Tópico 5
Algoritmos de Ordenação

Parte I - métodos de ordenação: inserção, mergesort, heapsort e quicksort.

Problema Computacional: Ordenação

- Problema computacional que surge em diversas situações.
- Definição:
  - Input: Uma sequência de n números:$\{a_1\ a_2, a_3, \ldots, a_n\}$
  - Output: Uma reordenação da sequência em $\{a'_1, a'_2, a'_3 \ldots a'_n\}$ tal que $a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n$

Insertion sort

```
INSERTION-SORT (A, n)
for j ←2 to n
        do key ← A[ j]
                i ← j –1
                while i > 0 and A[i] > key
                        do A[i+1] ← A[i]
                        i ← i –1
                A[i+1] = key
```

Key←12

i j

1	2	3	4	5	6	7	8	9	10
81	12	44	23	67	90	15	9	22	50

i j

Key←12

0	1	2	3	4	5	6	7	8	9	10
	81	81	44	23	67	90	15	9	22	50

i j

Key←12

0	1	2	3	4	5	6	7	8	9	10
	12	81	44	23	67	90	15	9	22	50

Insertion sort

```
INSERTION-SORT (A, n)
for j ←2 to n
     do key ← A[ j]
          i ← j –1
          while i > 0 and A[i] > key
                 do A[i+1] ← A[i]
                 i ← i –1
          A[i+1] = key
```

i j

Key←44

0	1	2	3	4	5	6	7	8	9	10
	12	81	44	23	67	90	15	9	22	50

i j

Key←44

0	1	2	3	4	5	6	7	8	9	10
	12	81	81	23	67	90	15	9	22	50

i j

Key←44

0	1	2	3	4	5	6	7	8	9	10
	12	44	81	23	67	90	15	9	22	50

Insertion sort

```
INSERTION-SORT (A, n)
for j ←2 to n
        do key ← A[ j]
                i ← j –1
                while i > 0 and A[i] > key
                        do A[i+1] ← A[i]
                        i ← i –1
                A[i+1] = key
```

Key←23

			i	j						
0	1	2	3	4	5	6	7	8	9	10
	12	44	81	23	67	90	15	9	22	50

Key←23

		i		j						
0	1	2	3	4	5	6	7	8	9	10
	12	44	81	81	67	90	15	9	22	50

Key←23

	i			j						
0	1	2	3	4	5	6	7	8	9	10
	12	44	44	81	67	90	15	9	22	50

Insertion sort

i j Key←23

0	1	2	3	4	5	6	7	8	9	10
	12	23	44	81	67	90	15	9	22	50

```
INSERTION-SORT (A, n)
for j ←2 to n
        do key ← A[ j]
                i ← j –1
                while i > 0 and A[i] > key
                        do A[i+1] ← A[i]
                        i ← i –1
                A[i+1] = key
```

i j Key←67

0	1	2	3	4	5	6	7	8	9	10
	12	23	44	81	67	90	15	9	22	50

i j Key←67

0	1	2	3	4	5	6	7	8	9	10
	12	23	44	67	81	90	15	9	22	50

Analisando Insertion sort

INSERTION-SORT (A, n)

1.	for j $\leftarrow$ 2 to n	$\Theta(n)$
2.	do key $\leftarrow$ A[j]	$\Theta(n)$
3.	i $\leftarrow$ j –1	$\Theta(n)$
4.	while i > 0 and A[i] > key	$n\mathbf{O}(n)=\mathbf{O}(n^2)$
5.	do A[i+1] $\leftarrow$ A[i]	$n\mathbf{O}(n)=\mathbf{O}(n^2)$
6.	i $\leftarrow$ i –1	$n\mathbf{O}(n) =\mathbf{O}(n^2)$
7.	A[i+1] = key	$\Theta(n)$

Tempo total de execução para entrada de tamanho n:

$\mathbf{O}(n^2)$ Prove!!!

Analisando Insertion sort

INSERTION-SORT (A, n)	Custo	Número de execuções
1.for j ←2 to n	c1	n
2. do key ← A[j]	c2	$n-1$
3. i ← j –1	c3	$n-1$
4. while i > 0 and A[i] > key	c4	$\sum_{j=2}^{n} t_j$
5. do A[i+1] ← A[i]	c5	$\sum_{j=2}^{n} (t_j - 1)$
6. i ← i –1	c6	$\sum_{j=2}^{n} (t_j - 1)$
7. A[i+1] = key	c7	$n-1$

Tempo total de execução para entrada de tamanho n:

$$T(n) = c_1 n + c2(n-1) + c3(n-1) + \\ + c4 \sum_{j=2}^{n} t_j + c5 \sum_{j=2}^{n} (t_j - 1) + c6 \sum_{j=2}^{n} (t_j - 1) + \\ c7(n-1)$$

PIOR CASO!!

i j Key←37

0	1	2	3	4	5	6	7	8	9	10
	42	53	74	81	37	30	22	15	9	1

i j Key←37

0	1	2	3	4	5	6	7	8	9	10
	42	53	74	81	81	30	22	15	9	1

i j Key←37

0	1	2	3	4	5	6	7	8	9	10
	42	53	74	74	81	30	22	15	9	1

INSERTION-SORT (A, n)

...

do A[i+1] ← A[i]

i ← i –1

...

PIOR CASO!!

INSERTION-SORT (A, n)
...
do A[i+1] ← A[i]
   i ← i –1
...

i j Key←37

0	1	2	3	4	5	6	7	8	9	10
	42	53	53	74	81	30	22	15	9	1

i j Key←37

0	1	2	3	4	5	6	7	8	9	10
	42	42	53	74	81	30	22	15	9	1

i j A[i+1] =key

0	1	2	3	4	5	6	7	8	9	10
	37	42	53	74	81	30	22	15	9	1

Analisando Insertion sort

- **Tempo total de execução para o pior caso - Vetor ordenado em ordem oposta ao ordenamento proposto.**

$$T(n) = c_1 n + c2(n-1) + c3(n-1) +$$
$$+ c4 \sum_{j=2}^{n} t_j + c5 \sum_{j=2}^{n} (t_j - 1) + c6 \sum_{j=2}^{n} (t_j - 1) +$$
$$c7(n-1)$$

$$t_j = j$$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = 2 + 3 + 4 + 5 + 6 \ldots n = \frac{n(n+1)}{2} -1$$

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) +$$

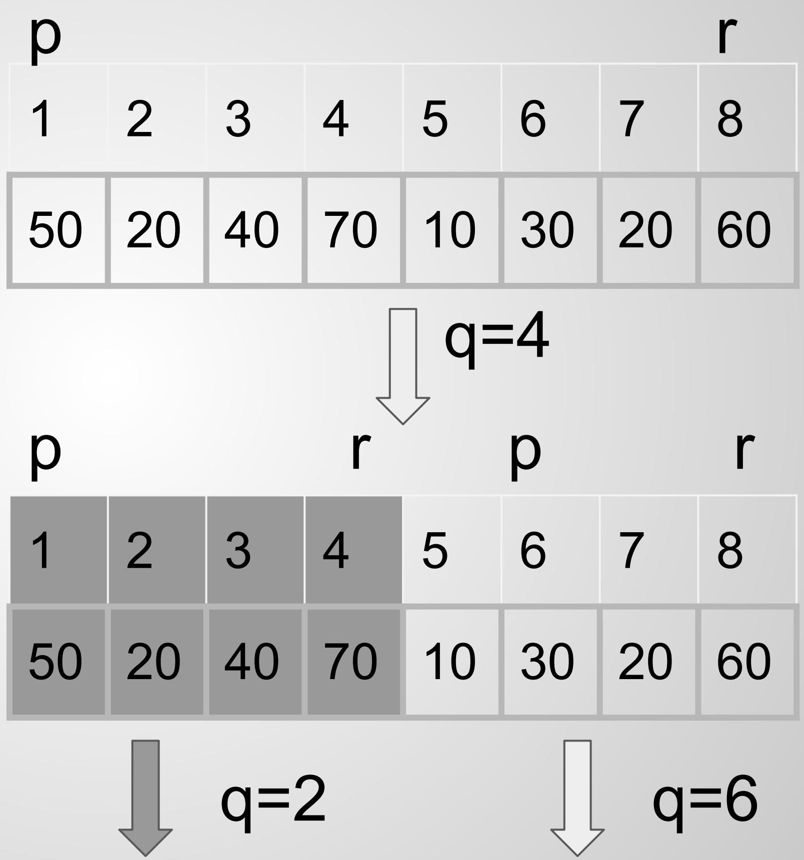
$$+ c_4 \frac{n(n+1))}{2} + + c_5 \frac{n(n-1)}{2} + c_6 \frac{n(n-1)}{2} + c_7(n-1)$$

$$T(n) = (\frac{c_4 + c_5 + c_6}{2}) n^2 + [c_1 + c_2 + c_3 + c_7 + \frac{(c_4 - c_5 - c_6)}{2}] n$$

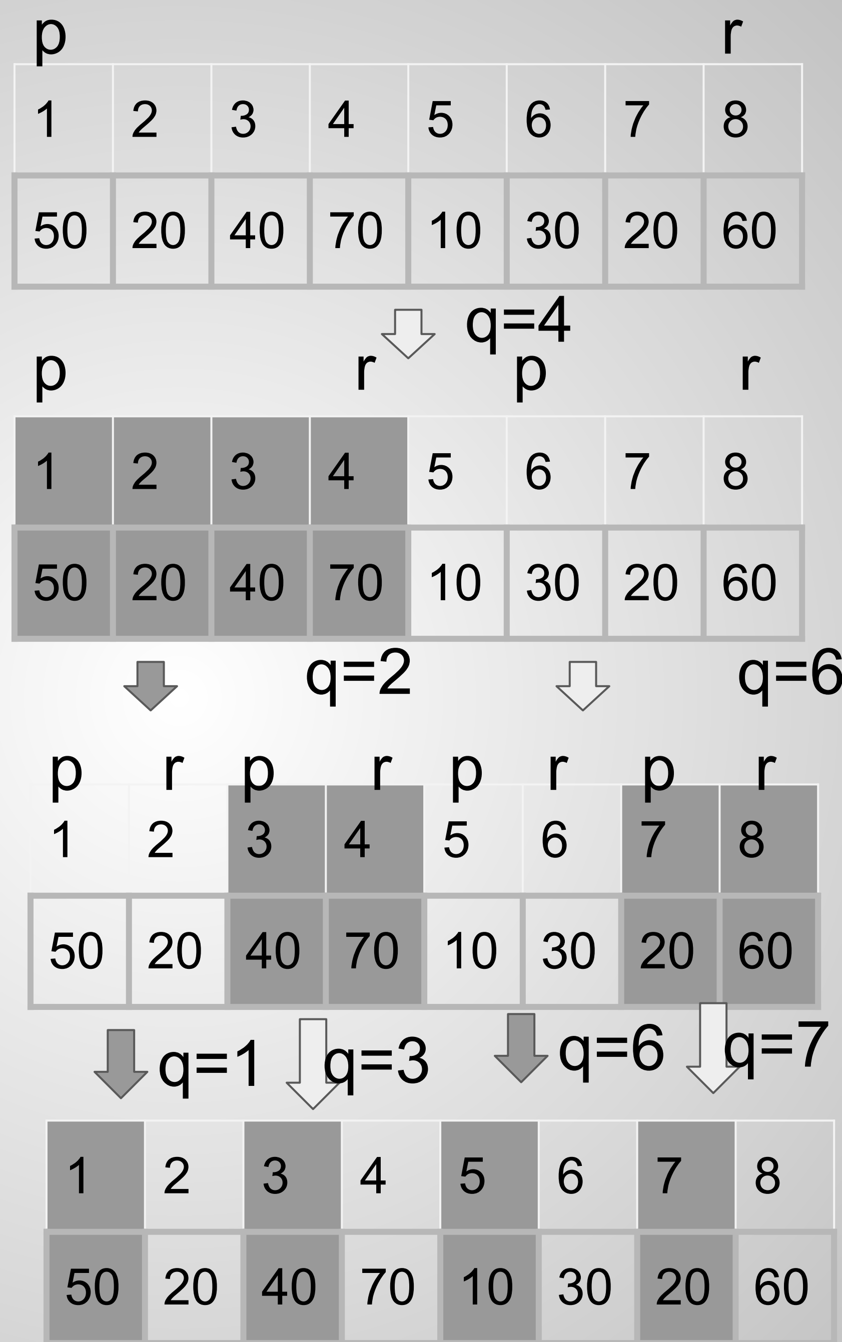
$$- (c_2 + c_3 + c_7)$$

Merge sort

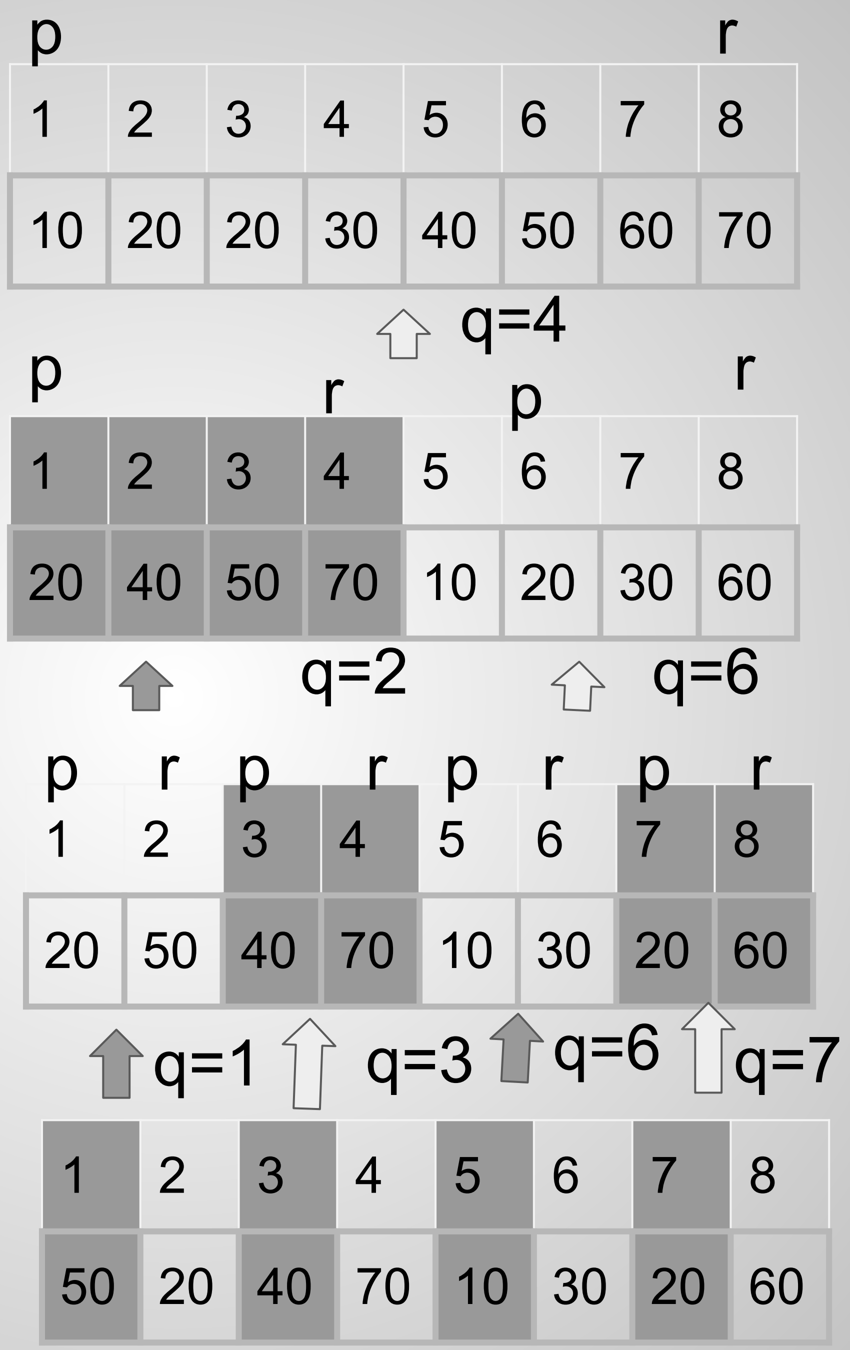
```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```



```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

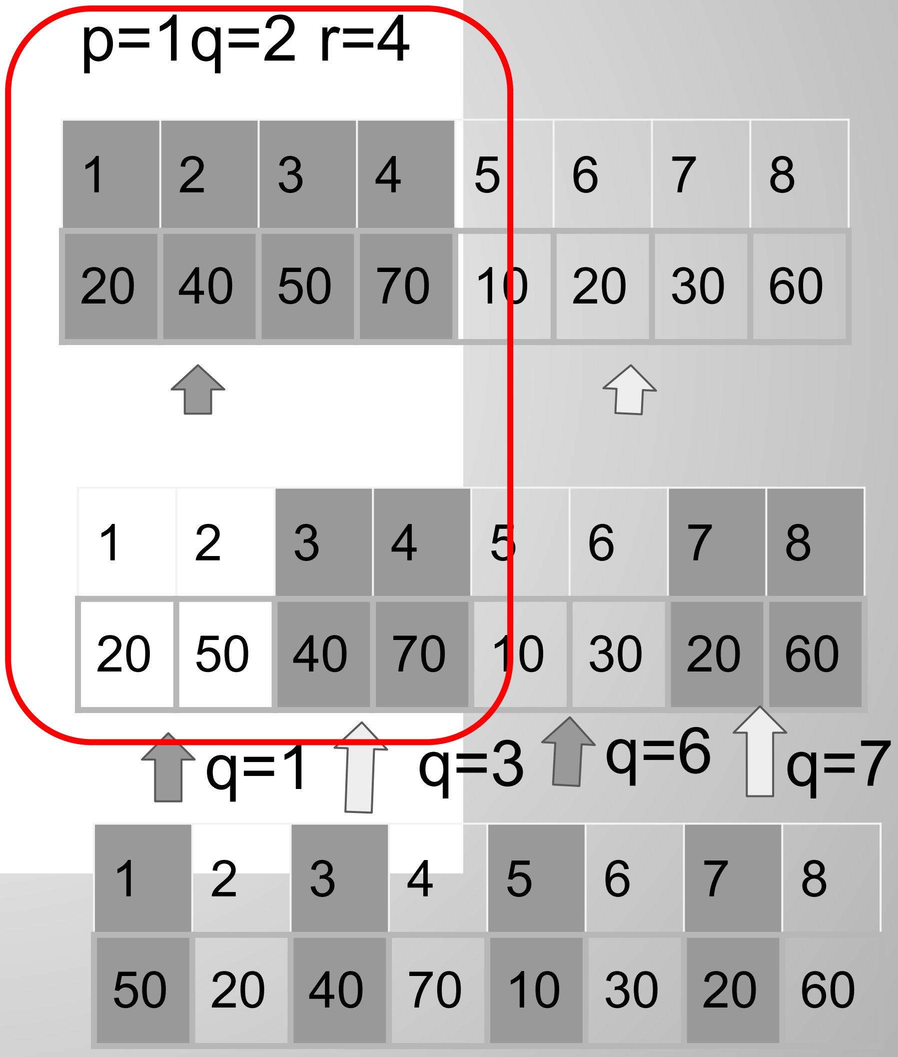


```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```



MERGE(*A*, *p*, *q*, *r*)

```
n₁ = q − p + 1
n₂ = r − q
let L[1 . . n₁ + 1] and R[1 . . n₂ + 1] be new arrays
for i = 1 to n₁
        L[i] = A[p + i − 1]
for j = 1 to n₂
        R[j] = A[q + j]
L[n₁ + 1] = ∞
R[n₂ + 1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
              A[k] = L[i]
              i = i + 1
        else A[k] = R[j]
              j = j + 1
```



Analisando Merge sort

- T (n) : tempo no pior caso (n = r − p + 1).

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

D(n): Tempo para dividir o problema

C(n): Tempo para combinar o problema

- Relação de recorrência:
  1. $T(1) = \Theta(1)$ n=1.
  2. $T(n) = 2.T(n/2)+D(n)+C(n)$ n>1.

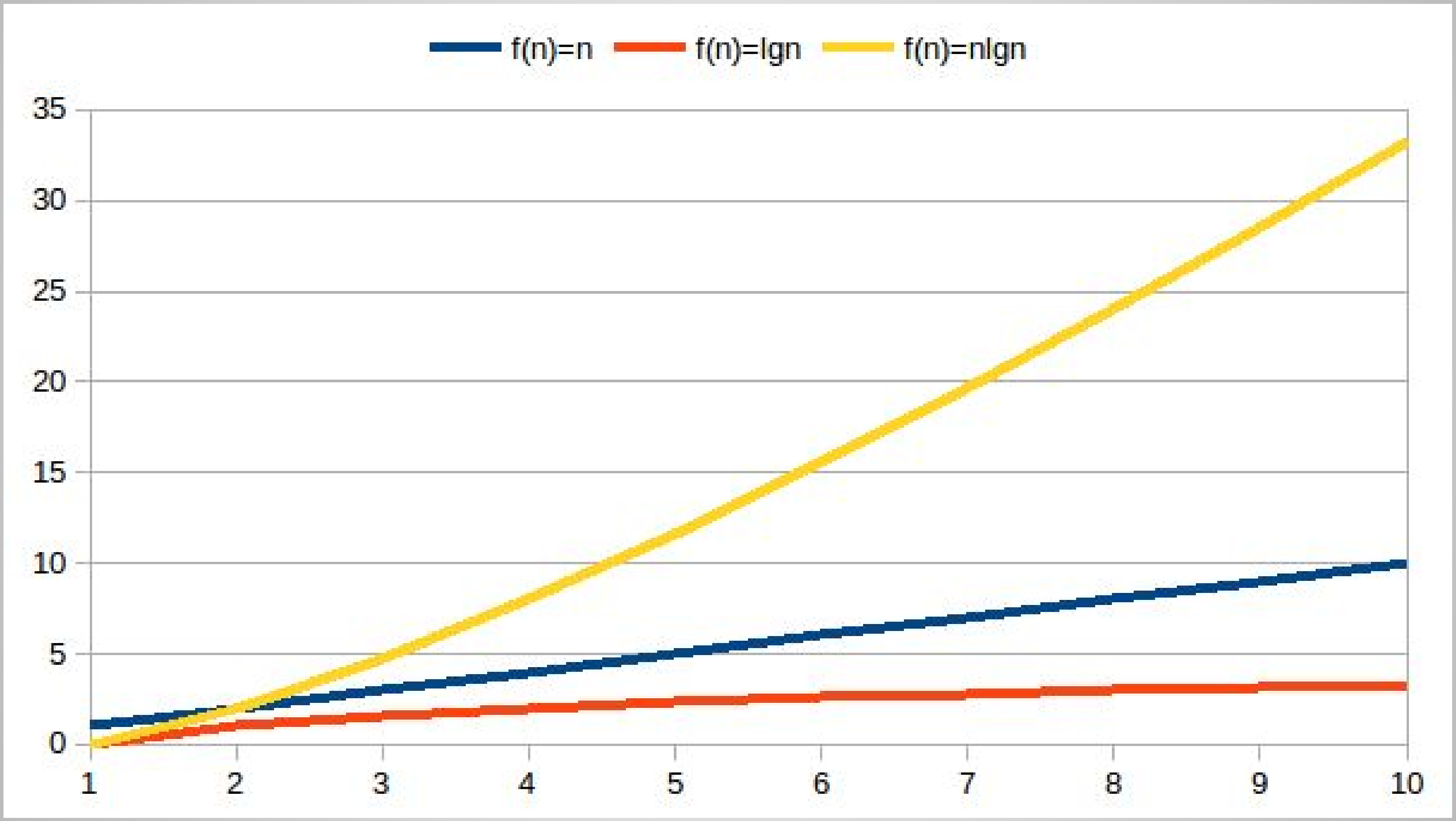
Analisando Merge sort

- Relação de recorrência:
  1. $T(n) = \Theta(1)$ $\quad$ n=1.
  2. $T(n) = 2.T(n/2)+D(n) + C(n)$ $\quad$ n>1.

$$T(n) = \begin{cases} \Theta(1) & n=1 \\ 2T(n/2) + \Theta(n) & n>1 \end{cases}$$

Usando árvore de recorrência e método da substituição, ou o teorema mestre, provamos que

T(n) = O(nlg)


f(n)=n
f(n)=lgn
f(n)=nlgn
35
30
25
20
15
10
5
0
1
2
3
4
5
6
7
8
9
10

Heaps

- Uma estrutura de dados heap (binária) é um vetor de objetos que pode ser vista como uma árvore binária “aproximadamente” completa.

- A árvore é completamente preenchida em todos os níveis, exceto possivelmente pelo nível mais baixo que é preenchido a partir da esquerda até um certo ponto da estrutura.

Heaps

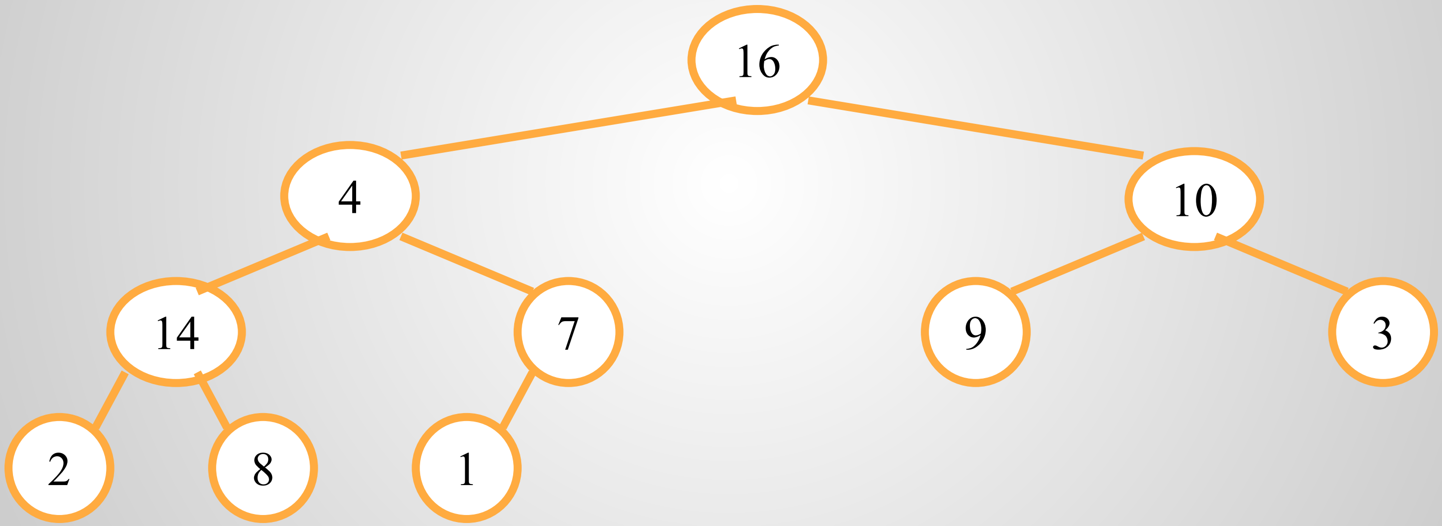
- Podemos representar uma heap como vetor, calculando os índices que ligam nós pais e filhos esquerdo e direito:

```
Parent(i)
1.return ⌊i/2⌋
```

```
Left(i)
1.return 2i
```

```
Right(i)
1.return 2i+1
```

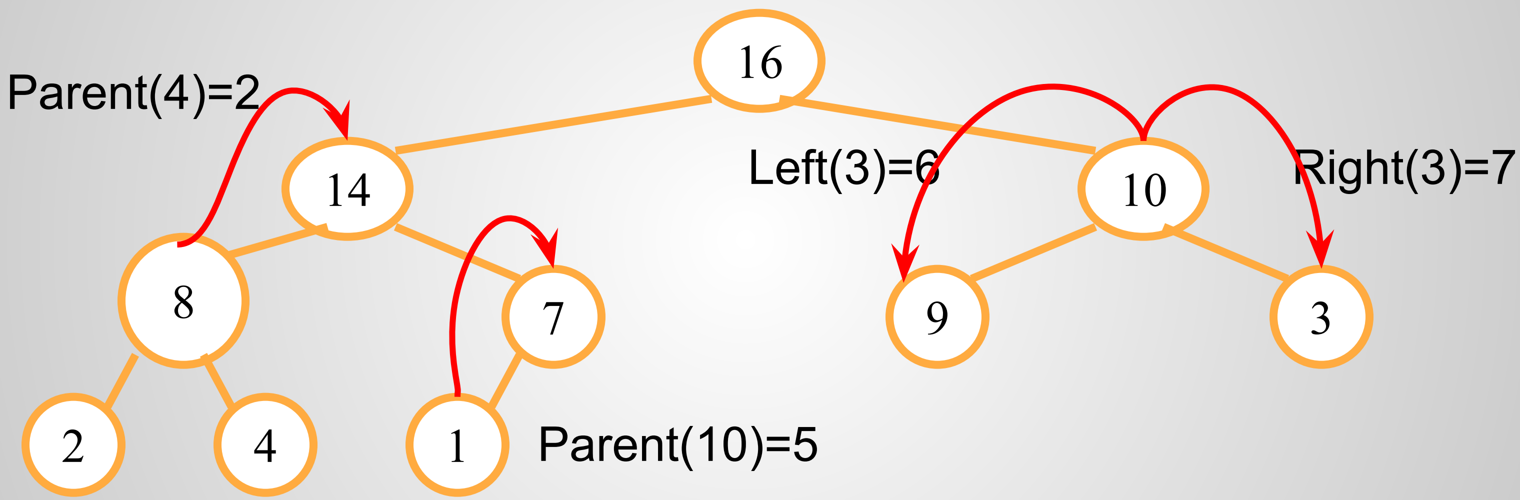
Heaps



A =

16	4	10	14	7	9	3	2	8	1

Heaps



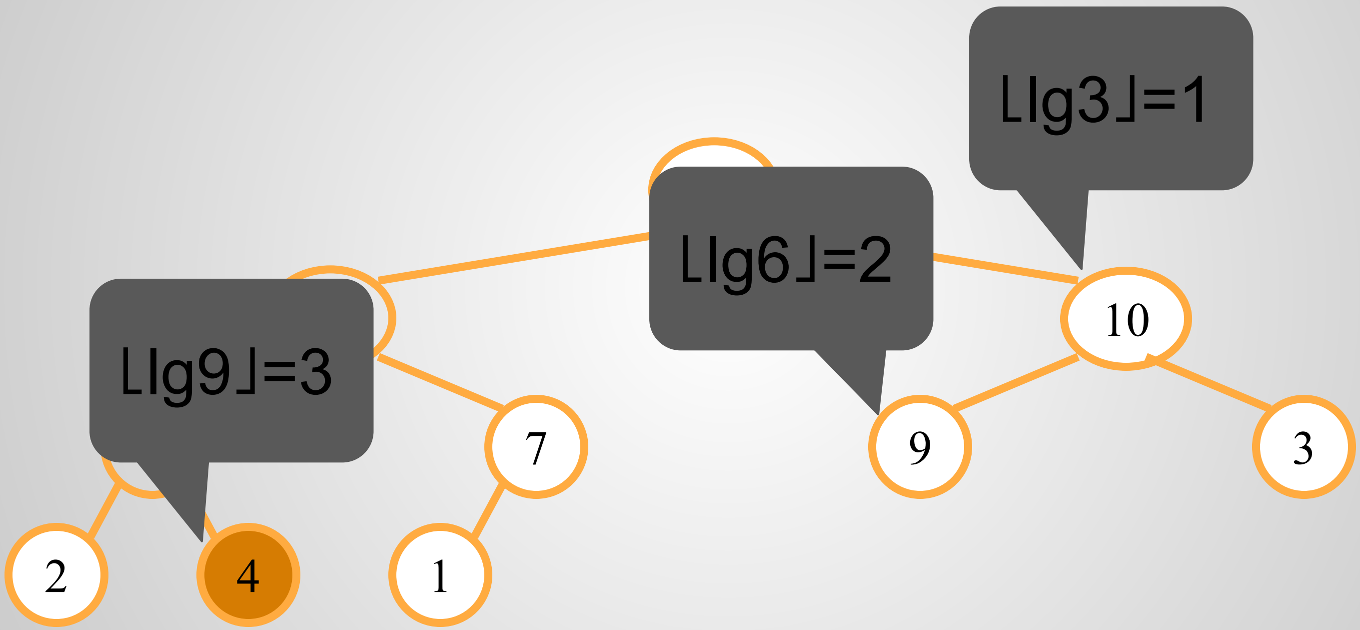
	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	8	7	9	3	2	4	1

Propriedades da Heap

- **Max-heap** : *A[Parent(i)]* $\geq$ *A[i]*
- **Min-heap** : *A[Parent(i)]* $\leq$ *A[i]*
- A altura de um nó na árvore é dada pelo número de arestas no caminho mais longo do nó atual até um nó folha.
- A altura da árvore é dada pela altura do nó raiz.
- A altura de uma heap de n elementos, baseada em uma árvore binária completa, será $\Theta(\lg n)$.

Propriedades da Heap

- Provar que o nó i pertence ao nível $\lfloor \lg i \rfloor$

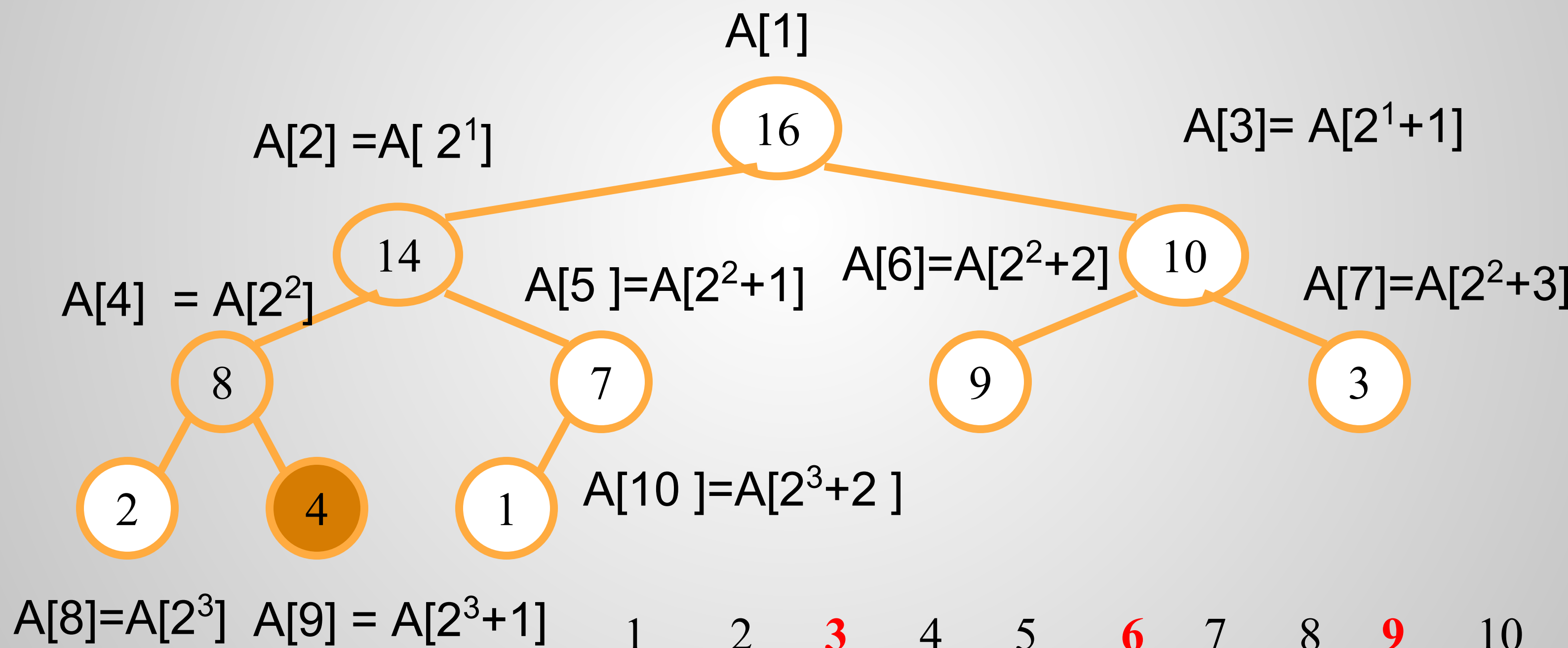


	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	8	7	9	3	2	4	1

Propriedades da Heap

- Provar que o nó i pertence ao nível $\lfloor \lg i \rfloor$

  Um nível m possui os nós $2^m, 2^m+1, 2^m+2, \ldots 2^{m+1}-1$.



	1	2	**3**	4	5	**6**	7	8	**9**	10
A =	16	14	10	8	7	9	3	2	4	1

Propriedades da Heap

- Provar que o nó i pertence ao nível $\lfloor \lg i \rfloor$

  Um nível m possui os nós $2^m, 2^m+1, 2^m+2, \ldots 2^{m+1}-1$.

  Um nó i em um nível m estará entre

  $$2^m \leq i < 2^{m+1}$$

  $$\lg 2^m \leq \lg i < \lg (2^{m+1})$$

  $$m \leq \lg i < m+1$$

  Logo, o nível do nó i será $m = \lfloor \lg i \rfloor$

- A altura de uma heap com n elementos será

  $$h = \lfloor \lg n \rfloor$$

- O total de níveis de uma heap com n elementos será

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

Rotinas para heap

- **Max-Heapify**
- **Build-Max-Heap**
- **Heapsort**
- Max-Heap-Insert
- Heap-Extract-Max
- Heap-Increase-Key
- Heap-Maximum

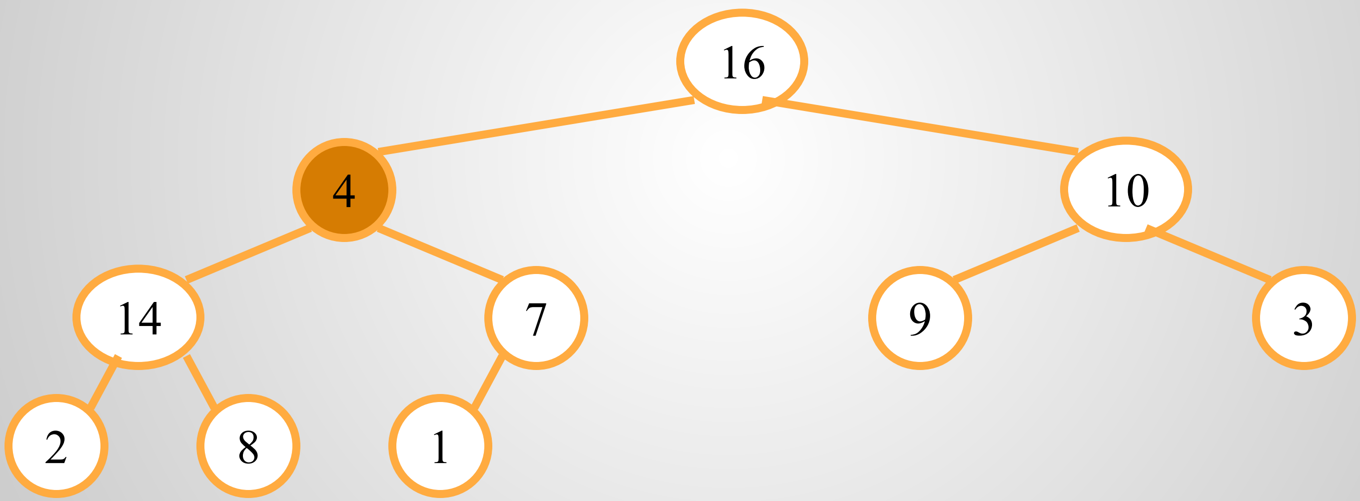
Max-Heapify(*A*, *i*)

- Essa rotina assegura a propriedade max-heap.

```
Max-Heapify(A,i)
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest  = l
else largest  = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
        exchange A[i] ↔ A[largest]
        Max-Heapify(A, largest)
```

Max-Heapify(A,i)

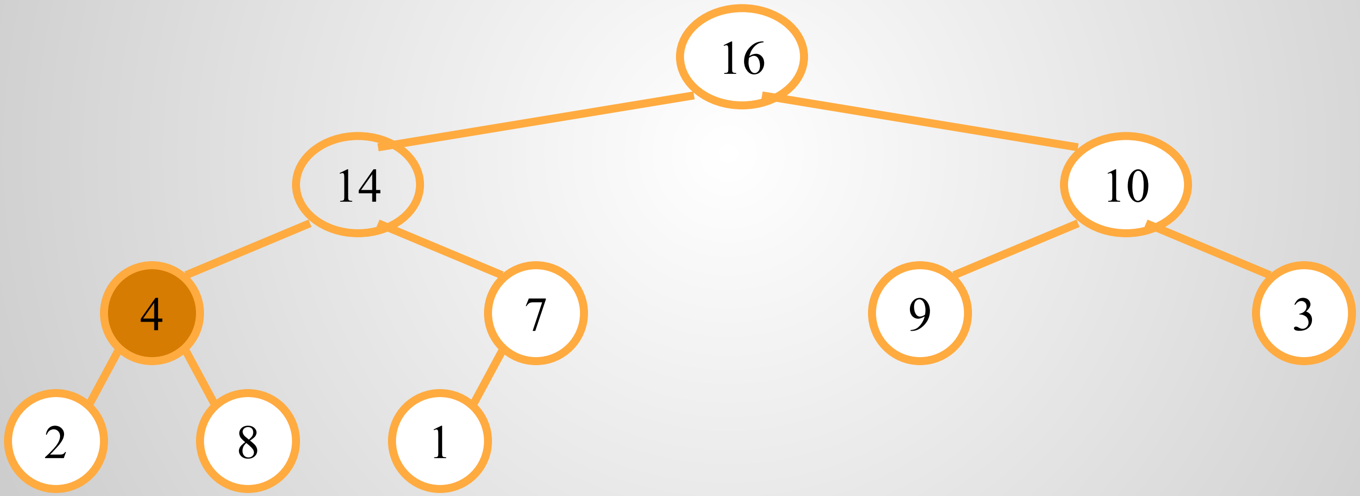
`Max-Heapify(A,2)`



	1	2	3	4	5	6	7	8	9	10
A =	16	4	10	14	7	9	3	2	8	1

Max-Heapify(A,i)

```
Max-Heapify(A,2)
    -> Max-Heapify(A,4)
```



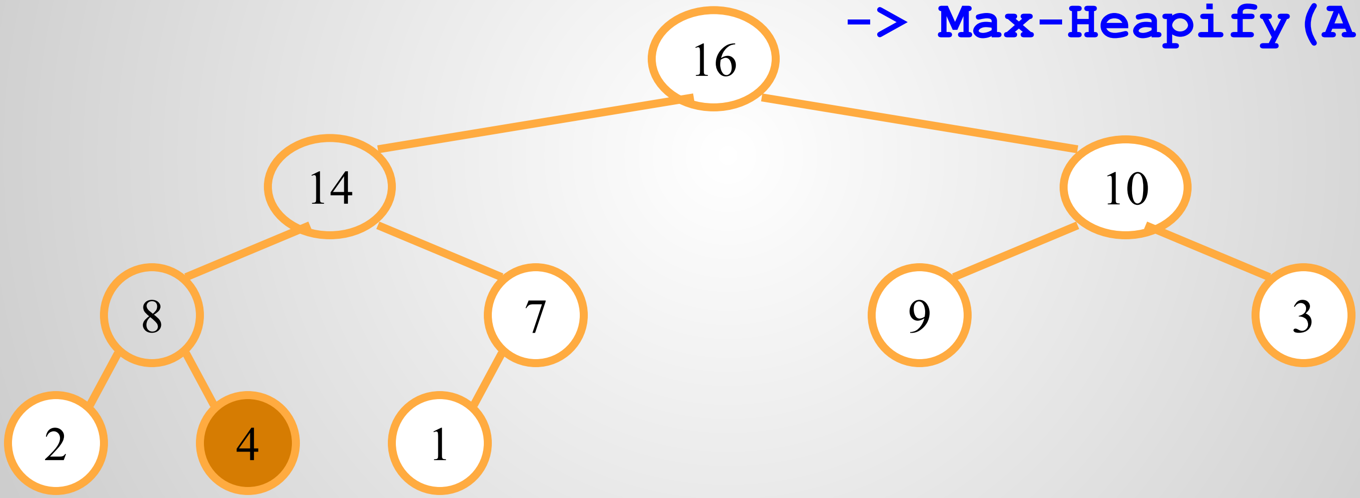
	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	4	7	9	3	2	8	1

Max-Heapify(A,i)

Max-Heapify(A,2)

-> Max-Heapify(A,4)

-> Max-Heapify(A,9)



	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	8	7	9	3	2	4	1

Analisando Max-Heapify(A, i)

- Essa rotina assegura a propriedade max-heap.

```
Max-Heapify(A,i)
l = Left(i)                                      Θ(1)
r = Right(i)                                     Θ(1)
if l ≤ A.heap-size and A[l] > A[i]               Θ(1)
    largest  = l                                 O(1)
else largest  = i                                O(1)
if r ≤ A.heap-size and A[r] > A[largest]         Θ(1)
    largest = r                                  O(1)
if largest ≠ i                                   Θ(1)
       exchange A[i] ↔ A[largest]                O(1)
       Max-Heapify(A, largest)                   T(2n/3)
```

Analisando Max-Heapify(A, i)

- De onde vem T(2n/3)??????

Prova:

  - Suponha uma árvore binária completa com altura h.
  - A heap terá um total de 2^h-1 nós.
  - No pior caso, a subárvore esquerda da heap será completa com altura h-1 e $2^{h-1}-1$ nós.
  - No pior caso, a subárvore direita da heap será completa com altura h-2 e $2^{h-2}-1$ nós.

Analisando Max-Heapify(A, i)

- De onde vem T(2n/3)??????

Prova:

  - $E = 2^{h-1}-1$ nós na subárvore esquerda.
  - $D = 2^{h-2}-1$ nós na subárvore direita.
  - $H = 1(\text{raiz}) + D + E = 1 + 2^{h-2}-1 + 2^{h-1}-1 = 3.2^{h-2}-1$.
  - E/H taxa de ocupação de E em H.

    $E/H = (2^{h-1}-1)/(3.2^{h-2}-1) = (2.2^{h-2}-1)/(3.2^{h-2}-1)$

    $E/H = \lim_{h\to\infty}(2.2^{h-2}-1)/(3.2^{h-2}-1) = 2/3$

Analisando Max-Heapify(*A*, *i*)

- Provando que Max-Heapify é O(lgn)
  - Considerando as instruções do algoritmo, temos:

    $T(n) \leq T(2n/3) + \Theta(1)$.

    Pelo teorema mestre, a=1, b=⅔ f(n)=k

    $n^{\log b(a)} = n^{\log 2/3(1)} = n^0 = 1$.

    Logo, $f(n) = \Theta(1)$ e estamos no caso 2

    $T(n) = \Theta(n^{\log b(a)} \lg n) = \Theta(1.\lg n) = \Theta(\lg n)$

Analisando Max-Heapify(A, i)

- Outra forma de analisar Max-Heapify é considerando a recursividade na altura da heap.
- Teremos $T(h)=T(h-1)+\Theta(1)$.
- Prove!!! Dica: A altura de uma heap com n elementos será $h=\lfloor \lg n \rfloor$

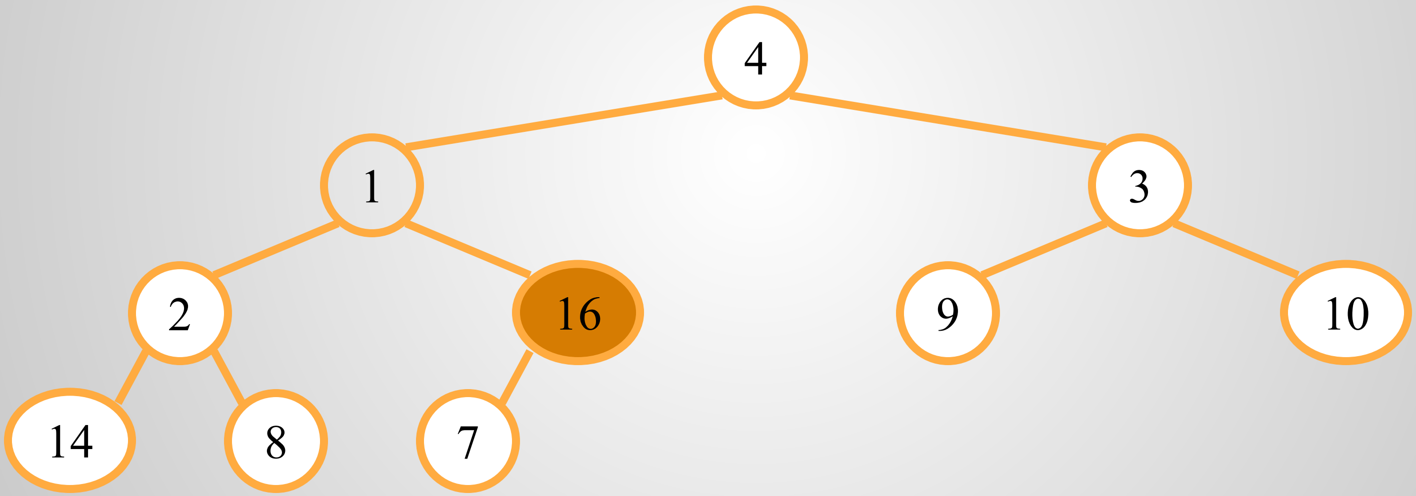
Build-Max-Heap(A)

- Retorna uma max-heap a partir de um vetor desordenado.

```
Build-Max-Heap(A)

A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
        Max-Heapify(A, i)
```

Build-Max-Heap(A)

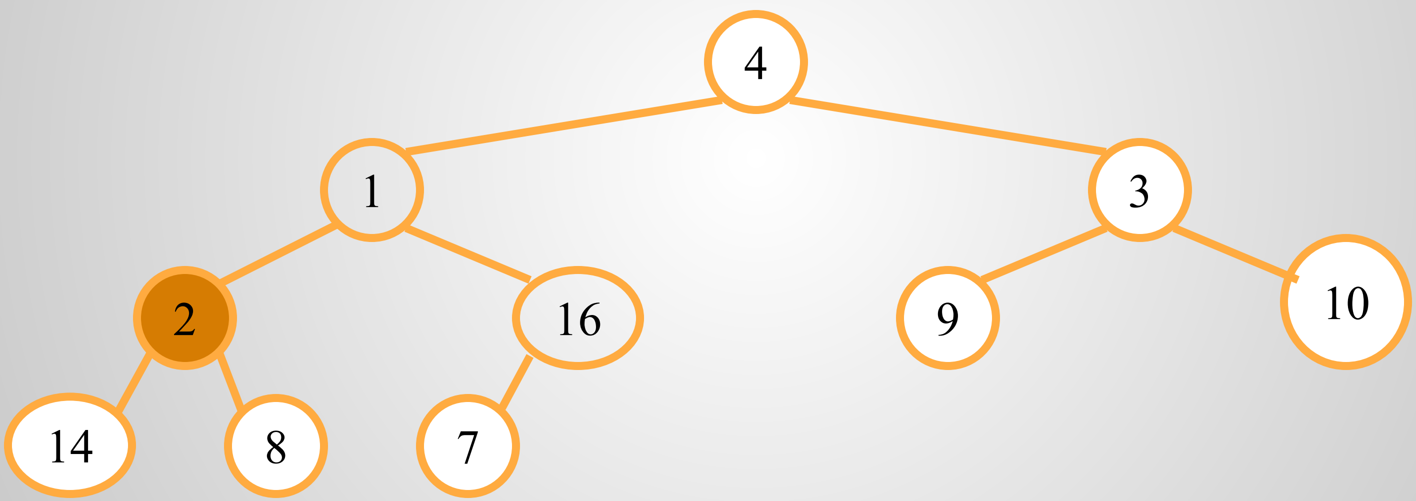
`Max-Heapify(A,5)`



	1	2	3	4	5	6	7	8	9	10
A =	4	1	3	2	16	9	10	14	8	7

Build-Max-Heap(A)

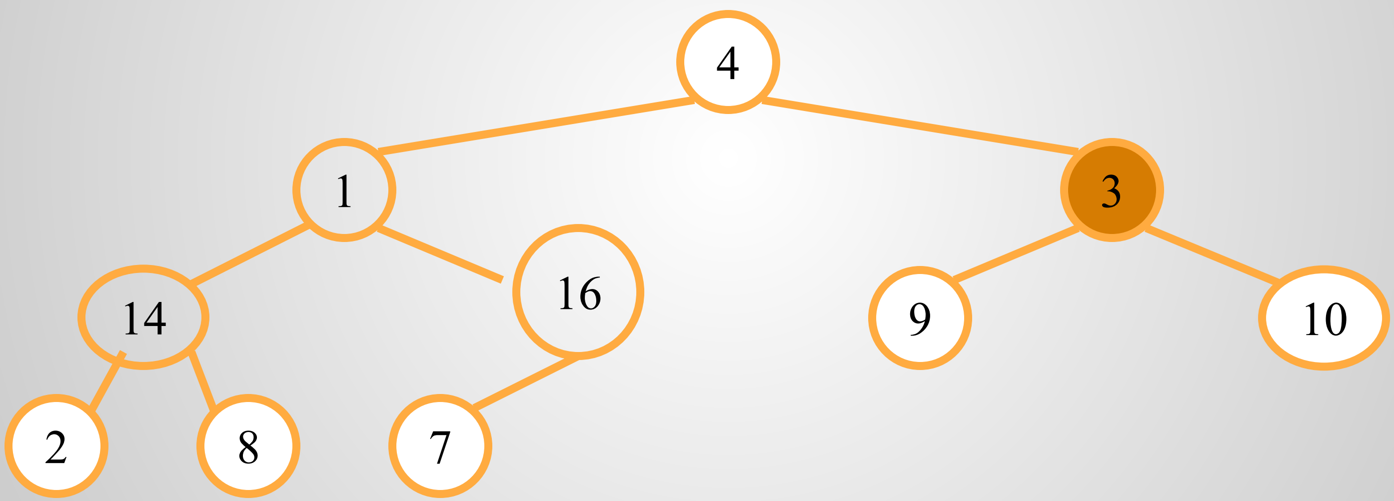
`Max-Heapify(A,4)`



	1	2	3	4	5	6	7	8	9	10
A =	4	1	3	2	16	9	10	14	8	7

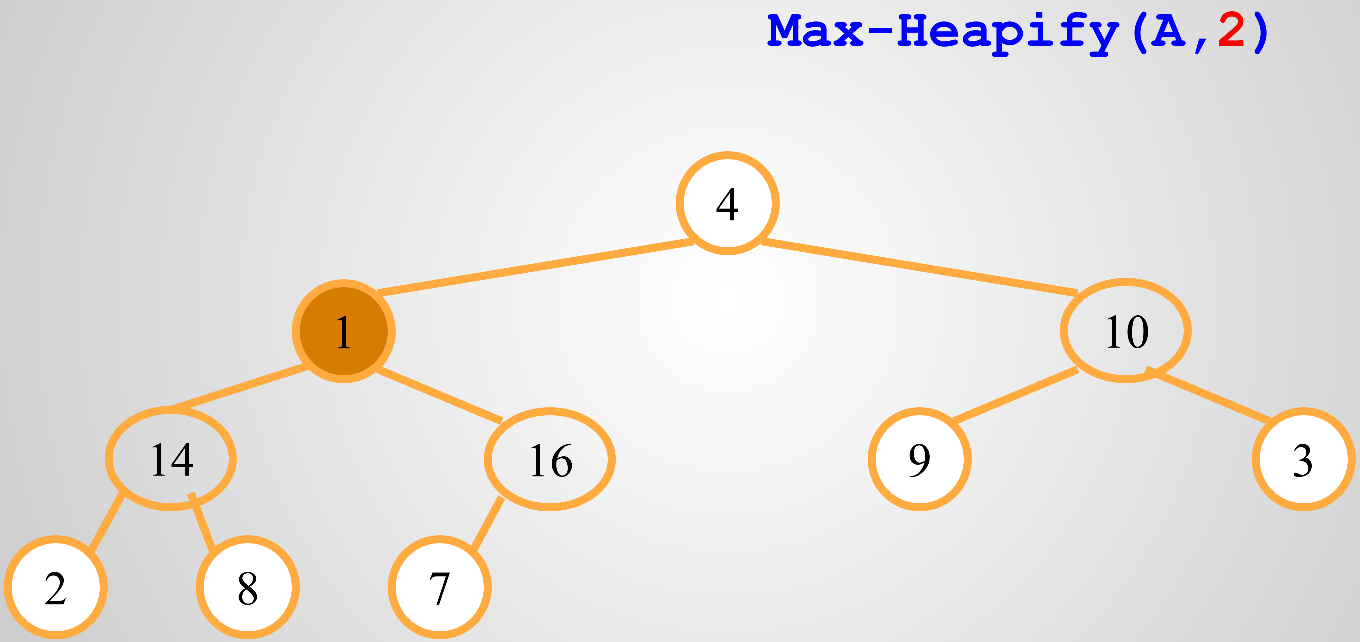
Build-Max-Heap(A)

Max-Heapify(A,3)



	1	2	3	4	5	6	7	8	9	10
A =	4	1	3	14	16	9	10	2	8	7

Build-Max-Heap(A)

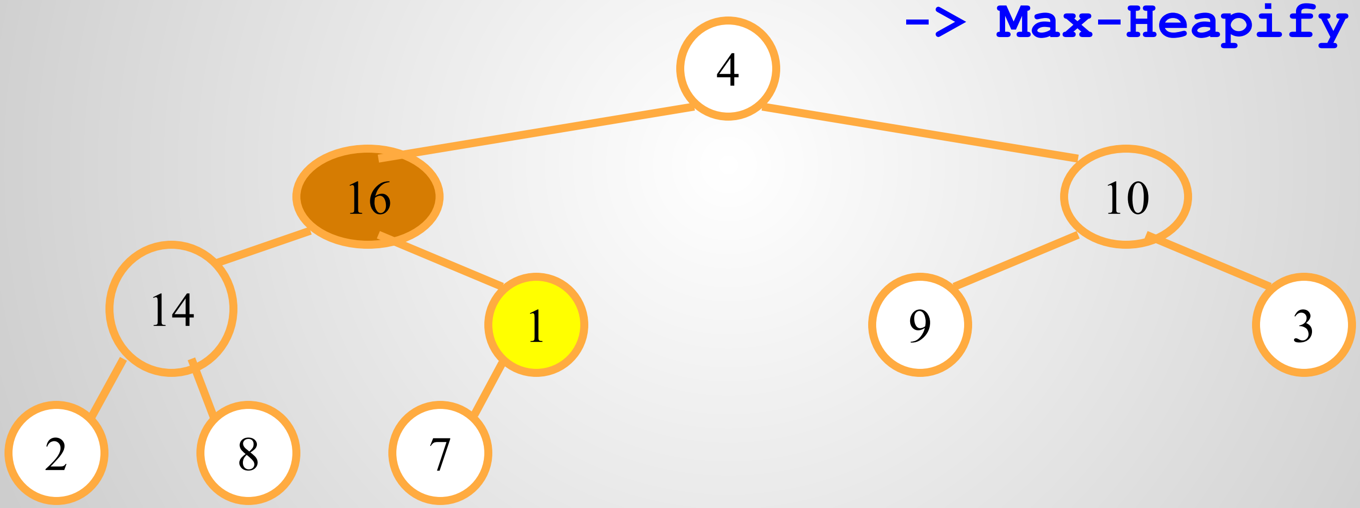


	1	2	3	4	5	6	7	8	9	10
A =	4	1	10	14	16	9	3	2	8	7

Build-Max-Heap(A)

Max-Heapify(A,2)

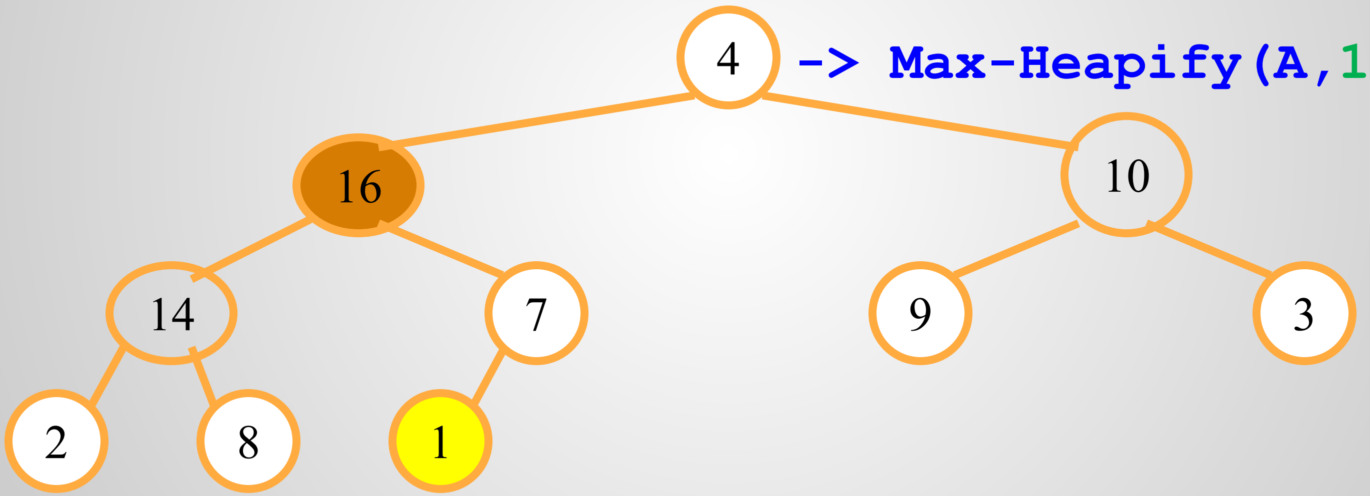
-> Max-Heapify(A,5)



	1	2	3	4	5	6	7	8	9	10
A =	4	16	10	14	1	9	3	2	8	7

Build-Max-Heap(A)

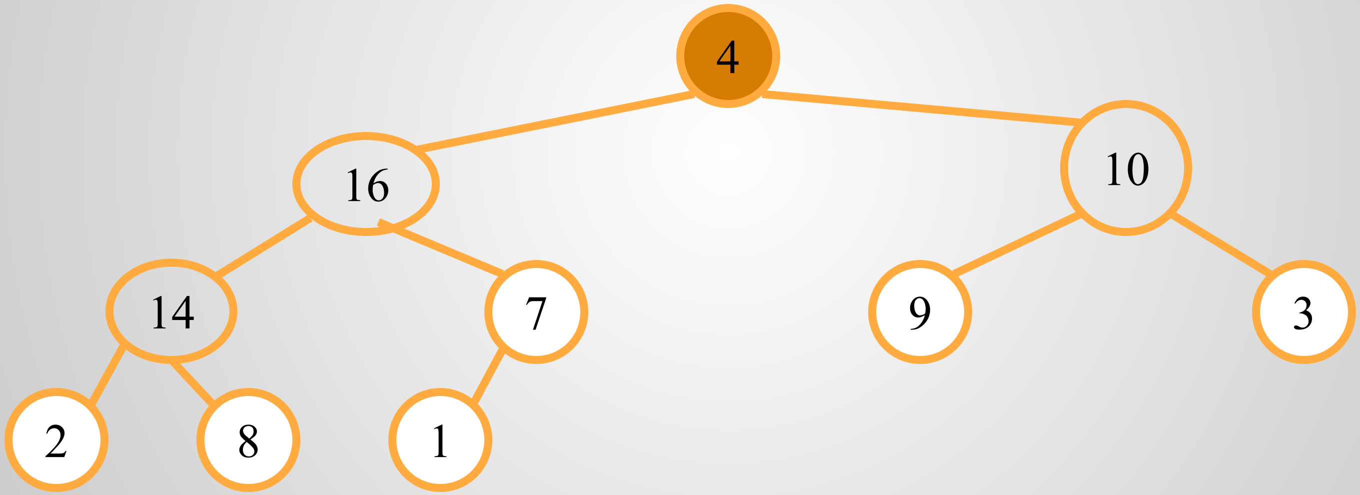
```
Max-Heapify(A,2)
    -> Max-Heapify(A,5)
    -> Max-Heapify(A,10)
```



	1	2	3	4	5	6	7	8	9	10
A =	4	16	10	14	7	9	3	2	8	1

Build-Max-Heap(A)

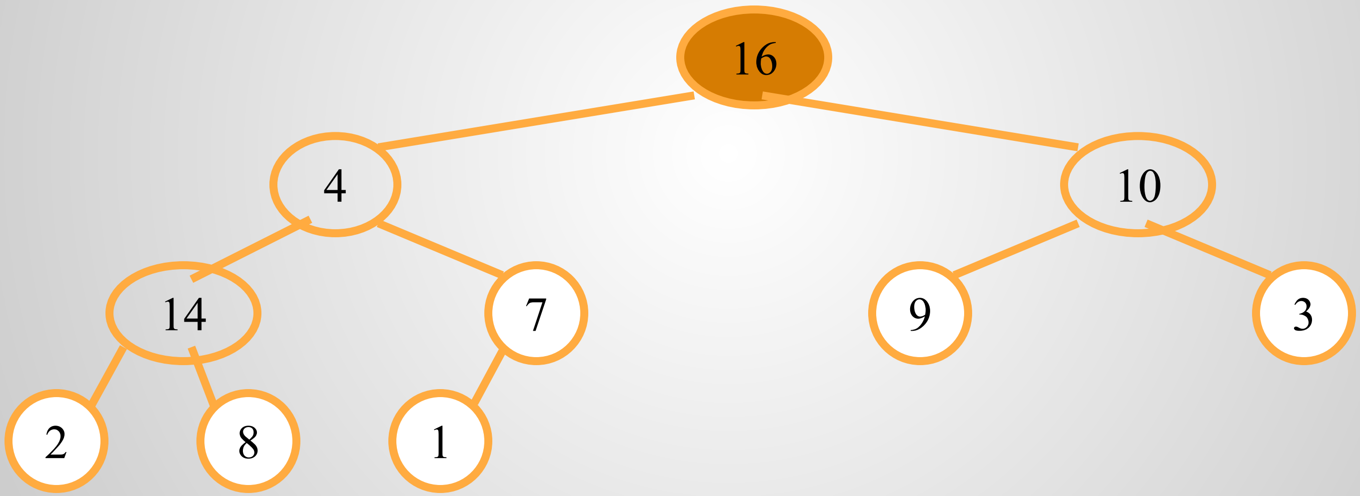
`Max-Heapify(A,1)`



	1	2	3	4	5	6	7	8	9	10
A =	4	16	10	14	7	9	3	2	8	1

Build-Max-Heap(A)

```
Max-Heapify(A,1)
   ->Max-Heapify(A,2)
```



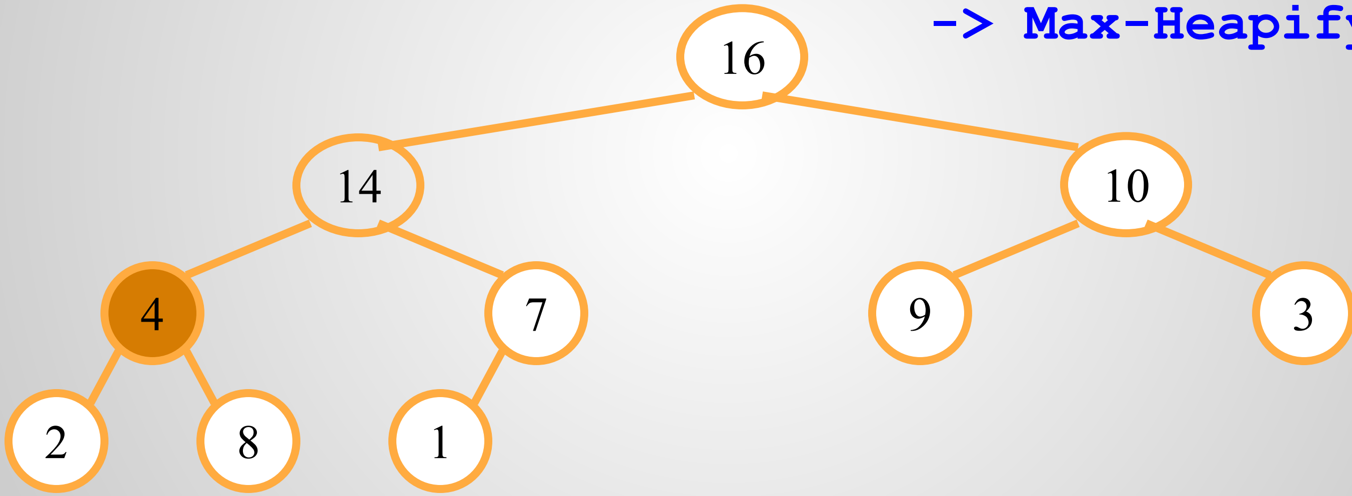
	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	8	7	9	3	2	4	1

Build-Max-Heap(A)

Max-Heapify(A,1)

-> Max-Heapify(A,2)

-> Max-Heapify(A,4)



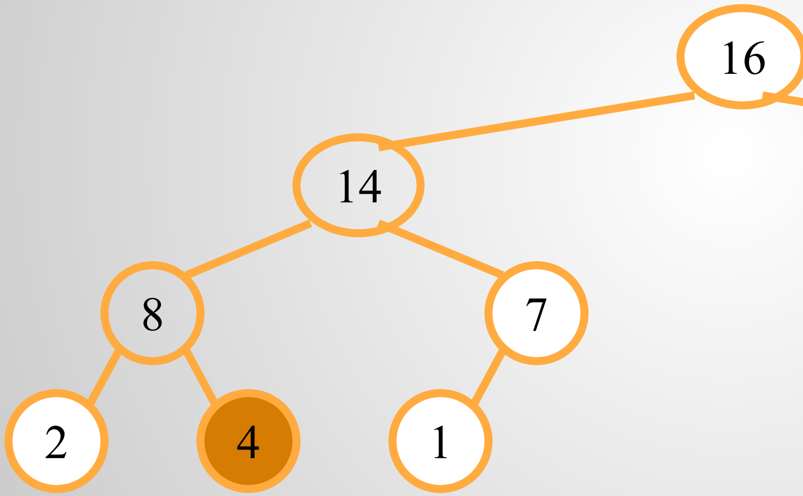
	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	4	7	9	3	2	8	1

Build-Max-Heap(A)

Max-Heapify(A,2)

-> Max-Heapify(A,4)

-> Max-Heapify(A,9)



	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	8	7	9	3	2	4	1

Analisando Build-Max-Heapify(A, i)

- Uma heap com n elementos tem altura $\lfloor \lg n \rfloor$ (Provado!!!).

- Uma heap com n elementos possui na altura h no máximo $\lceil n/2^{h+1} \rceil$ nós (Provar!!!).

- O tempo gasto por Max-Heapify quando executado para um nó é O(h).

Analisando Max-Heapify(A, i)

- No pior caso, teremos para todos os $\lceil n/2^{h+1} \rceil$ nós em todas as $\lfloor \lg n \rfloor$ alturas da heap:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) .$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

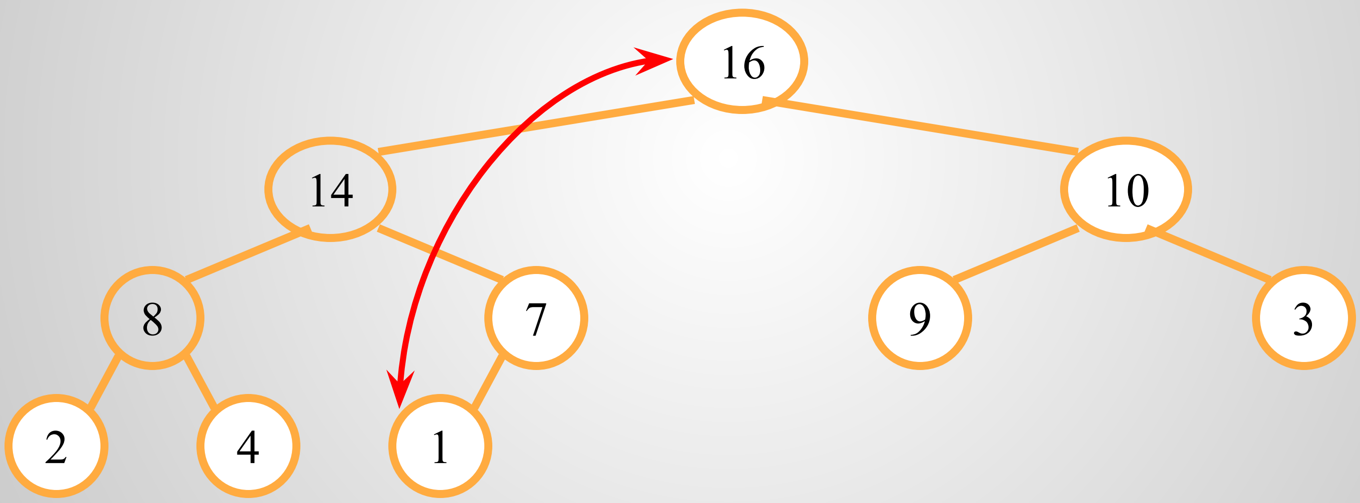
for $|x| < 1$.

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

Heapsort(A)

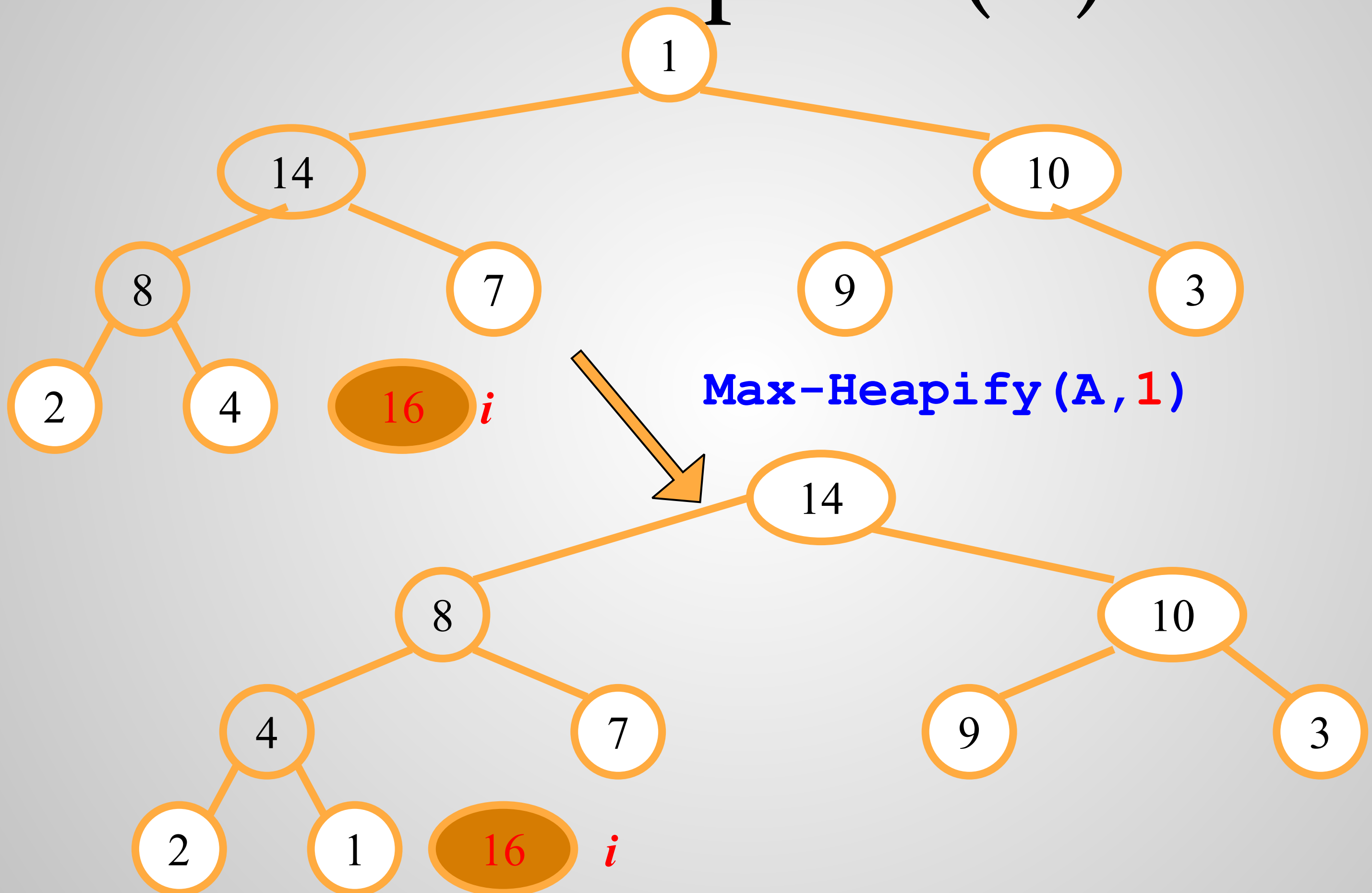
```
Heapsort(A)

Build-Max-Heap(A)
for i = A.length downto 2
    exchange A[1]↔A[i]
    A.heap-size = A.heap-size -1
    Max-Heapify(A,1)
```

Heapsort(A)

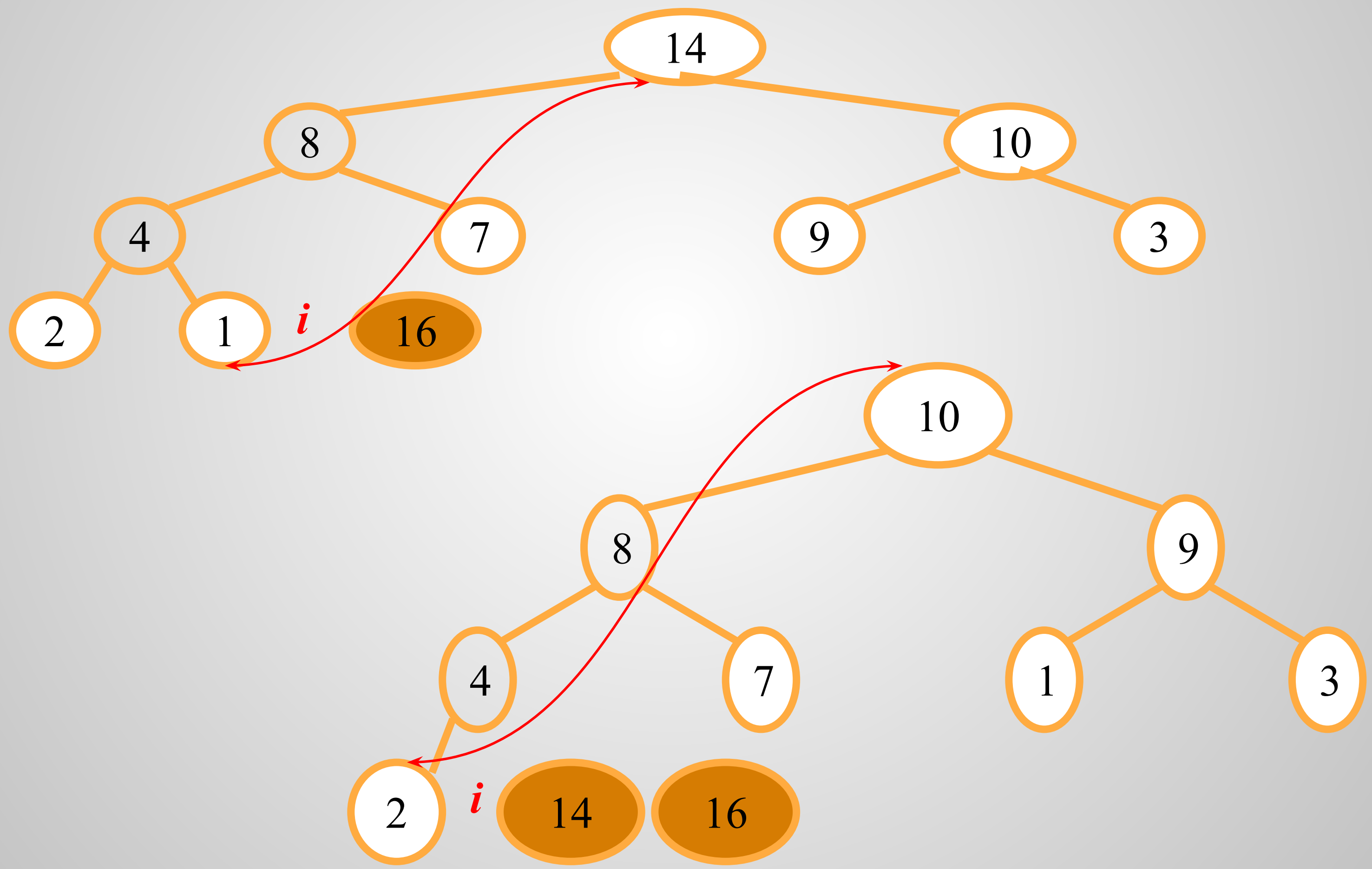


	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	8	7	9	3	2	4	1

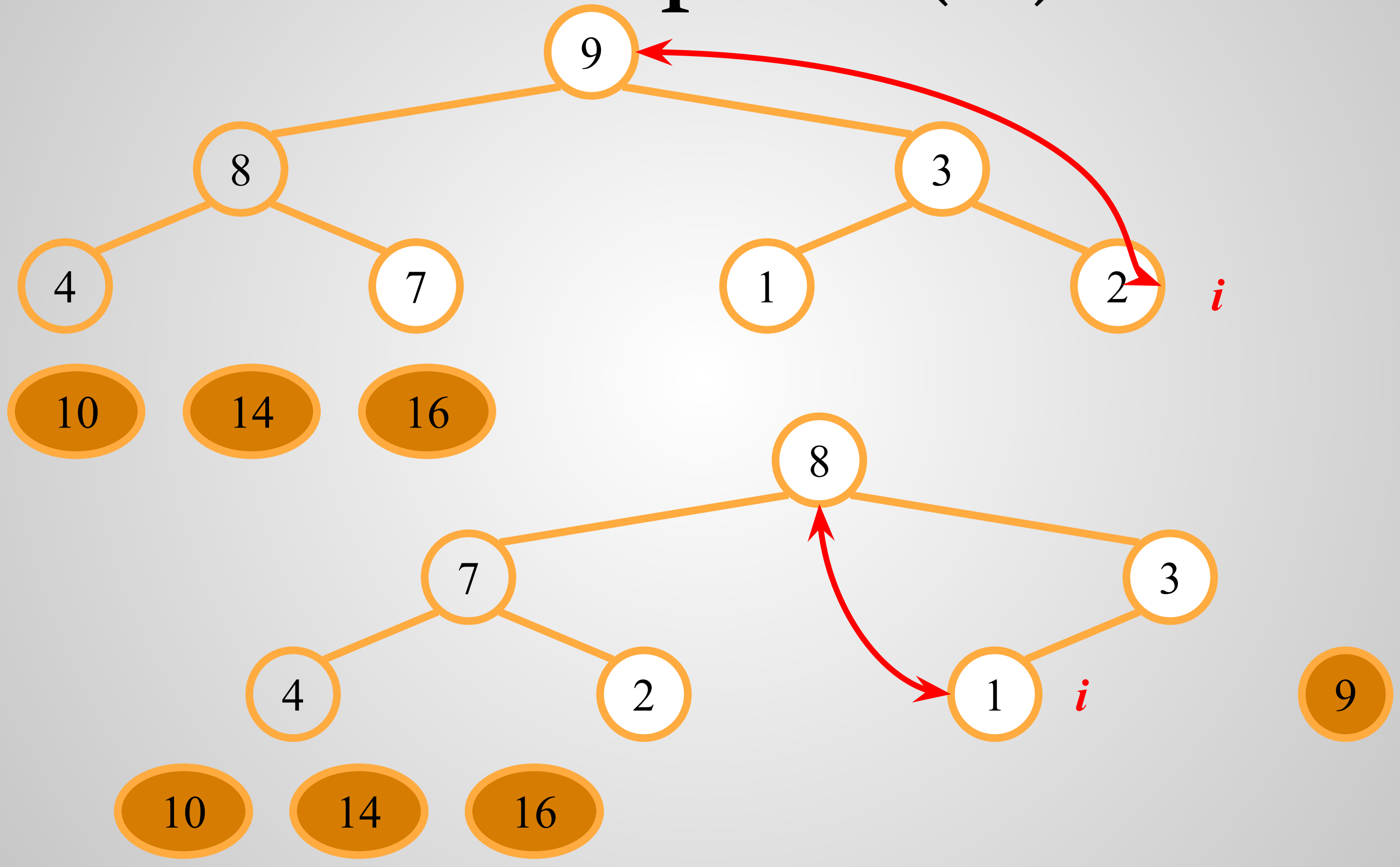
Heapsort(A)



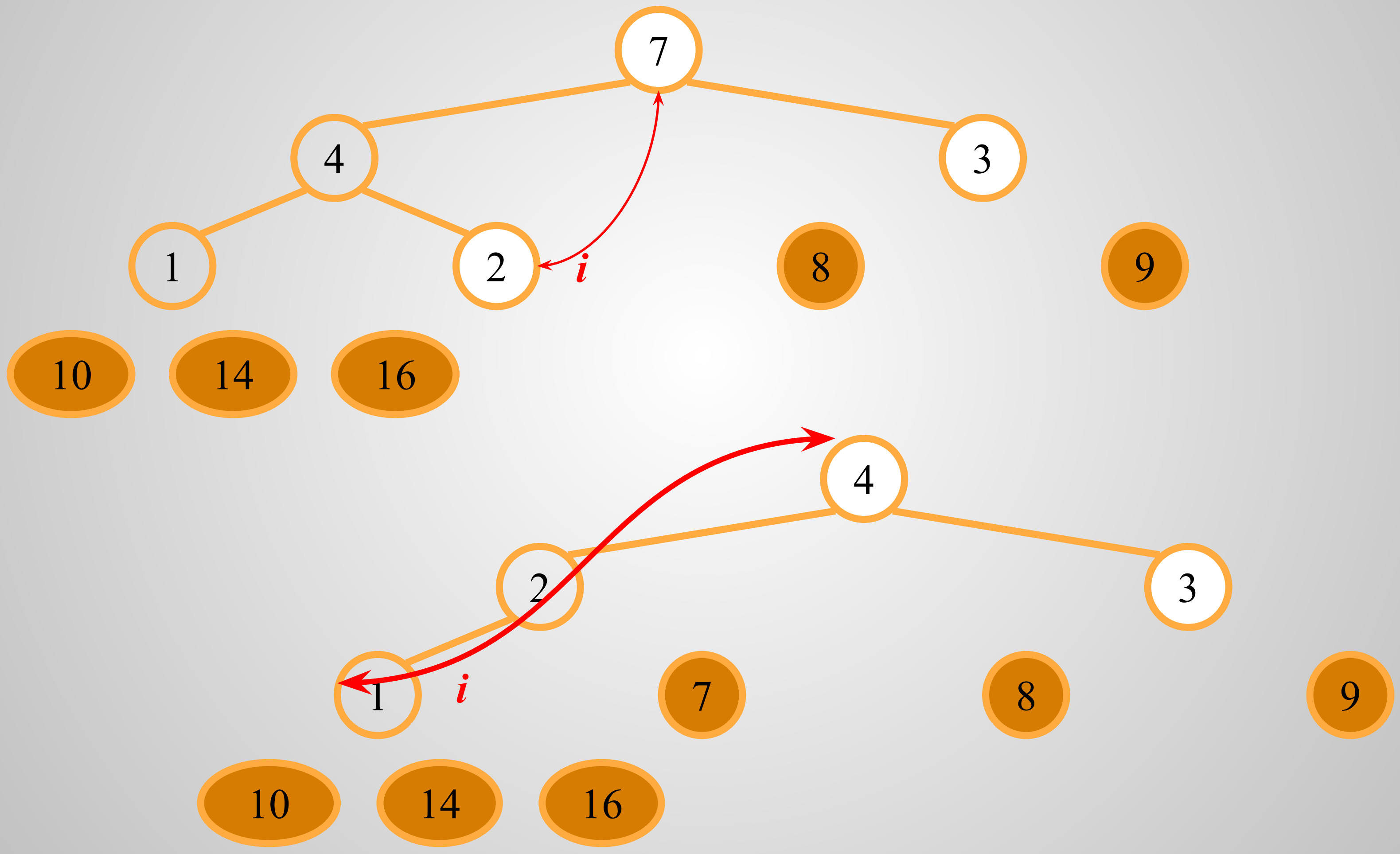
Heapsort(A)



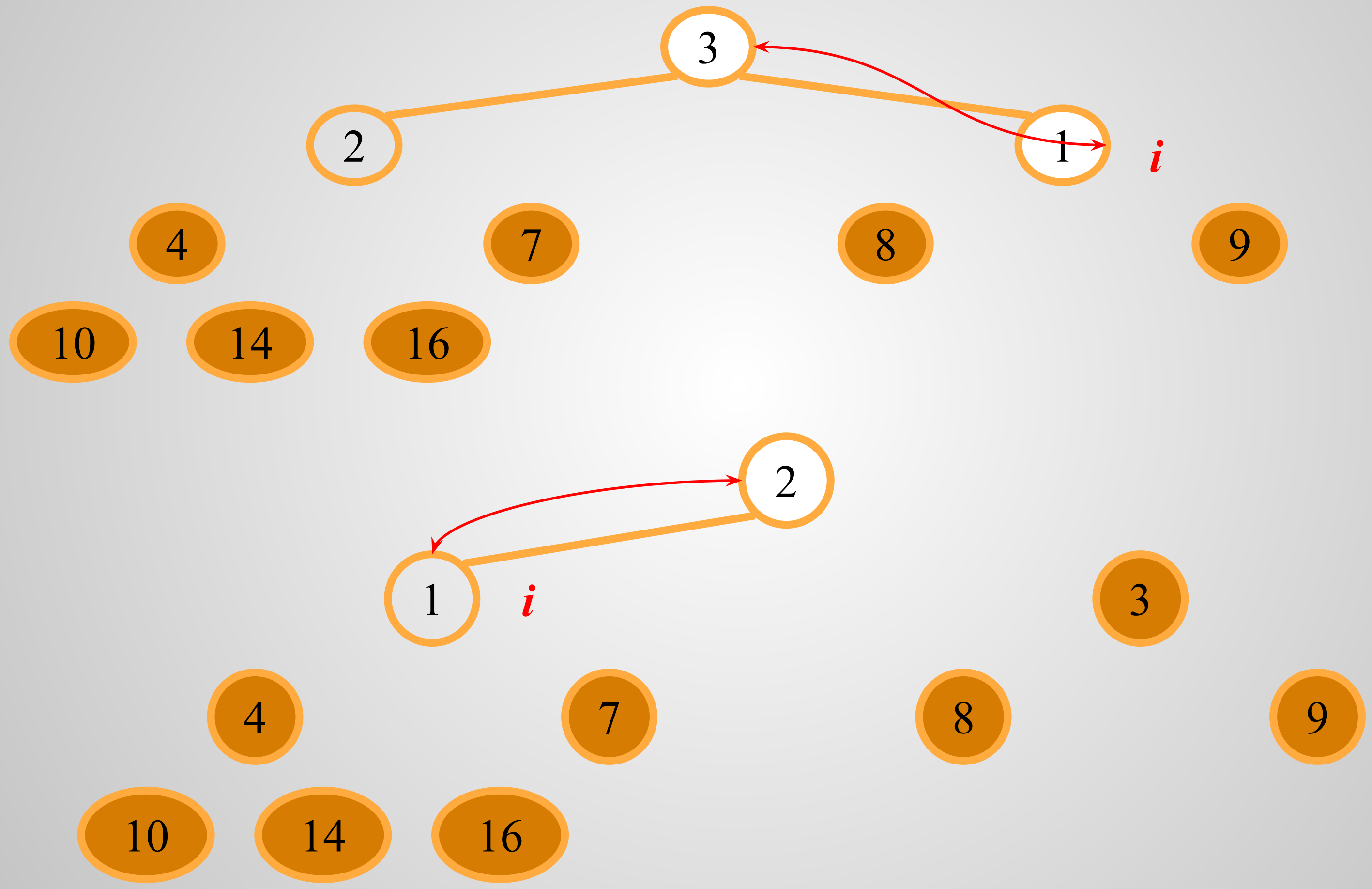
Heapsort(A)



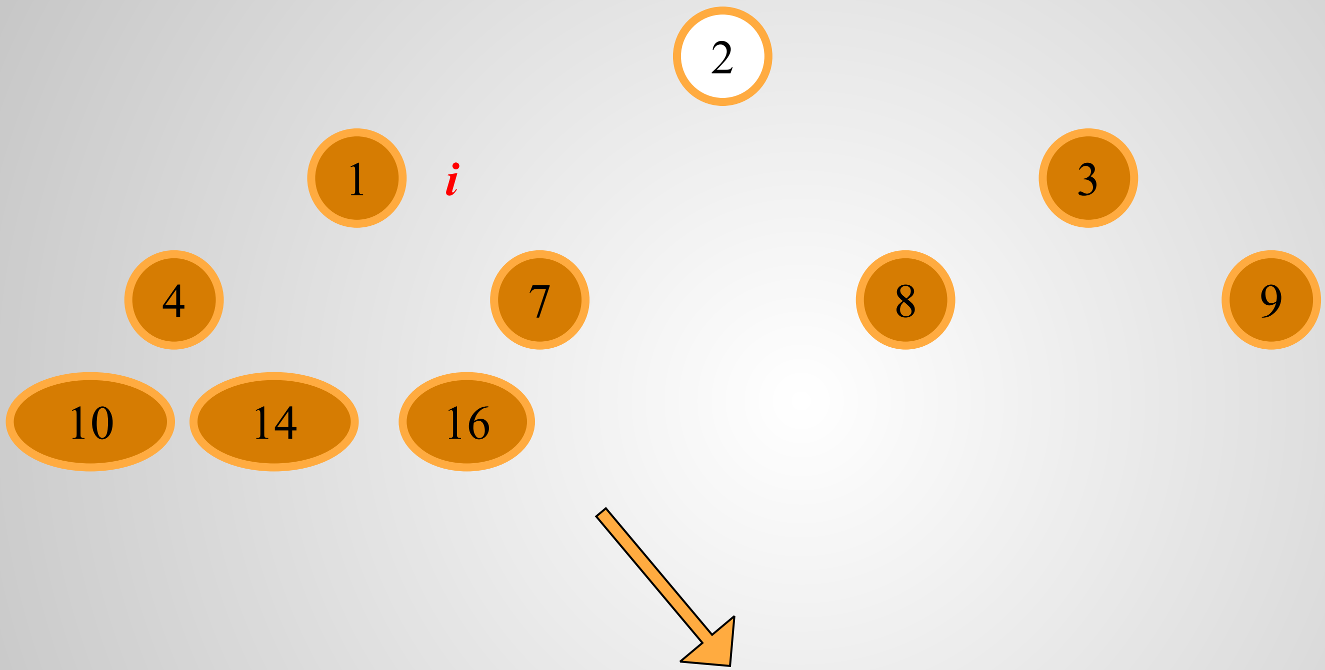
Heapsort(A)



Heapsort(A)



Heapsort(A)



	1	2	3	4	5	6	7	8	9	
A =	1	2	3	4	7	8	9	10	14	16

Analisando o Heapsort(A)

Heapsort(A)

`Build-Max-Heap(A)`	O(n)
`for i = A.length downto 2`	Θ(n)
`    exchange A[1]↔A[i]`	Θ(n)
`    A.heap-size = A.heap-size -1`	Θ(n)
`    Max-Heapify(A,1)`	nO(lgn)

Pior caso: O(nlgn)