Ordenação por seleção

→ Ideia: selecione o i-ésimo menor elemento do vetor e coloque-o na i-ésima posição

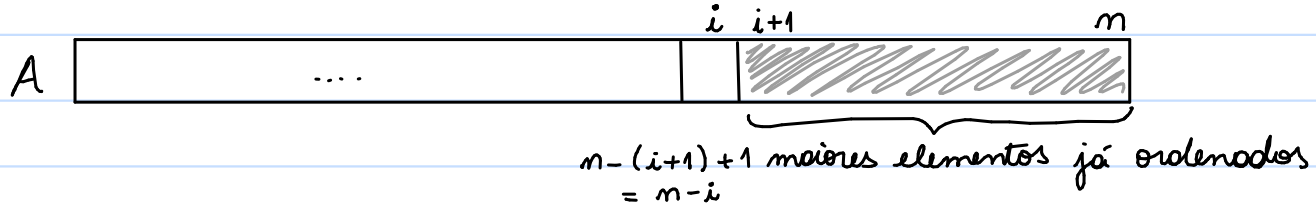
1	2	3	4	5	6	7	8
7	3	1	10	2	8	15	6

→ "Mesma" ideia: selecione o i-ésimo maior elemento e coloque-o na $(n-i+1)$-ésima posição

1	2	3	4	5	6	7	8
7	3	1	10	2	8	15	6

Selection Sort

→ Mantém o vetor dividido em dois subvetores contíguos separados por uma posição i



Selection Sort

1	2	3	4	5	6	7	8
7	3	1	10	2	8	15	6

```
SELECTIONSORT(A, n)
1 para i = n até 2, decrementando faça
2     iMax = i
3     para j = 1 até i − 1, incrementando faça
4         se A[j] > A[iMax] então
5             iMax = j
6     troca A[iMax] com A[i]
7 devolve A
```

Selection Sort

① O Selection Sort está correto?

Podemos provar que sim usando duas invariantes de laço.

Sobre o laço para interno:

$P(t)$ = "antes da t-ésima iteração começar, temos que $j = t$ e $A[iMax]$ é maior ou igual a qualquer elemento de $A[1..j-1]$ e maior ou igual a $A[i]$."

Sobre o laço para externo:

$R(t)$ = "antes da t-ésima iteração começar, temos que $i = n - t + 1$ e o subvetor $A[i+1..n]$ está ordenado de modo crescente e contém os $n-i$ maiores elementos de A."

② Qual seu tempo de execução?

$\Theta(n^2)$, pois para cada valor de i entre 2 e n, executamos o laço interno por completo (com j de 1 a $i-1$), fazendo operações constantes, e

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

Fila de prioridades

→ Coleção de elementos que possuem um campo prioridade.

→ Tipo abstrato de dados que suporta as operações:
- ↳ Remoção do elemento de maior prioridade
- ↳ Consulta ao elemento de maior prioridade
- ↳ Inserção de elemento novo
- ↳ Alteração de prioridade de um elemento
- ↳ Construção

Prioridade

u.prioridade ≥ v.prioridade

→ Consideraremos um campo numérico
- ↳ Assim, quanto maior valor, maior prioridade
- ↳ Exc: fila de banco — idade do cliente

→ E se quisermos que algo com menor valor tenha maior prioridade?
- ↳ Exc: estoque de remédios — quantidade de caixas
- ↳ multiplicamos por -1
- ↳ Exc: 10 vs. 30 ⇒ -10 ≥ -30

Heap binário

→ Estrutura de dados que implementa o tipo Fila de Prioridades

→ Cada elemento x tem campos x.prioridade e x.indice

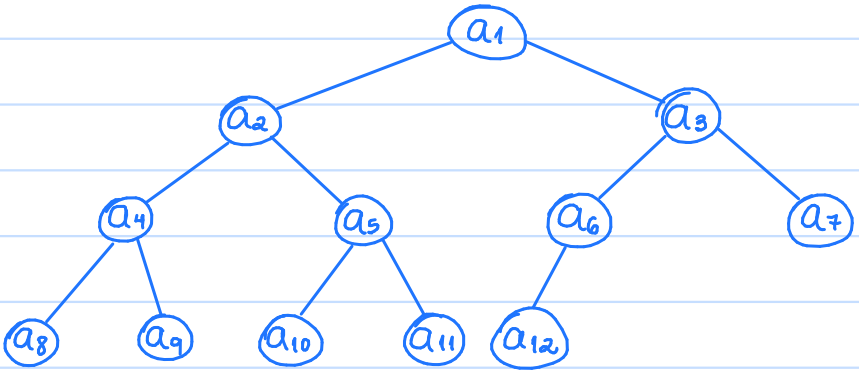
→ Implementado em um vetor
- ↳ H.capacidade, H.tamanho, H[i].indice = i

→ Visualizado como árvore

Vetor como árvore binária quase completa

$A = (a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12})$

nível 0 | nível 1 | nível 2



nível = nós até a raiz

altura = nós até uma folha

→ Ao percorrer o vetor da esq. para a dir., acessamos todos os nós do nível l consecutivamente antes de acessar os nós do nível $l+1$

→ Formalmente, o elem. na posição i (nó i) do vetor:
- ↳ tem filho esquerdo em $2i$ (se $2i \leq n$)
- ↳ tem filho direito em $2i+1$ (se $2i+1 \leq n$)
- ↳ tem pai em $\lfloor i/2 \rfloor$ (se $i > 1$)
- ↳ está no nível $\lfloor \lg i \rfloor$
- ↳ tem altura $\lfloor \lg (n/i) \rfloor$

→ Nós do nível l estão entre as posições 2^l e $\min\{n, 2^l - 1\}$

→ Podemos falar de subárvores:
- ↳ enraizada em i ou $A[i]$
- ↳ de um subvetor $A[1..k]$

Heap binário

DEFINIÇÃO: Um vetor é um heap binário se ele satisfaz a propriedade de heap.

DEFINIÇÃO: Um vetor satisfaz a propriedade de heap se todo nó tem prioridade maior ou igual à prioridade de seus filhos, se estes existirem.

→ Exemplos:

é heap: $A_1 = (15, 7, 9, 6, 5, 1, 4, 2)$

não é heap: $A_2 = (15, 7, 9, 8, 5, 10, 4, 2)$

→ Consequência da propriedade de heap:
o elemento de maior prioridade está em A[1]

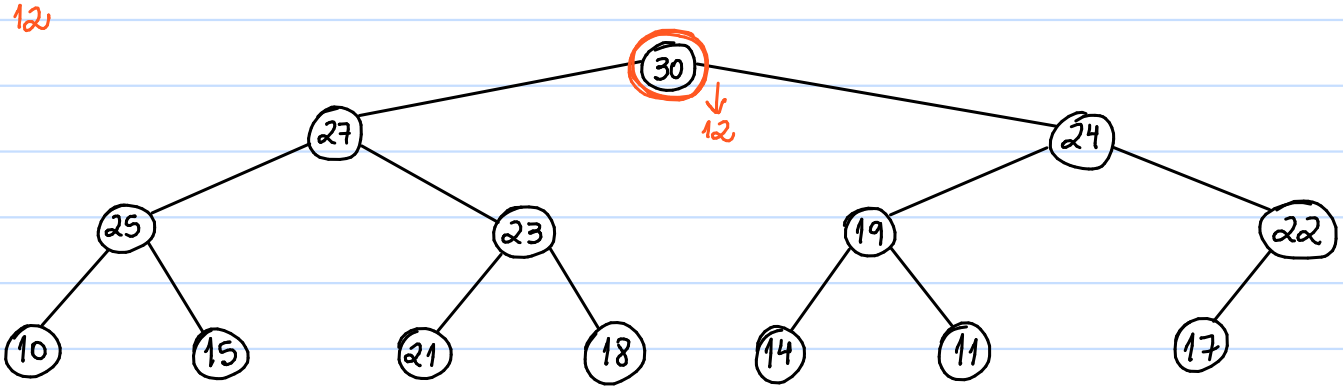
Operações

↳ Remoção do elemento de maior prioridade
↳ Consulta ao elemento de maior prioridade → constante!
↳ Inserção de elemento novo
↳ Alteração de prioridade de um elemento
↳ Construção

Restaurando a propriedade de heap se um elemento estraga

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	30	27	24	25	23	19	22	10	15	21	18	14	11	17

12



CorrigeHeapDescendo(H, i)

```
1  maior = i
2  se 2i ≤ H.tamanho e H[2i].prioridade > H[maior].prioridade então
3      maior = 2i
4  se 2i + 1 ≤ H.tamanho e H[2i + 1].prioridade > H[maior].prioridade
     então
5      maior = 2i + 1
6  se maior ≠ i então
7      troca H[i].indice com H[maior].indice
8      troca H[i] com H[maior]
9      CorrigeHeapDescendo(H, maior)
```

① Esse algoritmo está correto?

TEOREMA: O algoritmo CorrigeHeapDescendo recebe um vetor H e um índice i tal que as subárvores enraizadas em $H[2i]$ e $H[2i+1]$ são heaps, e modifica H de modo que a árvore enraizada em $H[i]$ é um heap ao término.

IDEIA DA PROVA: Por indução na altura h_i do nó i.

② Qual o tempo de execução?

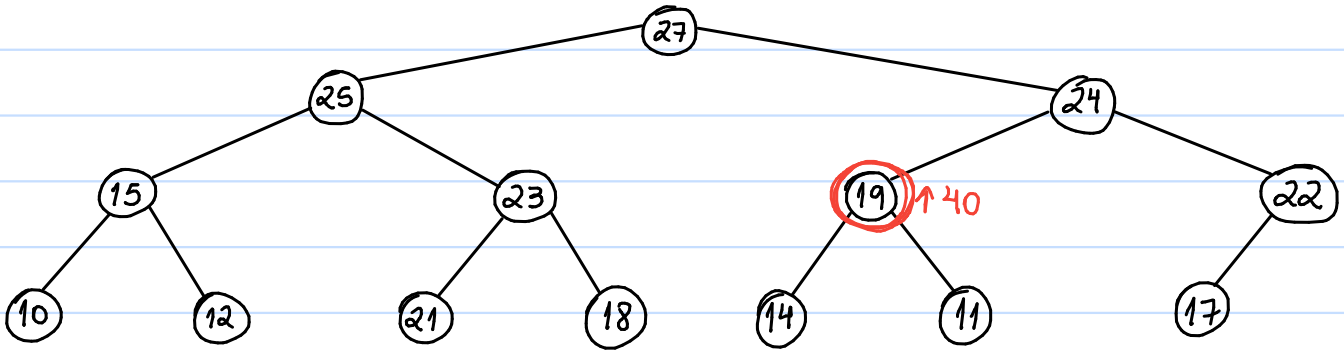
$O(\lg n)$, pois percorre no máximo um único ramo da árvore de sua posição até uma folha fazendo operações constantes, ou seja, proporcional à sua altura.

Mais precisamente, $O(\log \frac{n}{i})$ (em alguns casos pode ser mais conveniente escrever assim).

Restaurando a propriedade de heap se um elemento estraga

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	27	25	24	15	23	19	22	10	12	21	18	14	11	17

↑ 40




```
CorrigeHeapSubindo(H, i)
1 pai = ⌊i/2⌋
2 se i ≥ 2 e H[i].prioridade > H[pai].prioridade então
3     troca H[i].indice com H[pai].indice
4     troca H[i] com H[pai]
5     CorrigeHeapSubindo(H, pai)
```

① Esse algoritmo está correto?

TEOREMA: O algoritmo CorrigeHeapSubindo recebe um vetor H e um índice i tal que o subvetor H[1..i-1] é heap, e modifica H de modo que H[1..i] é heap ao término.

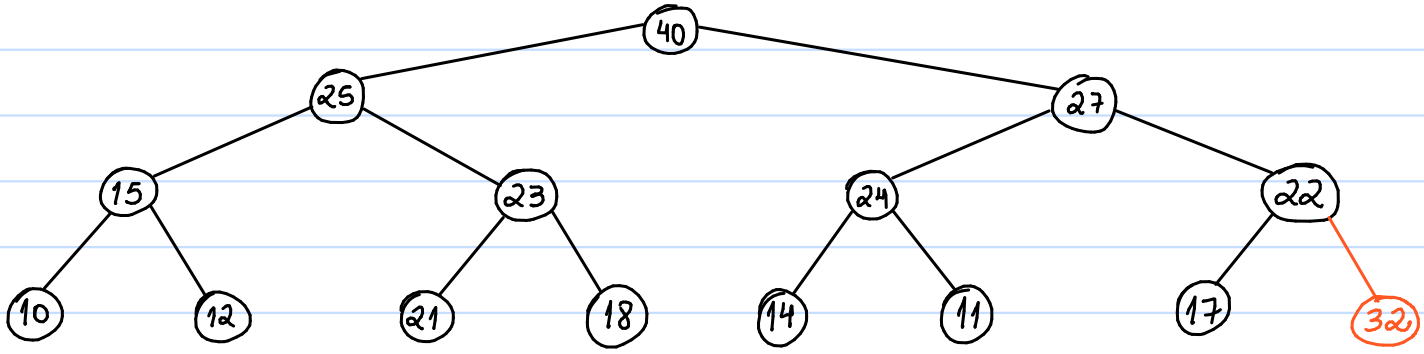
IDEIA DA PROVA: Por indução no nível i do nó i.

② Qual o tempo de execução?

O(lg n), pois percorre no máximo um único ramo da árvore de sua posição até a raiz fazendo operações constantes, ou seja, proporcional ao seu nível.
Mais precisamente, O(log i) (em alguns casos pode ser mais conveniente escrever assim).

Inserção de um novo elemento em heap

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	40	25	27	15	23	24	22	10	12	21	18	14	11	17	32




```
INSERENAHEAP(H, x)
1 se H.tamanho < H.capacidade então
2     H.tamanho = H.tamanho + 1
3     x.indice = H.tamanho
4     H[H.tamanho] = x
5     CORRIGEHEAPSUBINDO(H, H.tamanho)
```

① Esse algoritmo está correto?

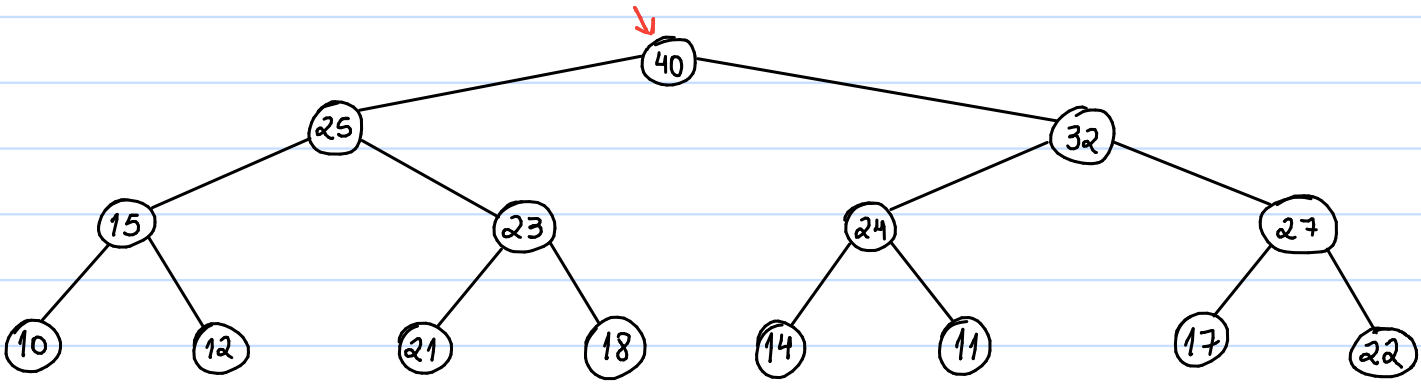
Sim, pois H inicialmente é heap, só alteramos um elemento e CORRIGEHEAPSUBINDO está correto.

② Qual o tempo de execução?

$O(\lg n)$, pois é o tempo de CORRIGEHEAPSUBINDO (n = H.tamanho)

Remoção de um elemento da heap (o de maior prioridade)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	40	25	32	15	23	24	27	10	12	21	18	14	11	17	22




```
REMOVEDAHEAP(H)
1 x = NULO
2 se H.tamanho ≥ 1 então
3     x = H[1]
4     H[H.tamanho].indice = 1
5     H[1] = H[H.tamanho]
6     H.tamanho = H.tamanho − 1
7     CORRIGEHEAPDESCENDO(H, 1)
8 devolve x
```

① Esse algoritmo está correto?

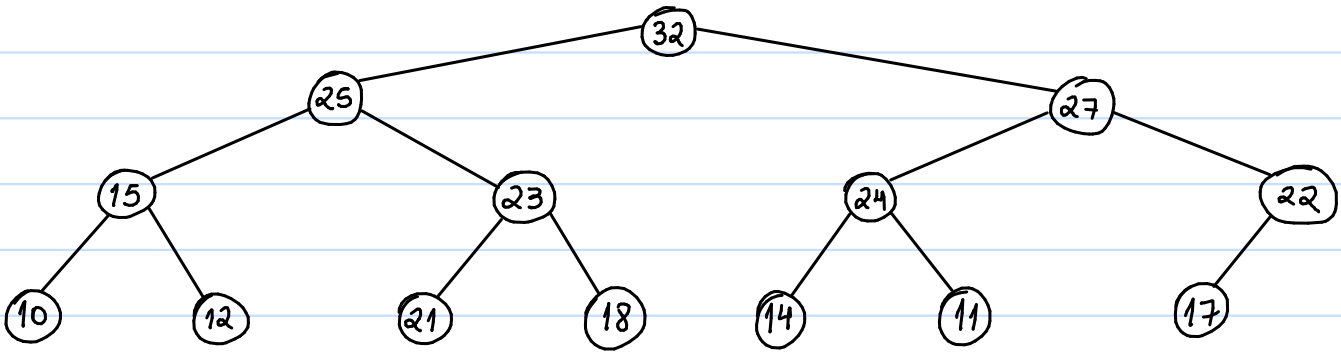
Sim, pois H inicialmente é heap, só alteramos um elemento e CORRIGEHEAPDESCENDO está correto.

② Qual o tempo de execução?

$O(\lg n)$, pois é o tempo do CORRIGEHEAPDESCENDO sobre um elemento na posição 1.

Alteração de um elemento da heap

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	32	25	27	15	23	24	22	10	12	21	18	14	11	17	




```
AlteraHeap(H, i, k)
1  aux = H[i].prioridade
2  H[i].prioridade = k
3  se aux < k então
4      CorrigeHeapSubindo(H, i)
5  se aux > k então
6      CorrigeHeapDescendo(H, i)
```

① Esse algoritmo está correto?

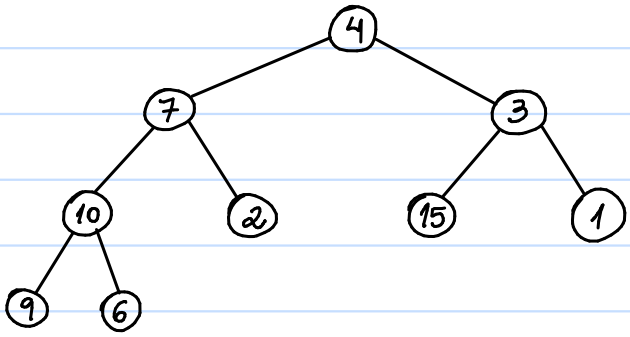
Sim, pois H inicialmente é heap, só alteramos um elemento e CORRIGEHEAPDESCENDO e CORRIGEHEAPSUBINDO estão corretos.

② Qual o tempo de execução?

$O(\lg n)$, que é o tempo de CORRIGEHEAPSUBINDO ou CORRIGEHEAPDESCENDO.

Construção de um heap a partir de elementos existentes

	1	2	3	4	5	6	7	8	9
A	4	7	3	10	2	15	1	9	6




```
CONSTROIHEAP(H, n)
1 H.tamanho = n
2 para i = 1 até H.tamanho, incrementando faça
3     H[i].indice = i
4 para i = ⌊H.tamanho/2⌋ até 1, decrementando faça
5     CORRIGEHEAPDESCENDO(H, i)
```

① Esse algoritmo está correto?

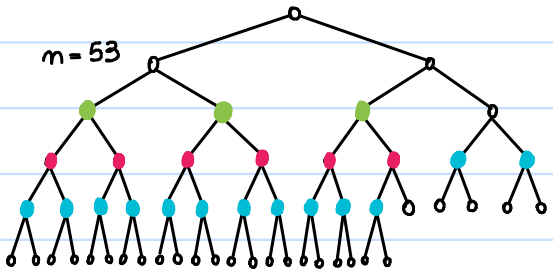
TEOREMA: O algoritmo ConstroiHeap(H, n) transforma qualquer vetor H em um heap.

IDEIA DA PROVA: Usamos a invariante

P(t) = "Antes da t-ésima iteração começar, vale que $i = \lfloor \frac{n}{2} \rfloor - t + 1$ e a árvore enraizada em H[j] é um heap para todo j tal que $i+1 \leq j \leq n = H.\text{tamanho}$."

② Qual o tempo de execução?

Resposta rápida: $O(n \log n)$



Resposta elaborada:

no máximo $\lceil n/2^{h+1} \rceil$ elementos têm altura h. (+ trocas → − elementos)

O tempo de CORRIGEHEAPDESCENDO é $O(h)$.

Então o tempo total é no máximo

$$\sum_{h=1}^{\log n} \frac{n}{2^{h+1}} O(h) \leq \sum_{h=1}^{\log n} n \cdot \frac{h}{2^{h+1}} = O(n)$$

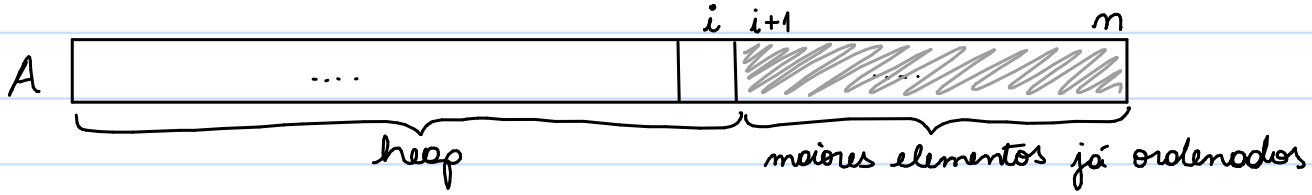
altura 3 = 3 $\leq \lceil \frac{53}{2^4} \rceil = 4$

altura 2 = 7 $\leq \lceil \frac{53}{2^3} \rceil = 7$

altura 1 = 13 $\leq \lceil \frac{53}{2^2} \rceil = 14$

Heapsort

→ Mantém o vetor dividido em dois subvetores contíguos separados por uma posição i



1	2	3	4	5	6	7	8
7	3	1	10	2	8	15	6

```
HEAPSORT(A, n)
1 CONSTROIHEAP(A)
2 para i = n até 2, decrementando faça
3     troca A[1] com A[i]
4     A.tamanho = A.tamanho − 1
5     CORRIGEHEAPDESCENDO(A, 1)
```

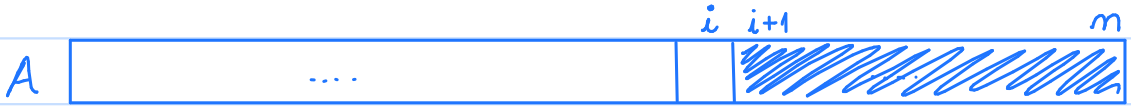
Heapsort

① O Heapsort está correto?

Podemos provar que sim usando uma invariante de laço.

Sobre o laço para externo:

$R(t)$ = "Antes da t-ésima iteração começar, temos que $i = n - t + 1$ o subvetor $A[i+1..n]$ está ordenado de modo crescente e contém os $n-i$ maiores elementos de A, $A.tamanho = i$ e $A[1..tamanho]$ é heap."



② Qual seu tempo de execução?

$O(n \lg n)$, pois são n execuções de CORRIGEHEAPDESCENDO, cujo tempo de uma execução é $O(\lg n)$.

Ordenação

	Pior caso	Melhor caso
Insertion Sort	$\Theta(n^2)$	$\Theta(n)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$
Heapsort	$\Theta(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$

Limite inferior de ordenação

Qualquer algoritmo baseado em comparações requer $\Omega(n \lg n)$ comparações no pior caso.