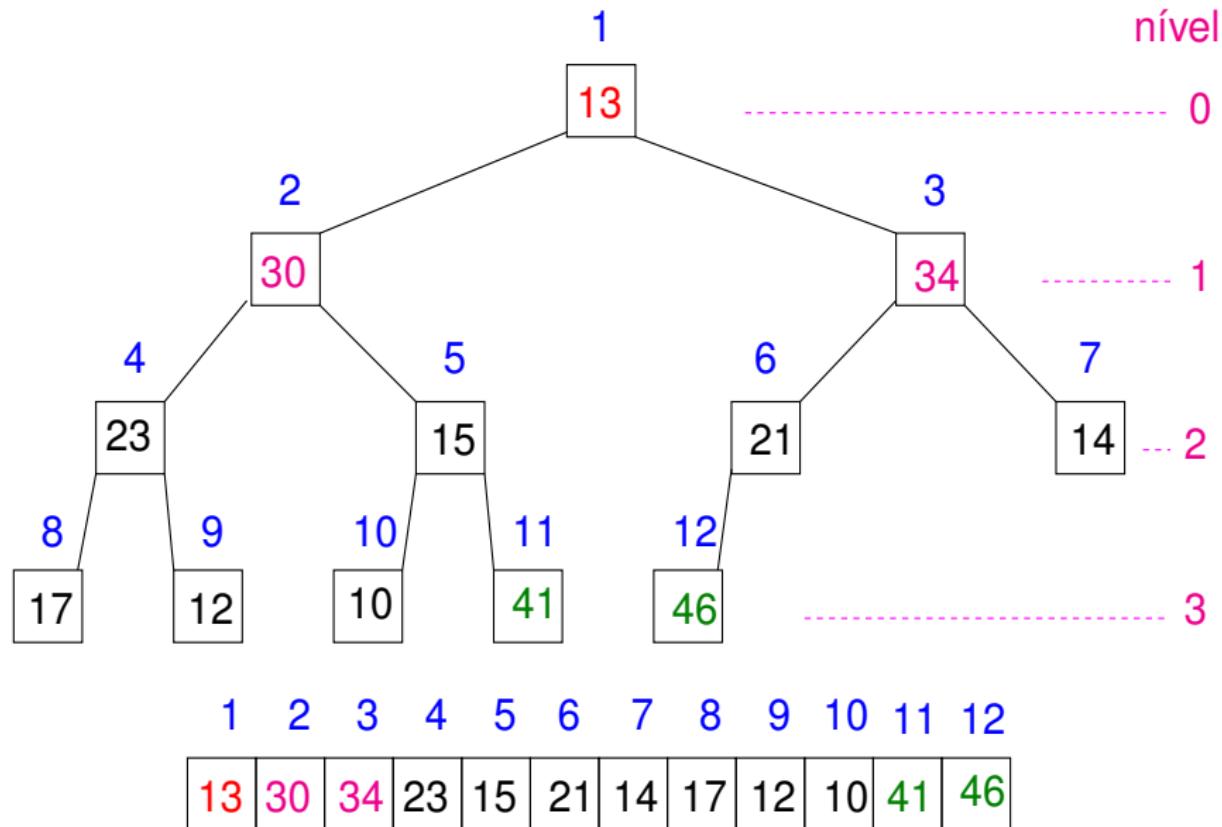
# HeapSort



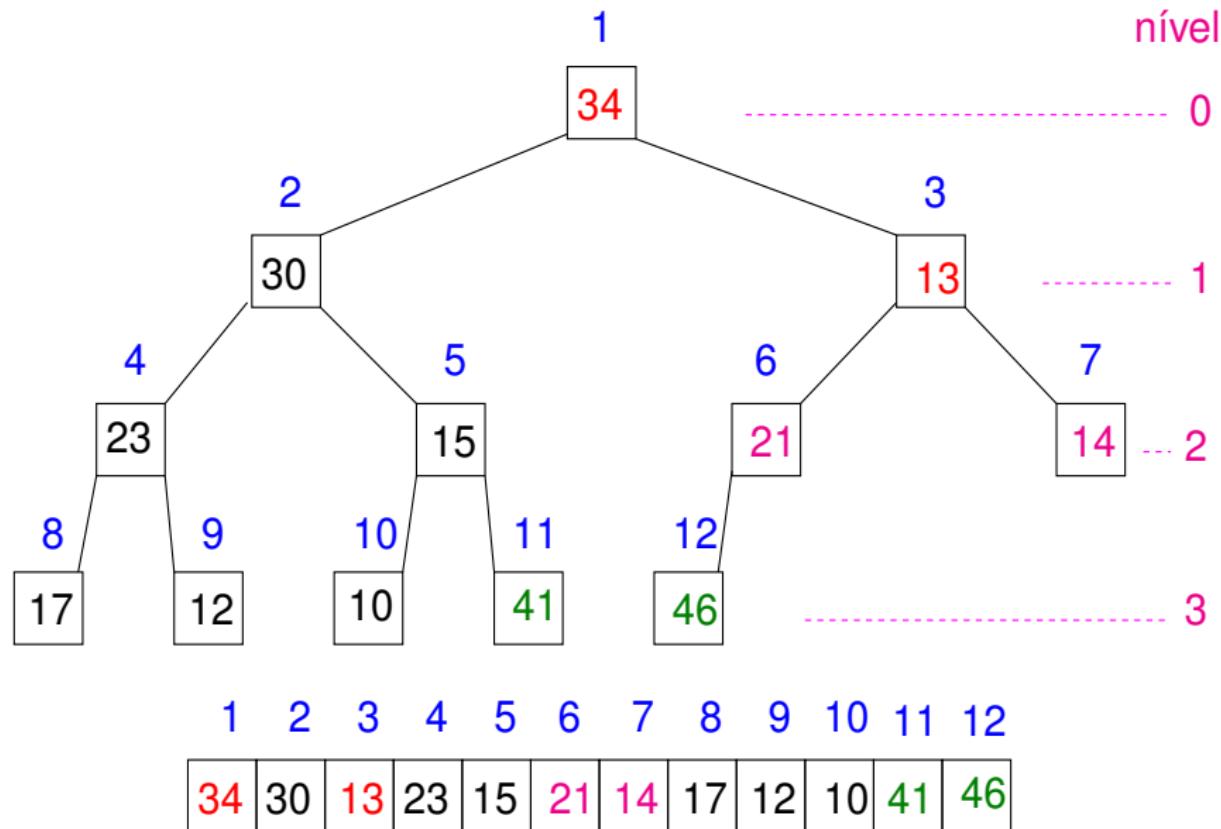
# HeapSort



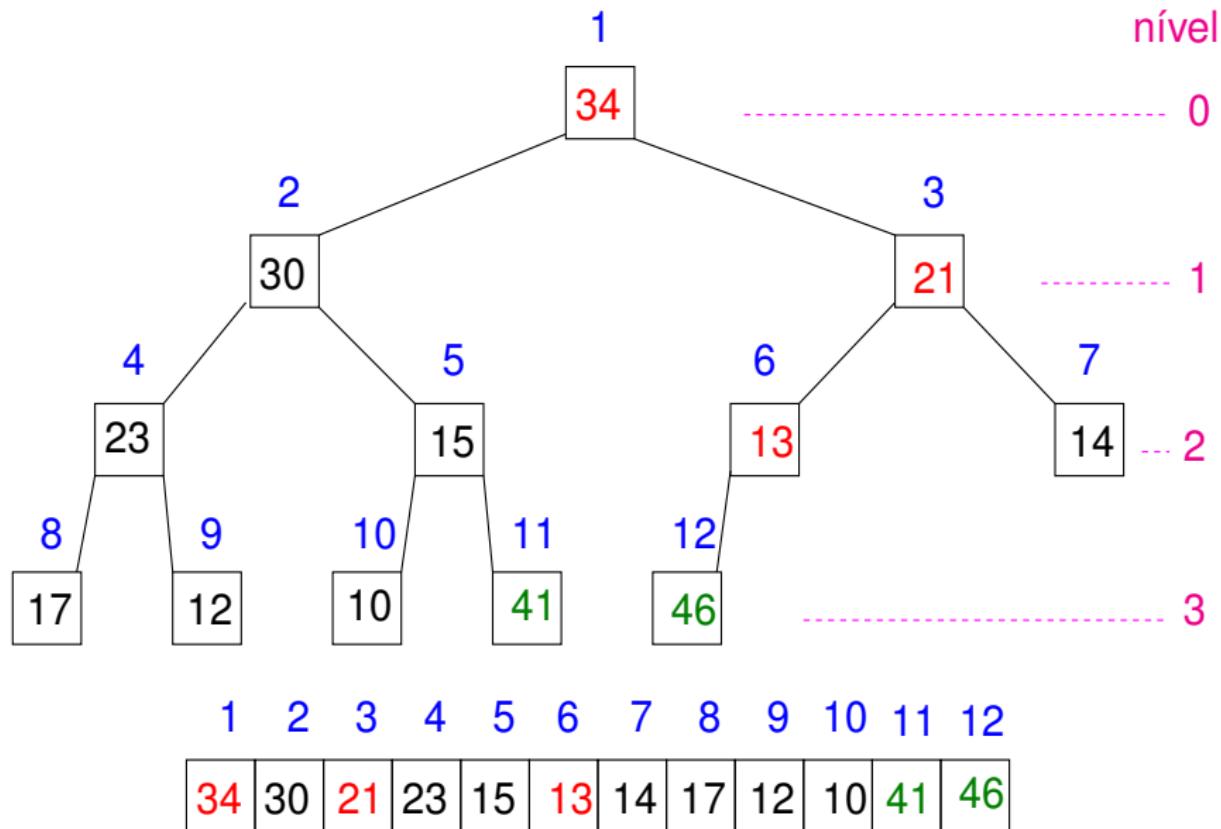
# HeapSort



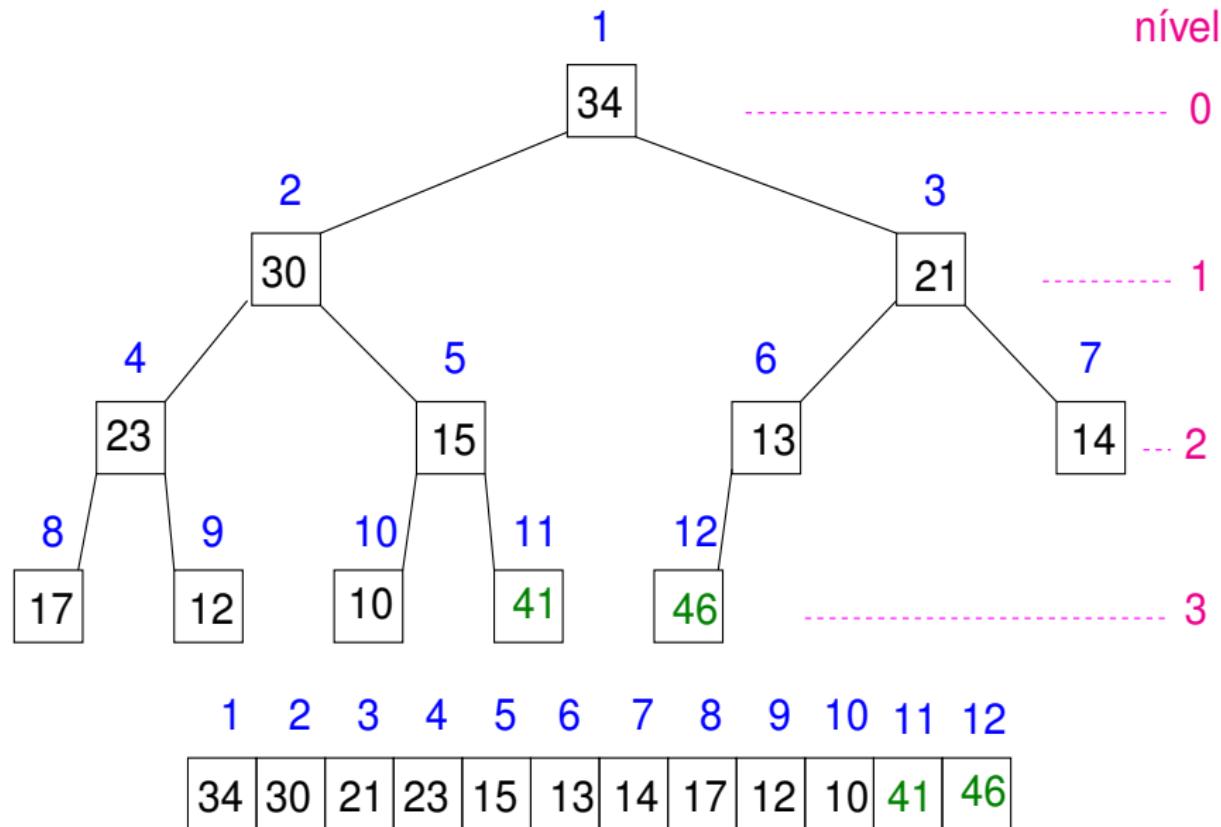
# HeapSort



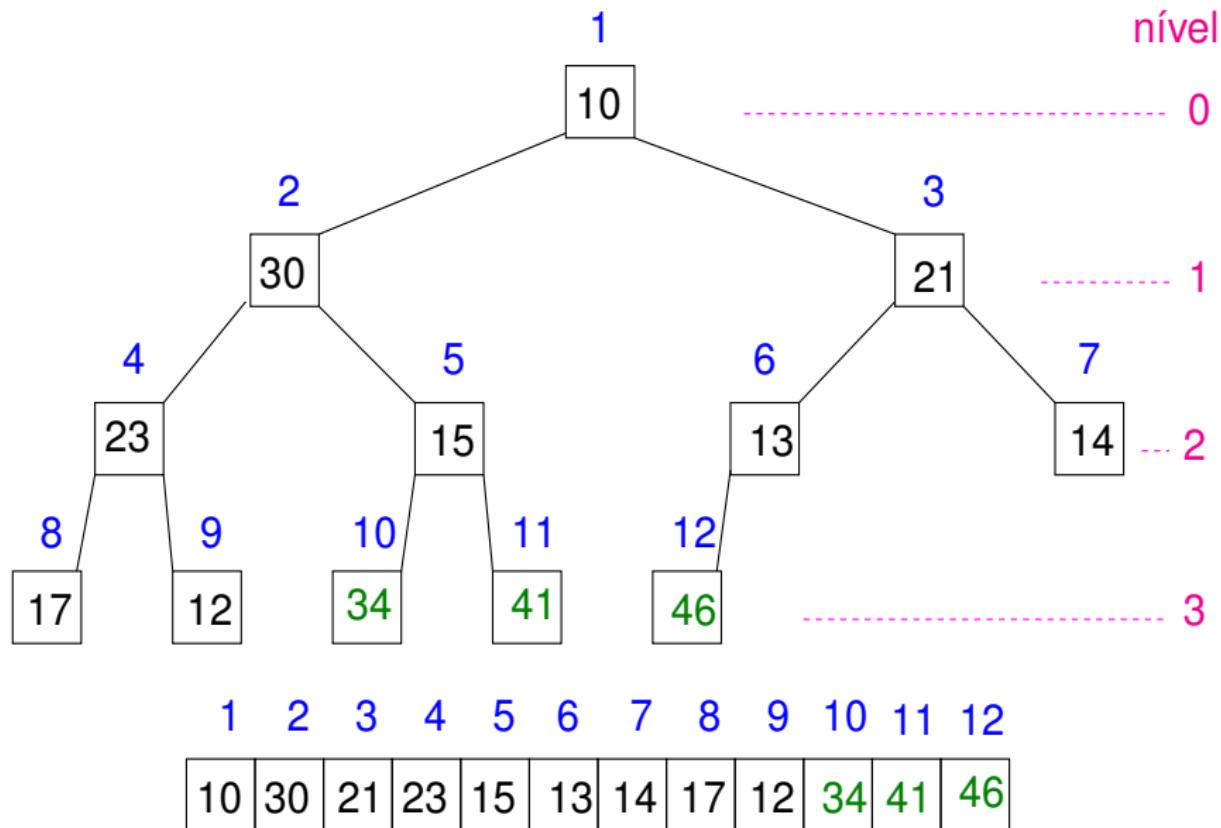
# HeapSort



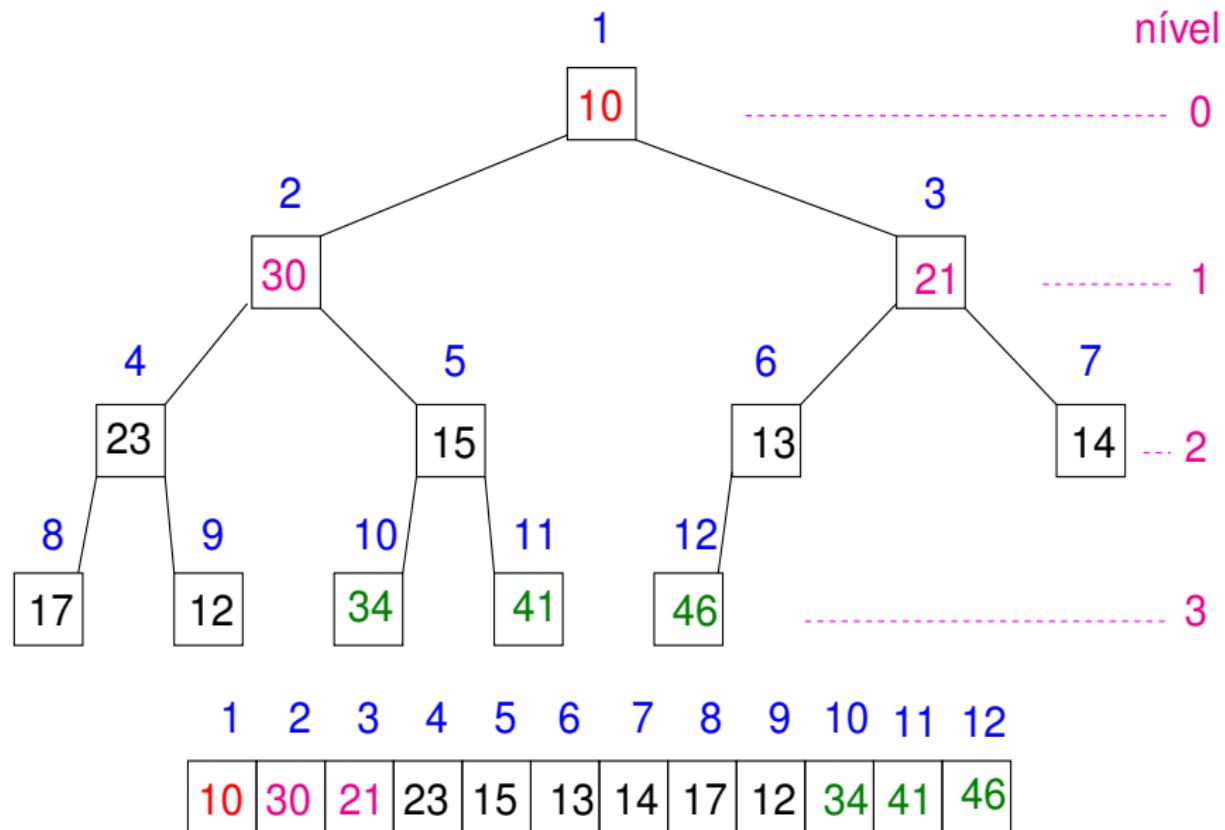
# HeapSort



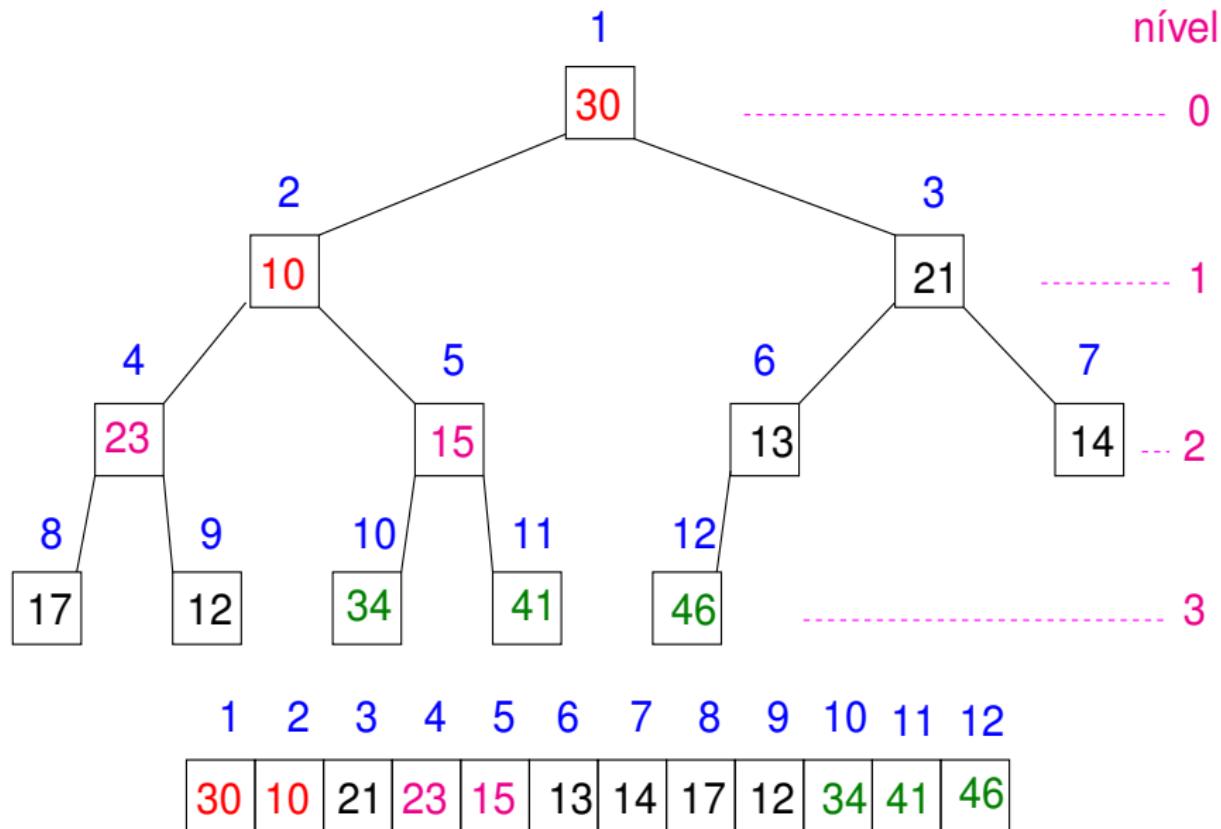
# HeapSort



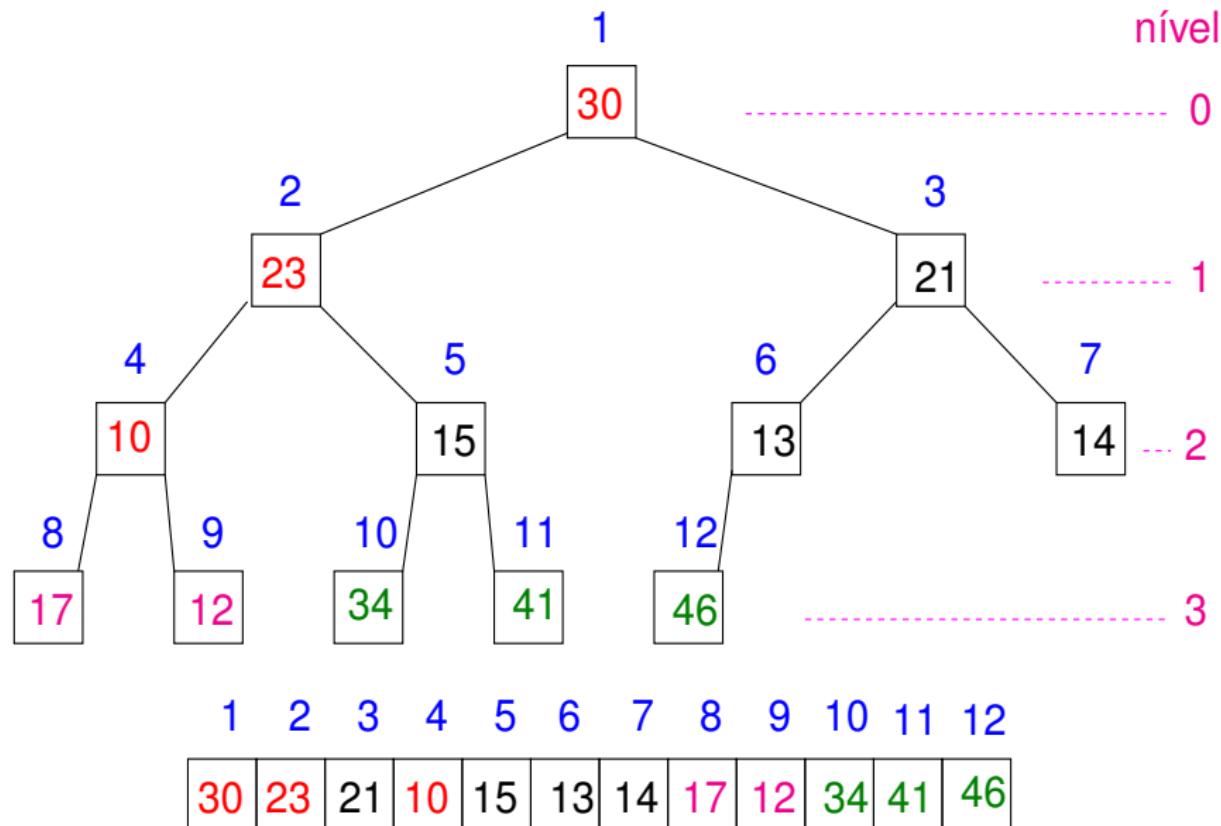
# HeapSort



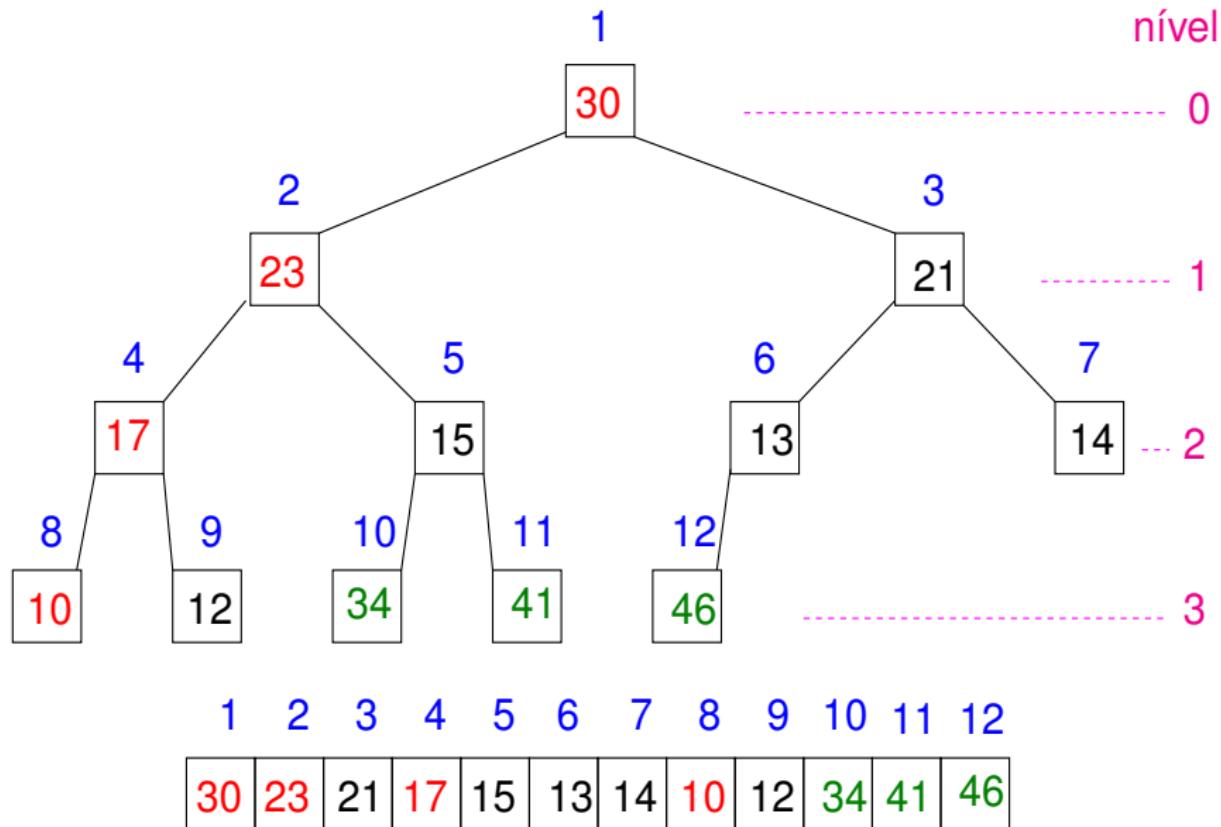
# HeapSort



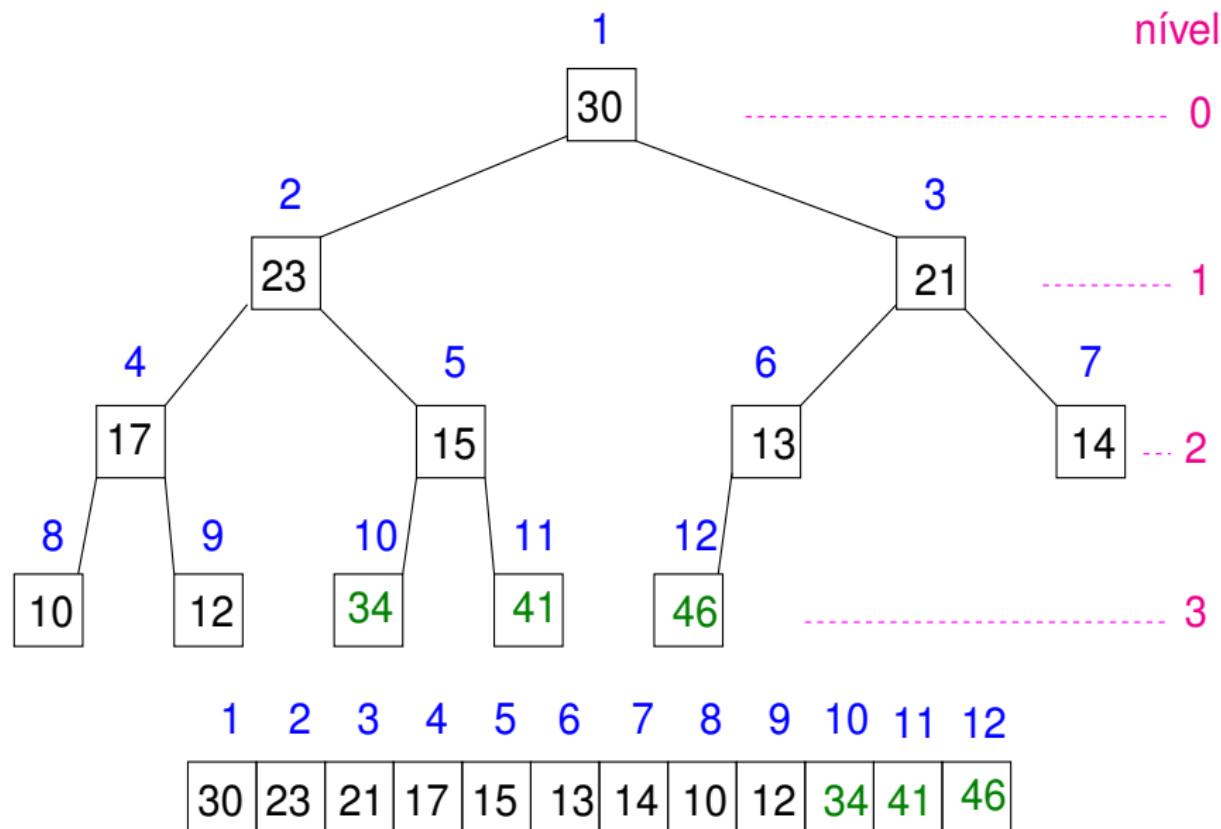
# HeapSort



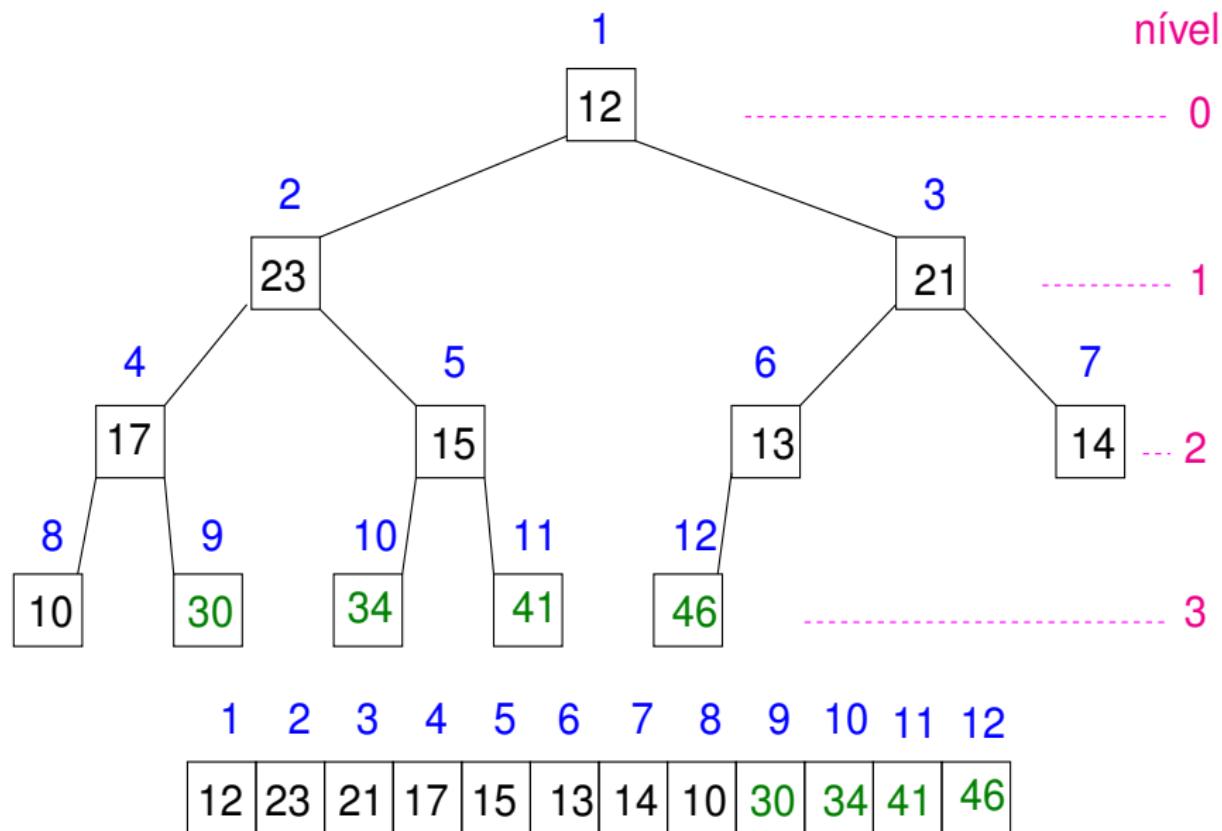
# HeapSort



# HeapSort



# HeapSort



# 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         $A[1] \leftrightarrow A[i]$
- 5         $m \leftarrow m - 1$
- 6        **MAX-HEAPIFY**( $A, m, 1$ )

Invariante:

No início de cada iteração na linha 3 vale que:

- ➊  $A[m \dots n]$  é crescente;
- ➋  $A[1 \dots m] \leq A[m + 1]$ ;
- ➌  $A[1 \dots m]$  é um max-heap.

# HeapSort

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

---

| $\text{HEAPSORT}(A, n)$                                             | Tempo |
|---------------------------------------------------------------------|-------|
| 1 $\text{BUILD-MAX-HEAP}(A, n)$                                     | ?     |
| 2 $m \leftarrow n$                                                  | ?     |
| 3 <b>para</b> $i \leftarrow n$ <b>decrescendo até</b> 2 <b>faça</b> | ?     |
| 4 $A[1] \leftrightarrow A[i]$                                       | ?     |
| 5 $m \leftarrow m - 1$                                              | ?     |
| 6 $\text{MAX-HEAPIFY}(A, m, 1)$                                     | ?     |

---

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

# HeapSort

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

---

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

---

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

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**( $A, n$ )

1 **devolva**  $A[1]$

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

**HEAP-EXTRACT-MAX**( $A, n$ )

1  $\triangleright n \geq 1$

2  $\text{max} \leftarrow A[1]$

3  $A[1] \leftarrow A[n]$

4  $\text{cor} \leftarrow n - 1$

5 **MAX-HEAPIFY**( $A, n, 1$ )

6 **devolva**  $\text{max}$

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

# Implementação com max-heap

**HEAP-INCREASE-KEY( $A, i, prior$ )**

- 1  $\triangleright$  Supõe que  $prior \geq A[i]$
- 2  $A[i] \leftarrow prior$
- 3 **enquanto**  $i > 1$  e  $A[\lfloor i/2 \rfloor] < A[i]$  **faça**
- 4      $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
- 5      $i \leftarrow \lfloor i/2 \rfloor$

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

**MAX-HEAP-INSERT( $A, n, prior$ )**

- 1  $n \leftarrow n + 1$
- 2  $A[n] \leftarrow -\infty$
- 3 **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( $A, B, n, k$ )

1 para  $i \leftarrow 0$  até  $k$  faça

2      $C[i] \leftarrow 0$

3 para  $j \leftarrow 1$  até  $n$  faça

4      $C[A[j]] \leftarrow C[A[j]] + 1$

▷  $C[i]$  é o número de  $j$ s tais que  $A[j] = i$

5 para  $i \leftarrow 1$  até  $k$  faça

6      $C[i] \leftarrow C[i] + C[i - 1]$

▷  $C[i]$  é o número de  $j$ s tais que  $A[j] \leq i$

7 para  $j \leftarrow n$  decrescendo até 1 faça

8      $B[C[A[j]]] \leftarrow A[j]$

9      $C[A[j]] \leftarrow C[A[j]] - 1$

## 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.

## Radix Sort

- Considere agora o problema de ordenar um vetor  $A[1 \dots n]$  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 | 657 | 355 | 657 |
| 720 | 329 | 457 | 720 |
| 355 | 839 | 657 | 839 |
|     | ↑   | ↑   | ↑   |

## 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 interpretamos 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  $20 \times 2^{20}$  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.

## Divisão e Conquista – Multiplicação de Inteiros

$$\begin{array}{r} 274 \\ \times 382 \\ \hline 104668 \end{array}$$

$$35871227428009 \times 11234908764388 = ?$$

## Problema

Dados números inteiros positivos  $u$  e  $v$  com  $n$  dígitos cada  
calcular o produto  $u \times v$ .

- ▶ exemplo:  $99998888 \times 77776666$
- ▶ exemplo:  $99998888 \times 00076666$
- ▶ imagine-se fazendo as contas com lápis e papel, dígito a dígito
- ▶ quanto tempo vai levar?
- ▶ tempo  $\cong$  número de operações elementares
- ▶ operação elementar = adição e multiplicação de dígitos
- ▶  $n$  grande: aplicações à criptografia, fast Fourier transform, etc.

Se  $u$  e  $v$  têm  $n$  dígitos cada então

- ▶  $u \times v$  tem no máximo  $2n$  dígitos
- ▶  $u + v$  tem no máximo  $n + 1$  dígitos

Exemplos:

- ▶  $9999 \times 9999 = 9998\,0001$
- ▶  $9999 + 9999 = 19998$

# Algoritmo de multiplicação ordinário

## Adição ordinária

$$\begin{array}{r} 9999 \\ 7777 \\ \hline 17776 \end{array} \quad \begin{array}{l} u \\ v \\ u+v \end{array}$$

- ▶ base do algoritmo: adição de dígitos
- ▶  $n$  operações elementares

## Multiplicação ordinária

$$\begin{array}{r} 9999 \\ \times 7777 \\ \hline 69993 \\ 69993 \\ 69993 \\ \hline 77762223 \end{array} \quad u \times v$$

- ▶ base do algoritmo: multiplicação e adição de dígitos
- ▶  $n^2$  operações elementares (na verdade,  $2n^2 + n$ )
- ▶  $n$  vezes mais lenta que adição
- ▶  $n^2$  é lento:  $(2n)^2 = 4n^2$  e  $(10n)^2 = 100n^2$

## Desafio:

- ▶ inventar um algoritmo de multiplicação mais rápido
- ▶ há 50 anos acreditava-se que não existe algoritmo mais rápido
- ▶ acreditava-se que  $n^2$  operações elementares são inevitáveis
- ▶ vou mostrar que  $n^{1.6}$  operações elementares são suficientes

|     | a |   |   |   | b |   |   |   |
|-----|---|---|---|---|---|---|---|---|
| $u$ | 9 | 9 | 9 | 9 | 8 | 8 | 8 | 8 |
| $v$ | 7 | 7 | 7 | 7 | 6 | 6 | 6 | 6 |
|     | c |   |   |   | d |   |   |   |

- ▶  $u = a \cdot 10^{n/2} + b$
- ▶  $v = c \cdot 10^{n/2} + d$
- ▶  $u \times v = (a \times c) \cdot 10^n + (a \times d + b \times c) \cdot 10^{n/2} + b \times d$
- ▶ estamos supondo  $n$  par

$$\begin{array}{r} 99998888 \\ \times 77776666 \\ \hline \end{array} \quad u$$

$$\begin{array}{r} 77776666 \\ \times 77776666 \\ \hline \end{array} \quad v$$

$$\begin{array}{r} 9999 \\ \times 8888 \\ \hline \end{array} \quad a$$

$$\begin{array}{r} 8888 \\ \times 7777 \\ \hline \end{array} \quad b$$

$$\begin{array}{r} 7777 \\ \times 6666 \\ \hline \end{array} \quad c$$

$$\begin{array}{r} 6666 \\ \times 6666 \\ \hline \end{array} \quad d$$

$$\begin{array}{r} 77762223 \\ \times 135775310 \\ \hline \end{array} \quad ac$$

$$\begin{array}{r} 135775310 \\ \times 59247408 \\ \hline \end{array} \quad ad + bc$$

$$\begin{array}{r} 59247408 \\ \times 59247408 \\ \hline \end{array} \quad bd$$

$$\begin{array}{r} 7777580112347408 \\ \times \phantom{1234567890} \\ \hline \end{array} \quad x$$

bom algoritmo para calculadora de bolso!

## Número de operações elementares:

- ▶  $u \times v = (a \times c) \cdot 10^n + (a \times d + b \times c) \cdot 10^{n/2} + b \times d$
- ▶ 4 multiplicações de tamanho  $n/2$  (mais 3 adições)
- ▶ operações elementares:  $4(n/2)^2 = n^2$
- ▶ não ganhamos nada...

Tente, então, repetir o truque recursivamente

Divisão e conquista ( $n$  é potência de 2)

DIVIDA-E-CONQUISTE ( $u, v, n$ )

- 1 se  $n = 1$
- 2 então devolva  $u \times v$
- 3 senão  $m \leftarrow n/2$
- 4  $a \leftarrow \lfloor u/10^m \rfloor$
- 5  $b \leftarrow u \bmod 10^m$
- 6  $c \leftarrow \lfloor v/10^m \rfloor$
- 7  $d \leftarrow v \bmod 10^m$
- 8  $ac \leftarrow$  DIVIDA-E-CONQUISTE ( $a, c, m$ )
- 9  $bd \leftarrow$  DIVIDA-E-CONQUISTE ( $b, d, m$ )
- 10  $ad \leftarrow$  DIVIDA-E-CONQUISTE ( $a, d, m$ )
- 11  $bc \leftarrow$  DIVIDA-E-CONQUISTE ( $b, c, m$ )
- 12  $x \leftarrow ac \cdot 10^{2m} + (ad + bc) \cdot 10^m + bd$
- 13 devolva  $x$

## Número $T(n)$ de operações elementares:

- ▶  $T(1) = 1$     e     $T(n) = 4 T(n/2) + n$     para  $n > 1$
- ▶ queremos uma “fórmula”
- ▶ solução da recorrência:

$$\begin{aligned}T(n) &= 4 T(n/2) + n \\&= 4 (4 T(n/4) + n/2) + n \\&= 16 T(n/4) + 3n \\&= 16 (4 T(n/8) + n/4) + 3n \\&= 64 T(n/8) + 7n \\&= (2^j)^2 T(n/2^j) + (2^j - 1)n \\&= n^2 T(n/n) + (n - 1)n \\&= 2n^2 - n\end{aligned}$$

- ▶  $T(n) = 2n^2 - n$
- ▶  $T(n)$  é proporcional a  $n^2$
- ▶ tão lento quanto o algoritmo ordinário...



# Truque de Karatsuba

Divisão e conquista:

- ▶  $u = a \cdot 10^{n/2} + b$        $v = c \cdot 10^{n/2} + d$
- ▶  $u \times v = (a \times c) \cdot 10^n + (a \times d + b \times c) \cdot 10^{n/2} + b \times d$
- ▶ 4 multiplicações de tamanho  $n/2$  (e 3 adições)

## Truque

- ▶  $y = (a + b) \times (c + d) = a \times c + a \times d + b \times c + b \times d$
- ▶  $u \times v = (a \times c) \cdot 10^n + (y - a \times c - b \times d) \cdot 10^{n/2} + b \times d$
- ▶ 3 multiplicações de tamanho  $n/2$  (e 6 adições)

Estamos supondo  $n$  par

$$\begin{array}{ll} 99998888 & u \end{array}$$

$$\begin{array}{ll} 77776666 & v \end{array}$$

$$\begin{array}{ll} 77762223 & ac \end{array}$$

$$\begin{array}{ll} 59247408 & bd \end{array}$$

$$\begin{array}{ll} 18887 & a + b \end{array}$$

$$\begin{array}{ll} 14443 & c + d \end{array}$$

$$\begin{array}{ll} 272784941 & y \end{array}$$

$$\begin{array}{ll} 135775310 & y - ac - bd \end{array}$$

$$\begin{array}{ll} 7777580112347408 & x \end{array}$$

## Número de operações elementares:

- ▶ 3 multiplicações de tamanho  $n/2$
- ▶  $3(n/2)^2 = 0.75 n^2$  operações elementares
- ▶ ganhamos 25% com *uma* aplicação do truque
- ▶ para ganhar mais, repita o truque recursivamente

## Algoritmo de Karatsuba (rascunho)

RASCUNHO-DE-KARATSUBA ( $u, v, n$ )

```
1 se $n = 1$
2 então devolva $u \times v$
3 senão $m \leftarrow n/2$
4 $a \leftarrow \lfloor u/10^m \rfloor$
5 $b \leftarrow u \bmod 10^m$
6 $c \leftarrow \lfloor v/10^m \rfloor$
7 $d \leftarrow v \bmod 10^m$
8 $ac \leftarrow \text{RASCUNHO-DE-KARATSUBA } (a, c, m)$
9 $bd \leftarrow \text{RASCUNHO-DE-KARATSUBA } (b, d, m)$
10 $y \leftarrow \text{RASCUNHO-DE-KARATSUBA } (a + b, c + d, m + 1)$
11 $x \leftarrow ac \cdot 10^{2m} + (y - ac - bd) \cdot 10^m + bd$
12 devolva x
```

## Número de operações elementares:

- ▶  $T(n) = 3 T(n/2) + n$  para  $n = 2, 4, 8, 16, \dots$  e  $T(1) = 1$
- ▶ solução da recorrência:  $T(n) = 3n^{\lg 3} - 2n$
- ▶ prova:
$$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3(3(n/2)^{\lg 3} - 2(n/2)) + n \\ &= 3(n^{\lg 3} - n) + n \\ &= 3n^{\lg 3} - 2n \end{aligned}$$
- ▶  $T(2^j) = 3 \cdot 3^j - 2 \cdot 2^j$
- ▶ conclusão:  $T(n)$  é proporcional a  $n^{\lg 3}$

| $n$  | $n^2$   | $n^{\lg 3}$ |       |
|------|---------|-------------|-------|
| 8    | 64      | 27          | 42%   |
| 16   | 256     | 81          | 32%   |
| 32   | 1024    | 243         | 24%   |
| 64   | 4096    | 729         | 18%   |
| 128  | 16384   | 2187        | 13%   |
| 256  | 65536   | 6561        | 10%   |
| 512  | 262144  | 19683       | 8%    |
| 1024 | 1048576 | 59049       | 6%    |
| 2048 | 4194304 | 177147      | 4%    |
|      | $2^j$   | $4^j$       | $3^j$ |

- ▶  $\lg 3 \approx 1.584$
- ▶  $n^{\lg 3} \approx n\sqrt{n}$
- ▶  $n^{\lg 3}$  cresce mais devagar que  $n^2$
- ▶ Karatsuba é mais rápido que o algoritmo ordinário  
(quando  $n$  é grande)

Algoritmo de Karatsuba: versão final,  $n$  arbitrárioKARATSUBA ( $u, v, n$ )

- 1 se  $n \leq 3$
- 2 então devolva  $u \times v$
- 3 senão  $m \leftarrow \lceil n/2 \rceil$
- 4  $a \leftarrow \lfloor u/10^m \rfloor$
- 5  $b \leftarrow u \bmod 10^m$
- 6  $c \leftarrow \lfloor v/10^m \rfloor$
- 7  $d \leftarrow v \bmod 10^m$
- 8  $ac \leftarrow \text{KARATSUBA } (a, c, m)$
- 9  $bd \leftarrow \text{KARATSUBA } (b, d, m)$
- 10  $y \leftarrow \text{KARATSUBA } (a + b, c + d, m + 1)$
- 11  $x \leftarrow ac \cdot 10^{2m} + (y - ac - bd) \cdot 10^m + bd$
- 12 devolva  $x$

Linha 10:  $m + 1 < n$  pois  $n > 3$

Testes de Liu, Huang, Lei:

| $n$  | milissegundos |           |      |
|------|---------------|-----------|------|
|      | ordinário     | Karatsuba |      |
| 8    | 0.000         | 0.008     | —    |
| 16   | 0.002         | 0.010     | 500% |
| 32   | 0.007         | 0.019     | 271% |
| 64   | 0.027         | 0.045     | 167% |
| 128  | 0.101         | 0.128     | 127% |
| 256  | 0.388         | 0.368     | 95%  |
| 512  | 1.541         | 1.111     | 72%  |
| 1024 | 6.079         | 3.400     | 56%  |
| 2048 | 24.460        | 12.000    | 49%  |

## Quem é/foi Karatsuba?

- ▶ A.A. Karatsuba (1937–2008), matemático russo [▶ homepage](#)
  - ▶ Karatsuba descobriu o algoritmo quando tinha 23 anos
  - ▶ o artigo foi publicado em 1962 sob os nomes de Karatsuba e Ofman
- 
- <http://www.mi.ras.ru/~karatsuba/index.html>

## O algoritmo de Karatsuba é o mais rápido?

- ▶ existem algoritmos de multiplicação ainda mais rápidos
- ▶ algoritmo Toom-Cook:  $n^{1.465}$
- ▶ algoritmo Schönhage-Strassen:  $n \lg n \lg \lg n$

# Conclusão

- ▶ nem sempre um algoritmo óbvio é o melhor
- ▶ algoritmos sofisticados podem ser mais rápidos (para  $n$  grande)
- ▶ matemática ajuda a projetar algoritmos mais sofisticados
- ▶ o poder do método de divisão-e-conquista
- ▶ o poder da recursão
- ▶ é útil saber resolver recorrências

Próximo passo: multiplicação rápida de matrizes

## Referências

- ▶ Knuth, *The Art of Computer Programming*, 1997
  - ▶ Brassard, and Bratley, *Algorithmics: Theory and Practice*, 1988
  - ▶ Dasgupta, Papadimitriou, and Vazirani, *Algorithms*, 2006
  - ▶ Kleinberg, and Tardos, *Algorithm Design*, 2005
  - ▶ *Multiplication algorithm* [▶ Wikipedia](#)
  - ▶ *Karatsuba algorithm* [▶ Wikipedia](#)
- 
- [https://en.wikipedia.org/wiki/Karatsuba\\_algorithm](https://en.wikipedia.org/wiki/Karatsuba_algorithm)