

Estrutura de Dados

Aula 6

GUSTAVO FORTUNATO PUGA



Retomando a Lista

- Forma simples de interligar os elementos de um conjunto.
- Agrupa informações referentes a um conjunto de elementos que se relacionam entre si de alguma forma.
- São úteis em aplicações tais como manipulação simbólica, gerência de memória, simulação e compiladores.
- Inúmeros tipos de dados podem ser representados por listas. Alguns exemplos de sistemas de informação são: informações sobre os funcionários de uma empresa, notas de alunos, itens de estoque, etc.



Estruturas

As estruturas de dados podem ser:

- lineares (ex. arrays) ou não lineares (ex. grafos);
- homogêneas (todos os dados que compõe a estrutura são do mesmo tipo) ou heterogêneas (podem conter dados de vários tipos);
- estáticas (têm tamanho/capacidade de memória fixa) ou dinâmicas (podem expandir).



Estruturas

ESTRUTURA DE DADOS LINEARES

Em uma estrutura de dados linear, os elementos de dados são organizados em uma ordem linear, onde cada um dos elementos é anexado ao seu adjacente anterior e ao próximo.

Na estrutura de dados linear, um único nível está envolvido.

Sua implementação é fácil em comparação com a estrutura de dados não linear.

Na estrutura de dados linear, os elementos de dados podem ser percorridos em uma única execução apenas.

Em uma estrutura de dados linear, a memória não é utilizada de forma eficiente.

Seus exemplos são: array, pilha, fila, lista vinculada, etc.

As aplicações de estruturas de dados lineares são principalmente no desenvolvimento de software de aplicação.

ESTRUTURA DE DADOS NÃO LINEAR

Em uma estrutura de dados não linear, os elementos de dados são anexados de maneira hierárquica.

Enquanto na estrutura de dados não linear, vários níveis estão envolvidos.

Embora sua implementação seja complexa em comparação com a estrutura de dados linear.

Enquanto na estrutura de dados não linear, os elementos de dados não podem ser percorridos em uma única execução apenas.

Embora em uma estrutura de dados não linear, a memória é utilizada de forma eficiente.

Enquanto seus exemplos são: árvores e gráficos.

As aplicações de estruturas de dados não lineares são em Inteligência Artificial e processamento de imagens.



Análise de Desempenho



Como analisar o desempenho de um algoritmo?

Duas métricas:

- Tempo
- Espaço (memória)

Três formas:

- Pior caso
- Melhor caso
- Caso médio



-

Big O $O(n)$

É o pior caso de um algoritmo

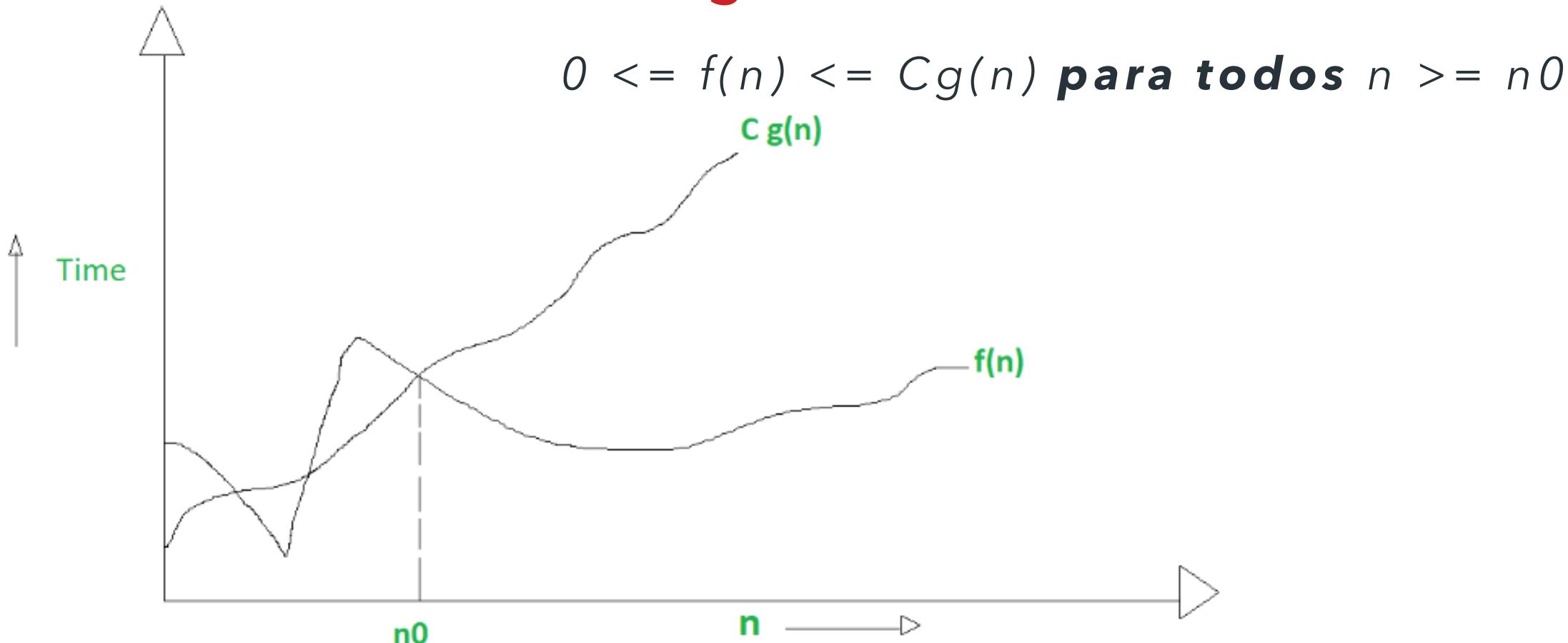
- Análise de quantos todos os passos (pior caso) que serão usados para completar a tarefa, por isso costumamos dizer que calcula o tempo máximo que o algoritmo executará, ainda que não represente o tempo exato, mas uma noção de grandeza/proporção de tempo.



Big O $O(n)$

Matematicamente, se $f(n)$ descreve o tempo de execução de um algoritmo; $f(n) \in O(g(n))$ se existir constante positiva C e n_0 tal que,

Pior caso de um algoritmo



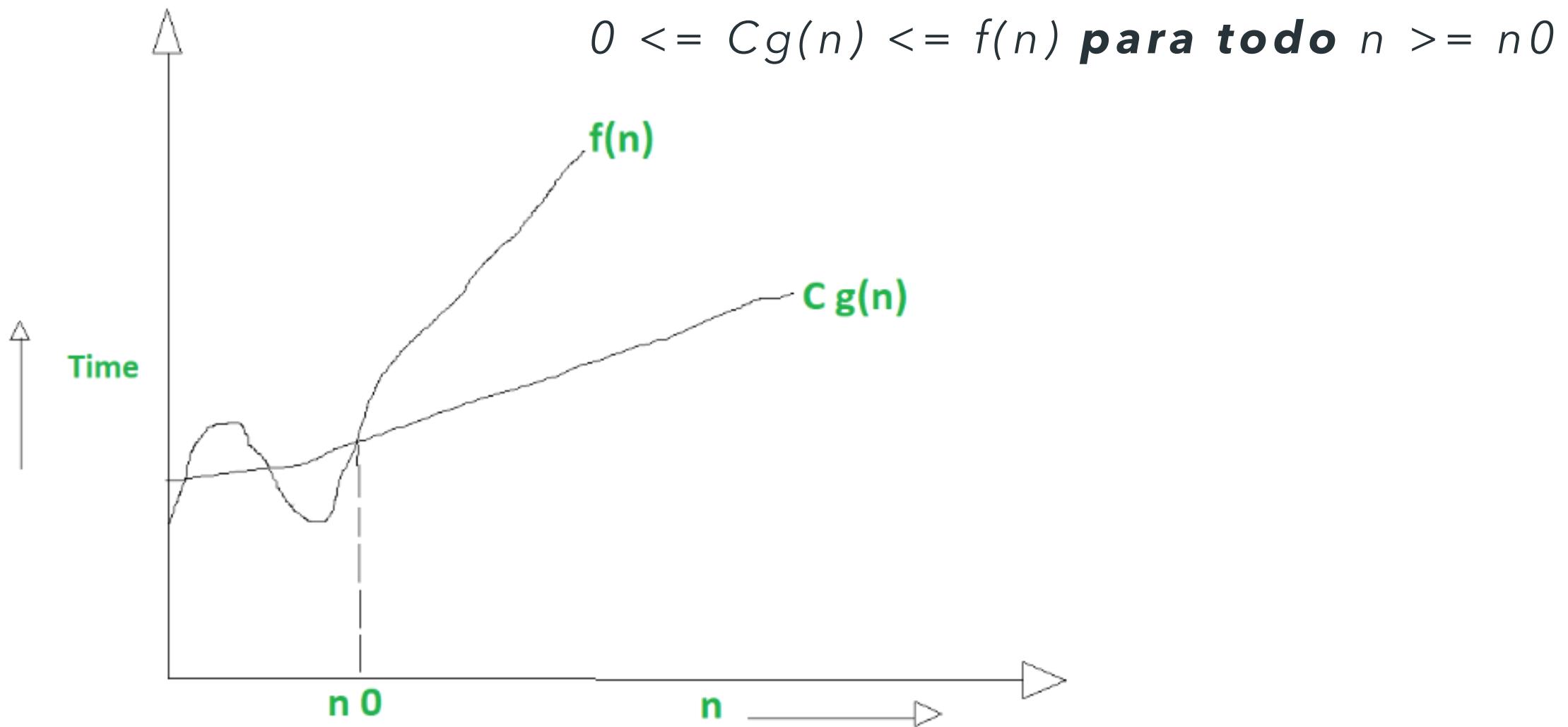


Melhor caso

- **Big Omega $\Omega(n)$** : É o melhor caso, ou seja, quantos passos mínimos serão executados para solucionar o problema



Melhor caso





Caso médio

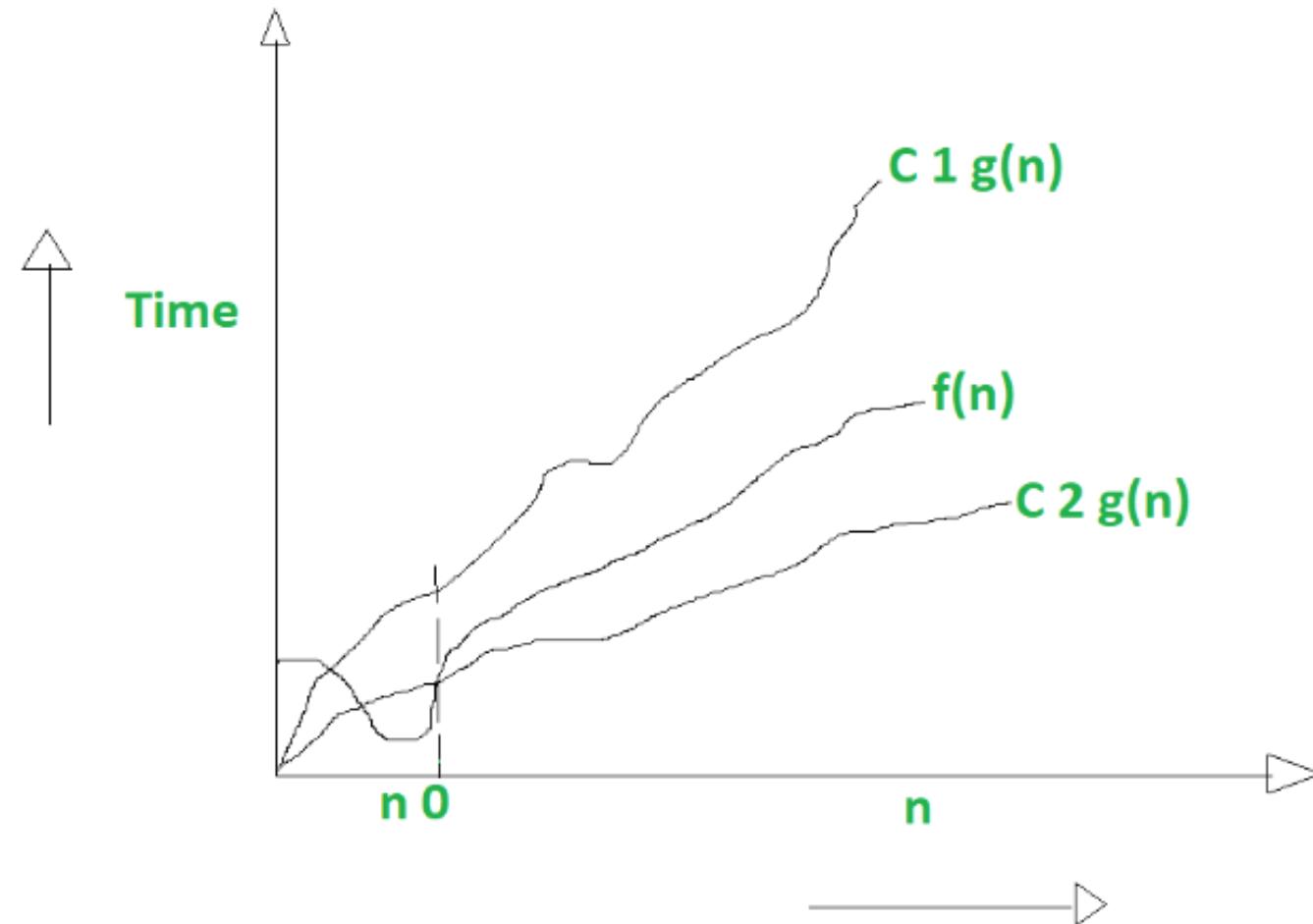
- **Big Theta $\Theta(n)$** : para descobrir a complexidade média de execução
 - Não corresponde a a média aritmética
 - Valor entre as medidas anteriores
 - Gera um valor menos previsível, ente os dois conceitos anteriores.

○ **Big Theta** quer saber a complexidade média de execução (não é que seja a média aritmética mesmo, ela fica entre as medidas anteriores, mas é algo que gera um valor menos previsível), então ele espera que esteja ente os dois conceitos anteriores. Em muitos casos a média (real) é o que queremos saber de verdade principalmente se a média ou próximo disto é o que mais acontecerá.



-

Melhor caso e caso médio



Exemplo inicial

```
public static int aboveMeanCount (double[ ] a, double mean) {  
    int n = a.length,  
        count = 0;  
    for (int i = 0; i < n; i++)  
        if (a [i] > mean)  
            count++;  
    return count;  
}
```

O que faz esse algoritmo?

Qual o desempenho dele?



Exemplo inicial

```
public static int aboveMeanCount (double[ ] a, double mean) {  
    int n = a.length,  
        count = 0;  
    for (int i = 0; i < n; i++)  
        if (a [i] > mean)  
            count++;  
    return count;  
}
```

6 instruções executadas 1 vez



Exemplo inicial

```
public static int aboveMeanCount (double[ ] a, double mean) {  
    int n = a.length,  
        count = 0;  
    for (int i = 0; i < n; i++)  
        if (a [i] > mean)  
            count++;  
    return count;  
}
```

**1 instrução executada
 $n+1$ vezes**



Exemplo inicial

```
public static int aboveMeanCount (double[ ] a, double mean) {  
    int n = a.length,  
        count = 0;  
    for (int i = 0; i < n; i++)  
        if (a [i] > mean)  
            count++;  
    return count;  
}
```

2 instruções executadas n vezes



Exemplo inicial

```
public static int aboveMeanCount (double[ ] a, double mean) {  
    int n = a.length,  
        count = 0;  
    for (int i = 0; i < n; i++)  
        if (a [i] > mean)  
            count++;  
    return count;  
}
```

E essa instrução? Quantas vezes será executada?



Exemplo inicial

```
public static int aboveMeanCount (double[ ] a, double mean) {  
    int n = a.length,  
        count = 0;  
    for (int i = 0; i < n; i++)  
        if (a [i] > mean)  
            count++;  
    return count;  
}
```

**Resposta para quase tudo na
computação:
R= Depende!**



Exemplo inicial

```
public static int aboveMeanCount (double[ ] a, double mean) {  
    int n = a.length,  
        count = 0;  
    for (int i = 0; i < n; i++)  
        if (a [i] > mean)  
            count++;  
    return count;  
}
```

Pior caso:

n-1

vezes

1 vez

Melhor caso:

n/2

Caso médio:

vezes



Exemplo inicial

```
public static int aboveMeanCount (double[ ] a, double mean) {  
    int n = a.length,  
        count = 0;  
    for (int i = 0; i < n; i++)  
        if (a [i] > mean)  
            count++;  
    return count;  
}
```

Desempenho geral

Pior caso: $4n+6$

Melhor caso: $3n+8$

Caso médio: $3,5n + 7$



Exemplo inicial

```
public static int aboveMeanCount (double[ ] a, double mean) {  
    int n = a.length,  
        count = 0;  
    for (int i = 0; i < n; i++)  
        if (a [i] > mean)  
            count++;  
    return count;  
}
```

Desempenho geral

Notação: O(n)

Crescimento linear



Um outro exemplo

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        System.out.println (i + j);
```

Quantas vezes a instrução `System.out.println` será executada?

Logo, esse algoritmo é **O(n^2)**



Funções de desempenho mais comuns

Função	Exemplo
$O(1)$	Imprimir último elemento de um array
$O(\log n)$	Busca binária em um array
$O(n)$	Encontrar maior elemento de um array
$O(n \log n)$	Alguns algoritmos de ordenação de array
$O(n^2)$	Encontrar menor elemento em uma matriz NxN.
$O(2^n)$	Método recursivo da Sequência de Fibonacci



Funções de desempenho mais comuns

Função	Exemplo
O(1)	Imprimir último elemento de um array
	Significa que a quantidade de instruções executadas não depende de algum parâmetro de entrada (como o tamanho do array 'n' no exemplo anterior)



Funções de desempenho mais comuns

Função	Descrição
O(log n)	Regra: quando o n é dividido por 2 dentro do loop
	<pre>while (n > 1) { n = n / 2; S } // while</pre>



Funções de desempenho mais comuns

Função	Descrição
O(n)	Quando o for é executado n vezes (ou $n/2$, $n/3\dots$)
	<pre>for (int i = 0; i < n; i++) { S } // for</pre>



Função	Descrição
O(n log n)	O for é executado 'n vezes' e o while $\log_2 n$
	<pre>int m; for (int i = 0; i < n; i++){ m = n; while (m > 1) { m = m / 2; S } // while } // for</pre>



Funções de desempenho mais comuns

Função	Descrição
$O(n^2)$	Um for dentro de outro, ambos executam n vezes
Exemplo 1	<pre>for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) { S } // for j</pre>

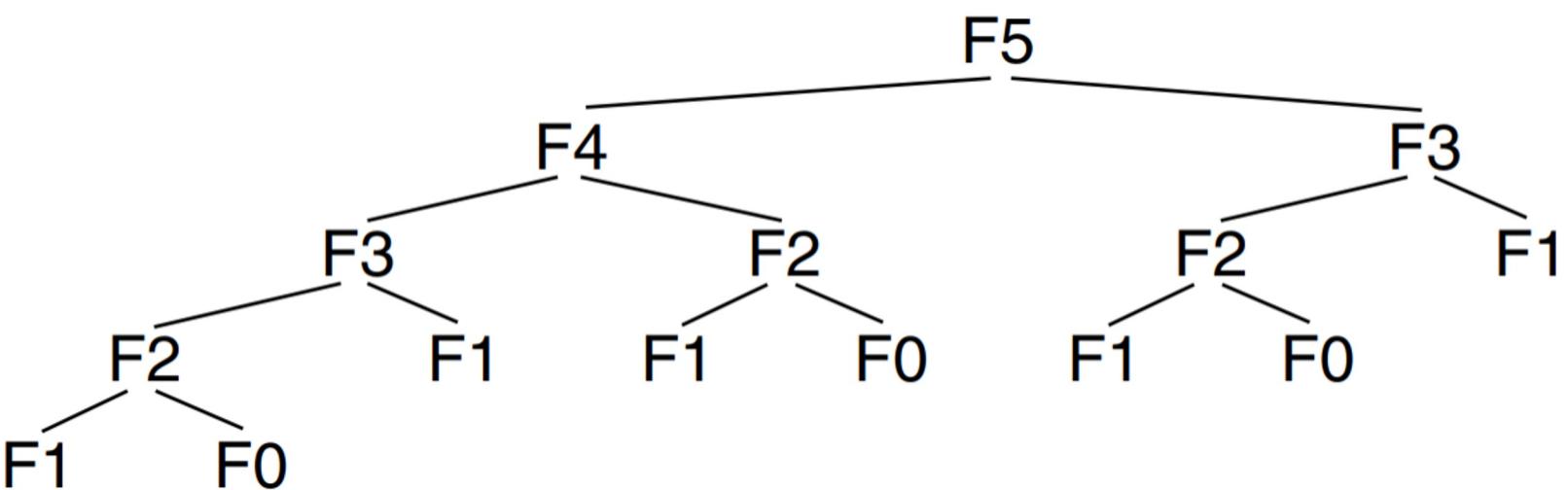


Função	Descrição
O(n²)	Um for dentro de outro, o primeiro executa n vezes e o segundo (n+1)/2. Permanece O(n ²)
Exemplo 2	<pre>for (int i = 0; i < n; i++) for (int k = i; k < n; k++) { S } // for k</pre>

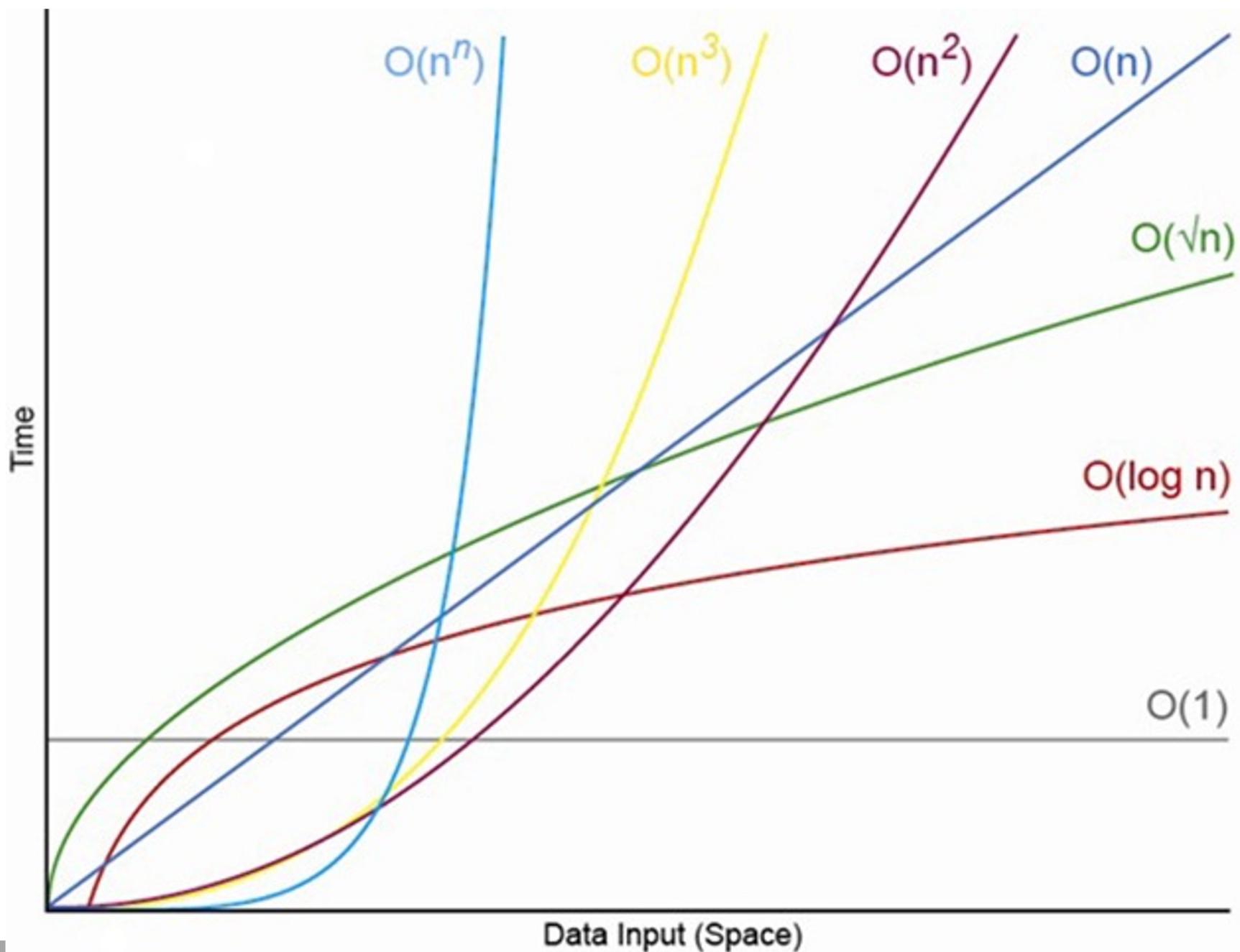


Função	Exemplo
O(2ⁿ)	Encontrar o n-ésimo termo da seq. de Fibonacci (método recursivo)
<p>Crescimento exponencial Pior desempenho entre todos!</p>	<pre>public static int fib(int n) { if(n<=1) return n; return fib(n-1) + fib(n-2); }</pre>



Função	Descrição
O(2ⁿ)	Encontrar o n-ésimo termo da seq. de Fibonacci
	 <p>A recursion tree illustrating the computation of the 5th Fibonacci number (F5). The root node is F5, which branches into F4 and F3. F4 branches into F3 and F2. F3 branches into F2 and F1. F2 branches into F1 and F0. F1 branches into F1 and F0. The leaves of the tree are F1, F0, F1, F0, F1, and F0, representing the six executions required to compute F5.</p> <p> $n=3 \rightarrow 5 \text{ execuções}$ $n=4 \rightarrow 9 \text{ execuções}$ $n=5 \rightarrow 15 \text{ execuções}$ $n=6 \rightarrow 24 \text{ execuções}$ </p>







Ordenação

Algoritmo	Comparações			Movimentações			Espaço
	Melhor	Médio	Pior	Melhor	Médio	Pior	
Bubble	$O(n^2)$			$O(n^2)$			$O(1)$
Selection	$O(n^2)$			$O(n)$			$O(1)$
Insertion	$O(n)$	$O(n^2)$		$O(n)$	$O(n^2)$		$O(1)$
Merge	$O(n \log n)$			-			$O(n)$

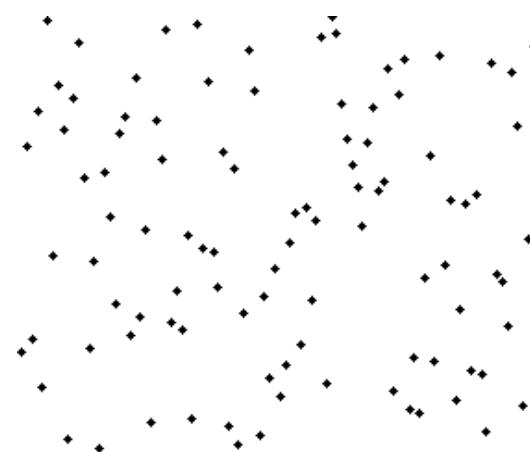


-

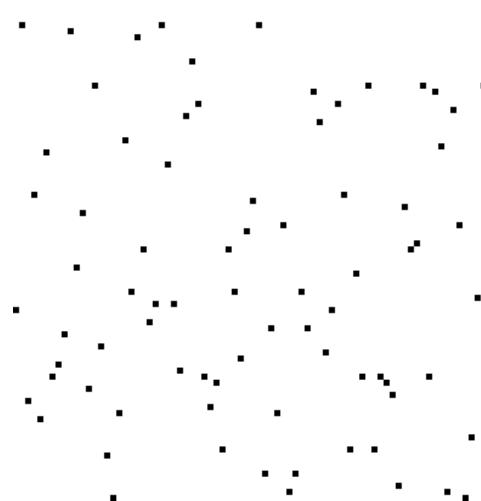
Ordenação

Apresentam desempenho significativamente melhor

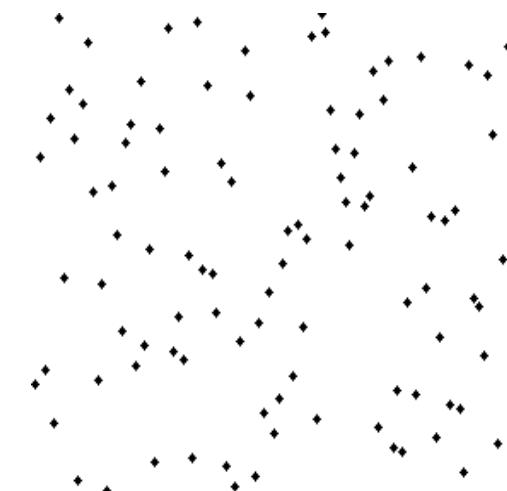
- *Merge sort*



- *Heap sort*



- *Quick sort*



Merge sort

- Criado por **Von Neumann** – matemático húngaro de origem judaica, naturalizado estadunidense – em 1945.
- Contribuições:
 - teoria dos conjuntos,
 - análise funcional,
 - mecânica quântica,
 - ciência da computação,
 - economia,
 - teoria dos jogos,
 - análise numérica,
 - hidrodinâmica das explosões,
 - estatística e muitas outras áreas da matemática.





Merge sort

- Merge sort, ou ordenação por mistura, é um exemplo de algoritmo de ordenação por comparação do tipo dividir-para-conquistar.
- Sua ideia básica consiste em dividir o problema em vários subproblemas e resolver esses subproblemas recursivamente.
- Conquistar significa que, após todos os subproblemas terem sido resolvidos, ocorre a união das resoluções dos subproblemas.
- Considerando que o algoritmo merge sort usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente para certos casos.





Merge sort

- **Vantagens**

- A ordenação de n elementos é feita em tempo proporcional a $n \log n$, independentemente dos dados!
- Comparado a algoritmos mais básicos de ordenação por comparação e troca (*bubble sort*, *insertion sort* e *selection sort*), o *merge* é mais rápido e eficiente quando é utilizado sobre uma grande quantidade de dados.

- **Desvantagens**

- Utiliza funções recursivas;
- Gasto extra de memória, pois o algoritmo cria uma cópia do vetor para cada nível da chamada recursiva, totalizando um uso adicional de memória igual a $O(n \log n)$.





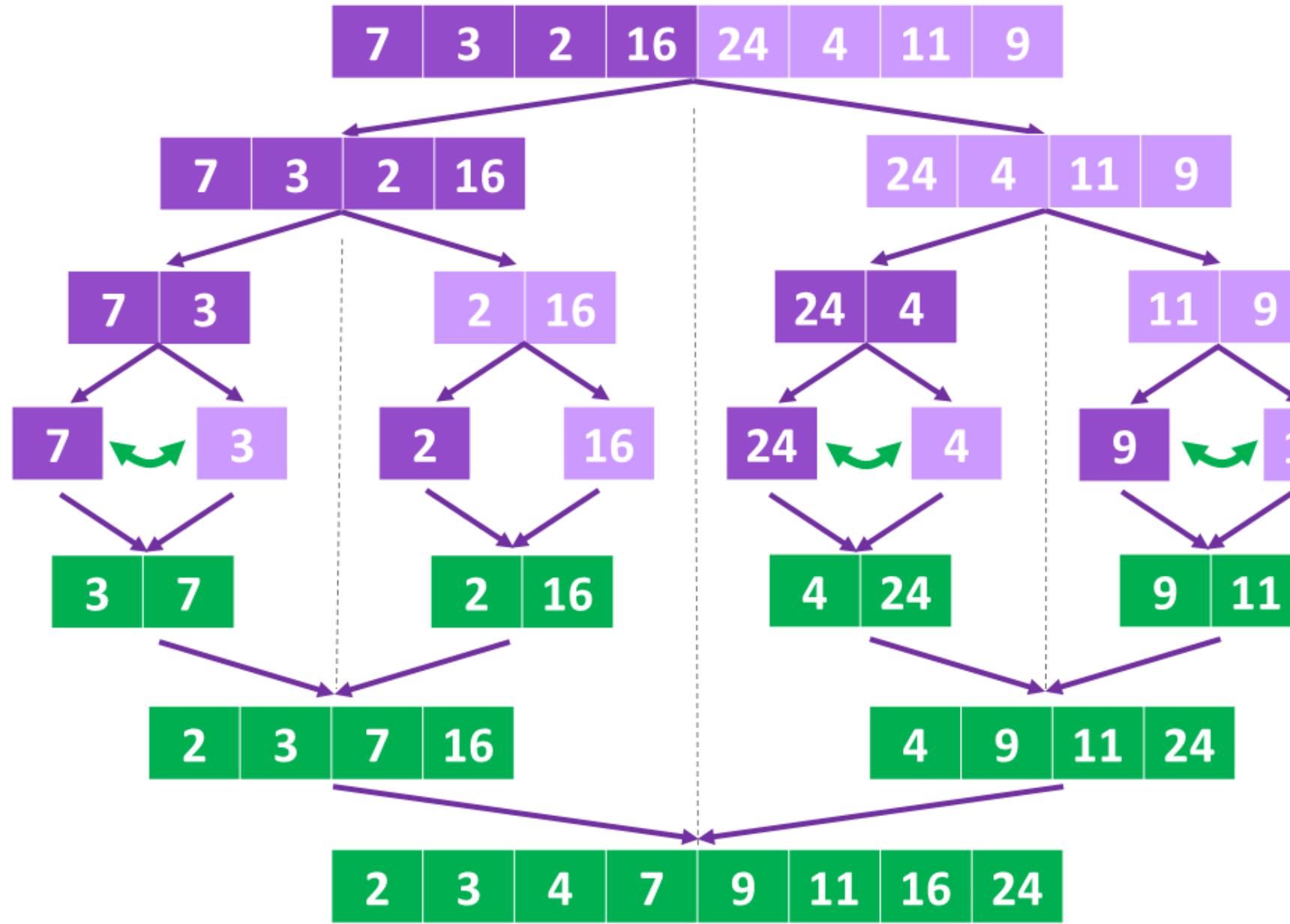
Merge sort

Funcionamento

- Ordenação por intercalação/junção/fusão: mais conhecida como Merge Sort.
- Dividir: divide a sequência de n elementos a serem ordenados em duas subsequências de tamanho $[n/2]$ e $[n/2]$.
- Conquistar: ordena as duas subsequências recursivamente por intercalação.
- Combinar: faz a intercalação das duas sequências ordenadas de modo a produzir a resposta ordenada.



Merge Sort



Step 1:
Split sub-lists in two until you reach pair of values.

Step 2:
Sort/swap pair of values if needed.

Step 3:
Merge and sort sub-lists and repeat process till you merge to the full list.

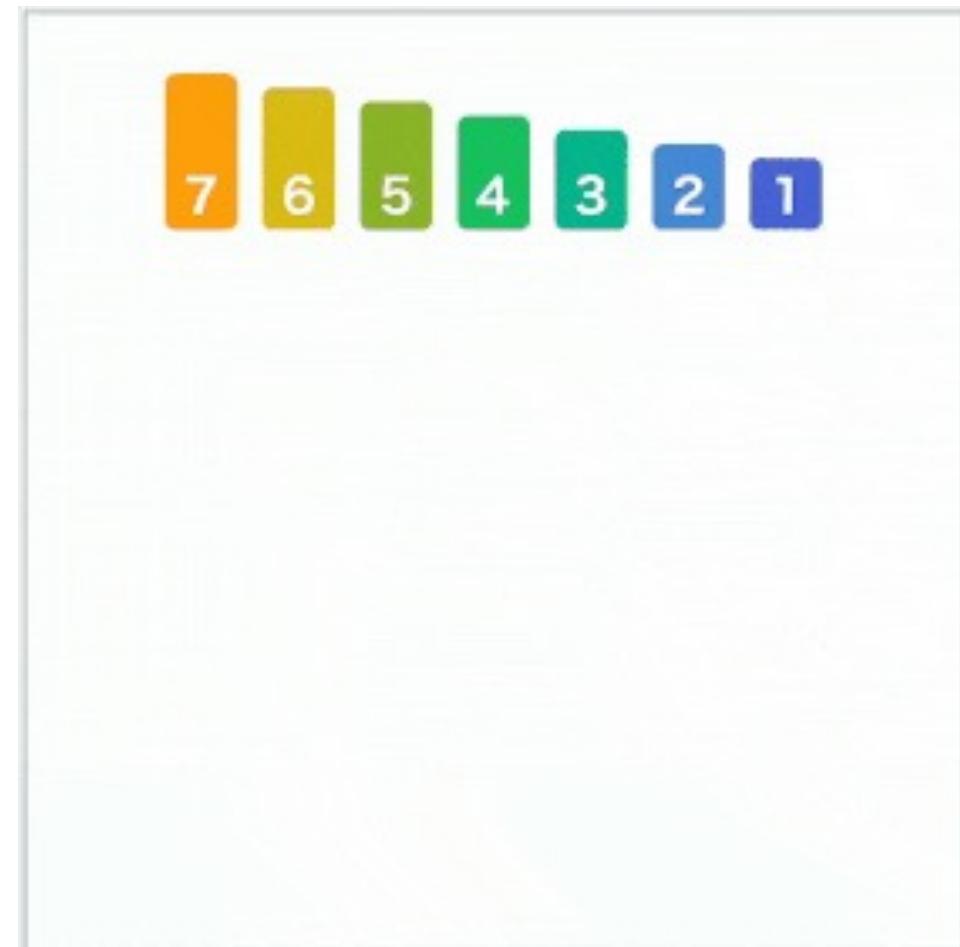


-

Merge sort

Funcionamento

6 5 3 1 8 7 2 4





Merge sort

- **p** índice inicio
- **r** índice fim
- **q** índice meio

MERGESORT (*p*, *r*, *V*)

```
1  se p < r
2    q := [ (p + r) / 2]
3    MERGESORT (p, q, V)
4    MERGESORT (q+1, r, V)
5    INTERCALA (p, q, r, V)
```

INTERCALA (*p*, *q*, *r*, *V*)

```
6  para i := p até q
7    W[i] := V[i]
8  para j := q+1 até r
9    W[r+q+1-j] := V[j]
10  i := p
11  j := r
12  para k := p até r
13    se W[i] ≤ W[j]
14      V[k] := W[i]
15      i := i+1
16    senão V[k] := W[j]
17      j := j-1
```



-
-

Merge sort

```
void mergesort (int p, int r, int v[]) {  
  
    if (p < r-1) {  
  
        int q = (p + r) /2;  
  
        mergesort (p, q, v);  
  
        mergesort (q, r, v);  
  
        intercala (p, q, r, v);  
  
    }  
}
```





Merge sort

```
void intercala (int p, int q, int r, int v[]) {  
    int i, j, *w;  
    w = malloc ((r-p) * sizeof (int));  
  
    for (i = p; i < q; ++i){ w[i-p] = v[i] };  
  
    for (j = q; j < r; ++j){ w[r-p+q-j-1] = v[j] };  
  
    i = 0; j = r-p-1;  
  
    for (int k = p; k < r; ++k){  
        if (w[i] <= w[j]) { v[k] = w[i++] };  
        else v[k] = { w[j--] };  
    }  
    free (w);  
}
```



-
-

Merge sort

```
void mergesortIterativo (int n, int v[]) {  
    int b = 1;  
  
    while (b < n) {  
  
        int p = 0;  
  
        while (p + b < n) {  
  
            int r = p + 2*b;  
  
            if (r > n)  
  
                r = n;  
  
            intercala (p, p+b, r, v);  
  
            p = p + 2*b;  
  
        }  
  
        b = 2*b;  
  
    }  
}
```



Quick sort

Algoritmo **criado** pelo cientista da computação britânico **Charles Antony Richard Hoare em 1964**, por ocasião de sua visita à Universidade de Moscovo como estudante

Hoare estava trabalhando num projeto de tradução de máquina para o National Physical Laboratory e criou o **quicksort** ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que pudesse ser resolvidos de modo mais fácil e rápido. Depois de vários refinamentos, seu trabalho foi publicado em 1962.



-
-

Quick sort

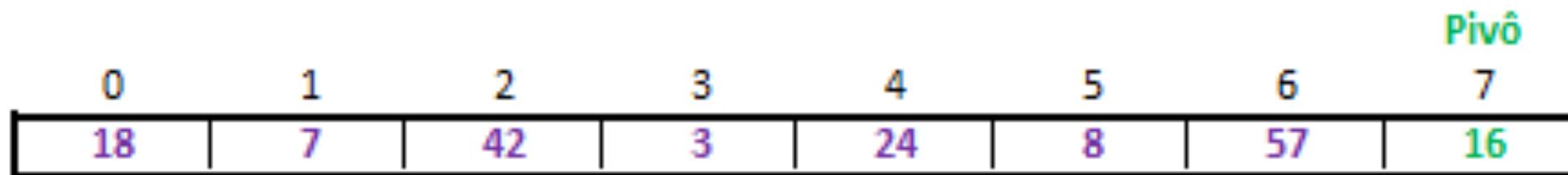
- Adota a estratégia de divisão e conquista.
- Este algoritmo se vale do **particionamento**:
 - Dado um elemento (o pivô)
 - cria-se um vetor tal que
 - todos os menores que o pivô estão à esquerda dele
 - e todos os maiores à direita.





Quick sort

- Dado um elemento (o pivô)
 - cria-se um vetor tal que
 - todos os menores que o pivô estão à esquerda dele
 - e todos os maiores à direita.





Quick sort

1º vetor (com elementos menores que o pivô):

3	7	8	16
---	---	---	----

2º vetor (com elementos maiores que o pivô):

16	24	42	57
----	----	----	----

Juntando os dois vetores:

3	7	8	16	24	42	57
---	---	---	----	----	----	----

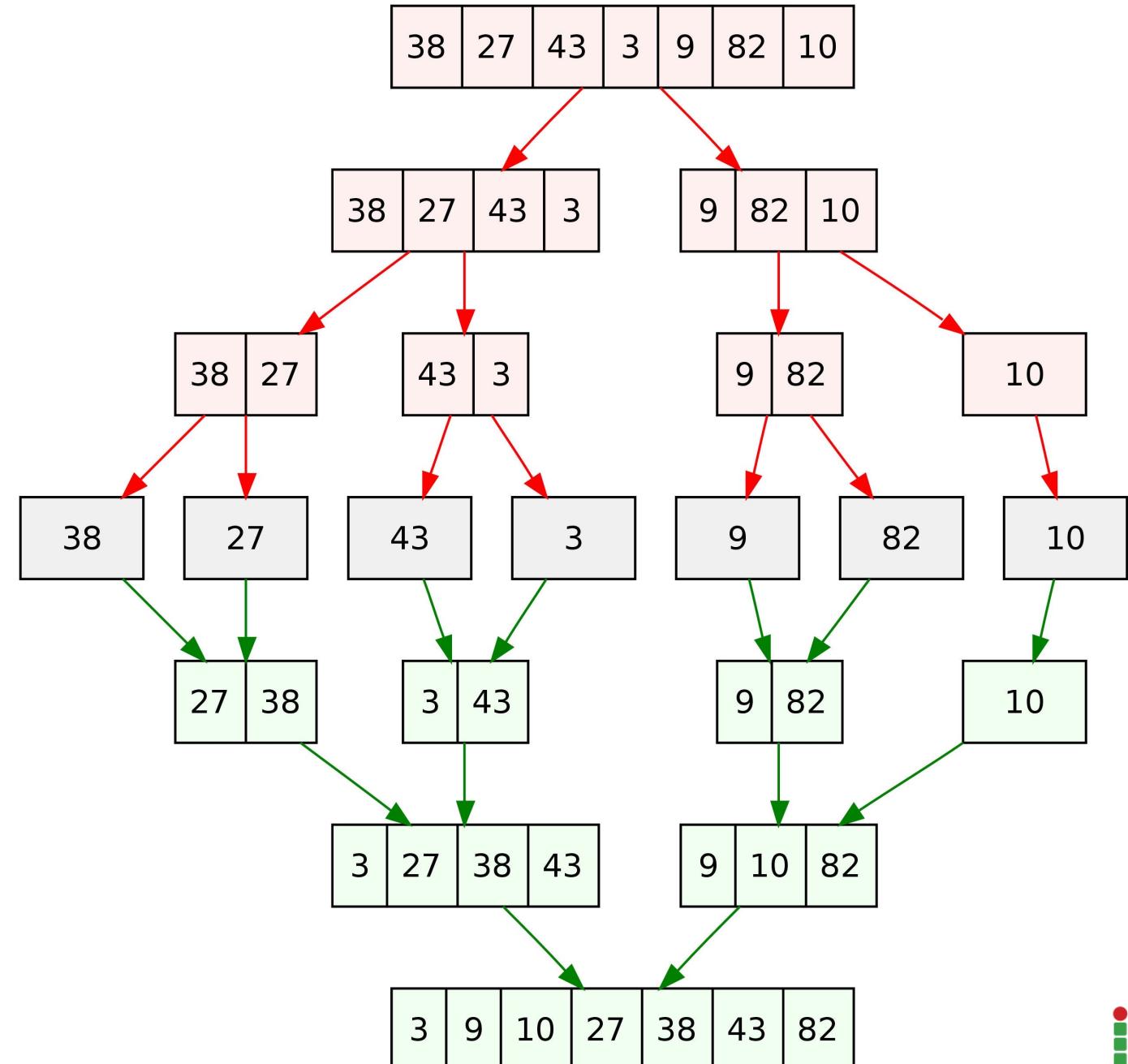


Quick sort

- Percorrido o vetor, o pivô troca de posição com a do maior elemento
- O maior elemento passa a ser o novo pivô

Depois de particionado o vetor:

- chama-se recursivamente o *QuickSort* para cada um dos lados do particionamento, ordenando-os.
- Caso base: $\text{início} < \text{fim}$.



-

Quick sort

Funcionamento

2	6	5	1	4	3
---	---	---	---	---	---





Quick sort

DIVIDE (A, p, r) $\triangleright p \leq r$

- 1 $x := A[r]$ \triangleright pivô
- 2 $i := p-1$
- 3 **para** $j := p$ até $r-1$
- 4 **se** $A[j] \leq x$
- 5 $i := i+1$
- 6 troque $A[i]$ com $A[j]$
- 7 troque $A[i+1]$ com $A[r]$
- 8 **devolva** $i + 1$

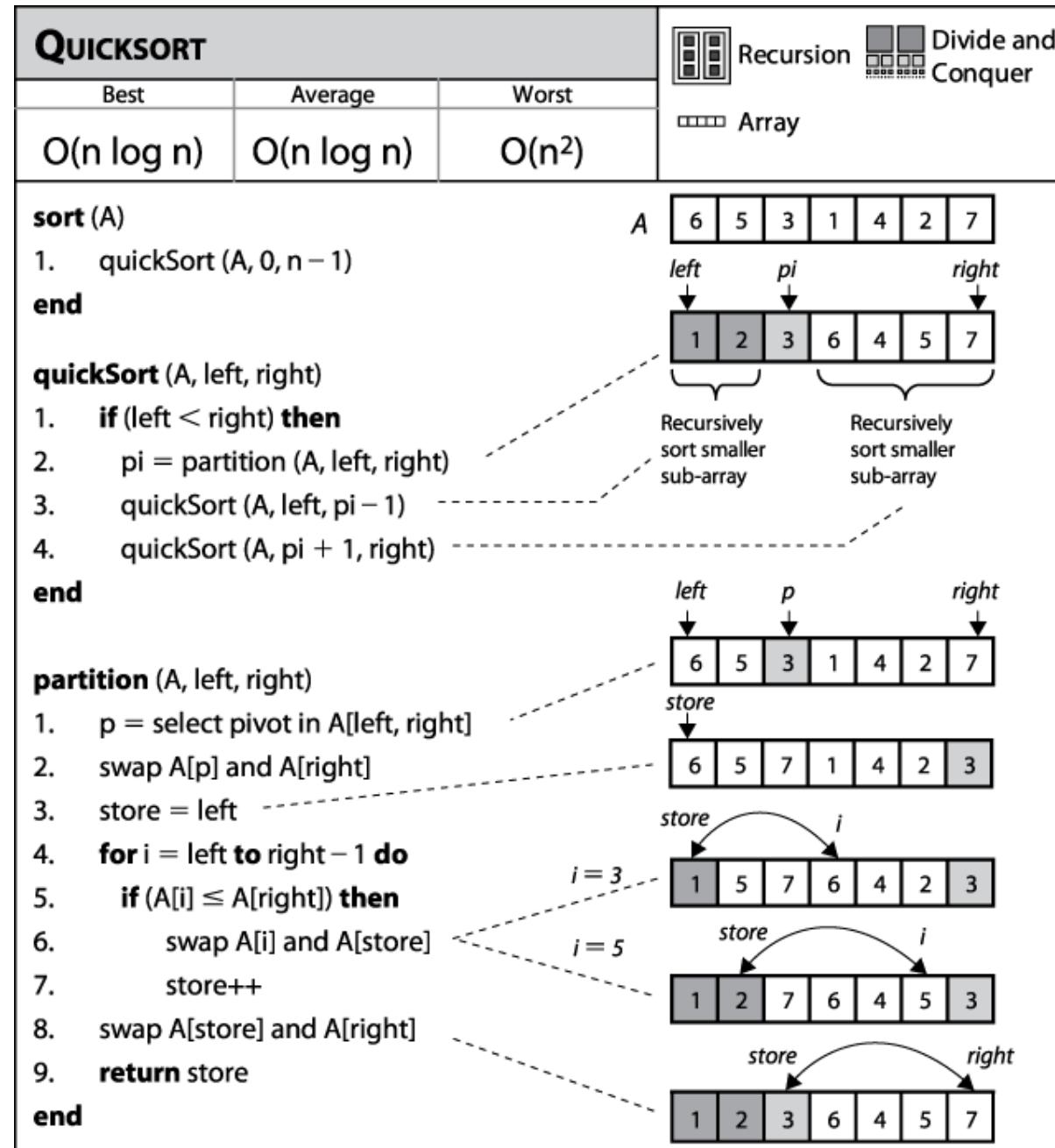
Quicksort (A, p, r) $\triangleright p \leq r+1$

- 1 **se** $p \leq r$
- 2 $q := \underline{\text{Divide}}(A, p, r)$
- 3 **Quicksort** ($A, p, q-1$)
- 4 **Quicksort** ($A, q+1, r$)



Quick sort

oreilly.com/library/view/algorithms-in-a-9780596516246/ch04s04.html





Quick sort

- Recebe vetor $v[p..r]$ com $p \leq r$.
- Rearranja os elementos do vetor e devolve j em $p..r$ tal que $v[p..j-1] \leq v[j] < v[j+1..r]$.

```
int separa (int v[], int p, int r) {  
    int c = v[r]; // pivô  
    int t, j = p;  
for (int k = p; /*A*/ k < r; ++k) {  
    if (v[k] <= c) {  
        t = v[j], v[j] = v[k], v[k] = t;  
        ++j;  
    }  
    t = v[j], v[j] = v[r], v[r] = t;  
    return j;  
}
```





Quick sort

```
void quicksort (int v[], int p, int r) {  
    if (p < r) {  
        int j = separa (v, p, r);  
        quicksort (v, p, j-1);  
        quicksort (v, j+1, r);  
    }  
}
```





Quick sort

É possível **eliminar a segunda chamada recursiva** e **trocar o if por um while**.

Isso produz uma função equivalente ao **quicksort** :

```
void qcksrt (int v[], int p, int r) {  
  
    while (p < r) {  
  
        int j = separa (v, p, r);  
  
        qcksrt (v, p, j-1);  
  
        p = j + 1;  
  
    }  
}
```



Quick sort

A função rearranja o vetor $v[p..r]$, com $p \leq r+1$, de modo que ele fique em ordem crescente.

```
void quicksort_it (int v[], int p, int r) {  
    int j, *pilhap, *pilhar, t;  
    pilhap = malloc ((r-p+1) * sizeof (int));  
    pilhar = malloc ((r-p+1) * sizeof (int));  
    pilhap[0] = p;  
    pilhar[0] = r; t = 0;  
    while (t >= 0) {  
        p = pilhap[t];  
        r = pilhar[t]; --t;  
        if (p < r) {  
            j = separa (v, p, r); ++t;  
            pilhap[t] = p;  
            pilhar[t] = j-1; ++t;  
            pilhap[t] = j+1;  
            pilhar[t] = r;  
        }  
    }  
}
```





Quick sort

```
void qcksrt (int v[], int p, int r) {  
    pilhap[0] = p; pilhar[0] = r; t = 0;  
    while (t >= 0) {  
        p = pilhap[t]; r = pilhar[t]; --t;  
        while (p < r) {  
            j = separa (v, p, r); ++t;  
            pilhap[t] = p;  
            pilhar[t] = j-1; p = j + 1;  
        }  
    }  
}
```

Pode-se deixar de empilhar o segundo par de índices, uma vez que ele seria desempilhado logo em seguida



-

Quick sort

Um **mau pivô** tem um **efeito negativo** sobre o **número de comparações** que **QUICKSORT** faz.





Quick sort

Para tentar evitar um mau pivô, podemos **escolher o pivô aleatoriamente**.

RAND-DIVIDE (A, p, r)

- 1 $i := \text{RANDOM}(p, r)$
- 2 troca $A[i]$ com $A[r]$
- 3 $q := \text{DIVIDE}(A, p, r)$
- 4 devolva q

RAND-QUICKSORT (A, p, r)

- 1 se $p \leq r$
- 2 $q := \text{RAND-DIVIDE}(A, p, r)$
- 3 $\text{RAND-QUICKSORT}(A, p, q-1)$
- 4 $\text{RAND-QUICKSORT}(A, q+1, r)$





Quick sort

1º vetor (com elementos menores que o pivô):

3	7	8	16
---	---	---	----

2º vetor (com elementos maiores que o pivô):

16	24	42	57
----	----	----	----

Juntando os dois vetores:

3	7	8	16	24	42	57
---	---	---	----	----	----	----



-

Heap sort

Algoritmo criado por **Robert W Floyd e J.W.J Williams** em 1964





Heap sort

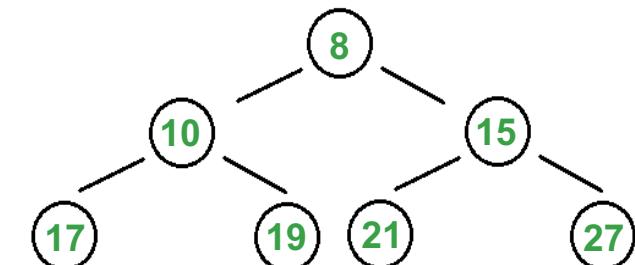
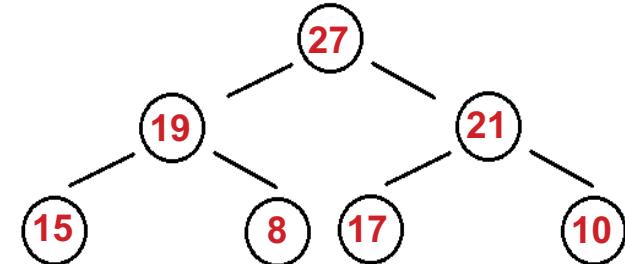
- O *heap sort* é um algoritmo de ordenação do tipo ordenação por seleção. Ele resolve o problema da ordenação de um vetor, deixando-o em ordem crescente.
- É semelhante ao tipo de seleção, onde primeiro encontramos o elemento máximo e o colocamos no final. Repetimos o mesmo processo para os elementos restantes.





Heap sort

- Existem vários tipos de heap:
 - **heap de máximo**, nos quais o maior número é localizado na “primeira casa”, chamada de raiz da árvore binária.
 - **heap de mínimo**, nos quais o menor número é localizado na raiz da árvore binária.



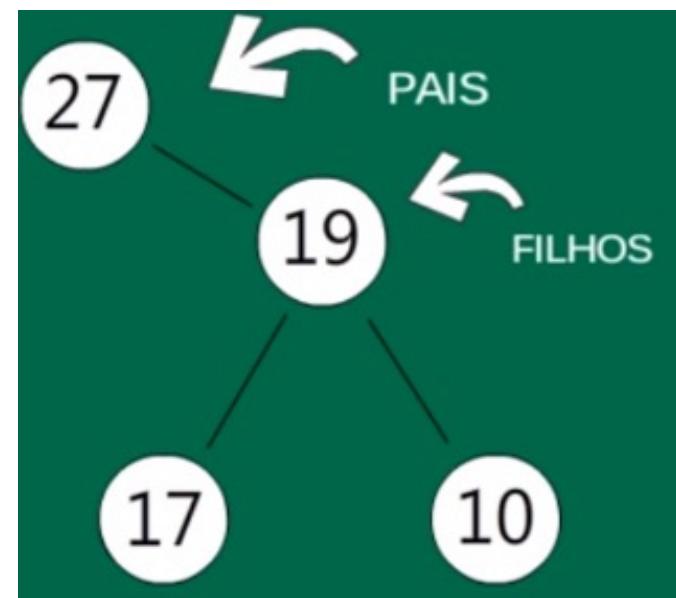


Heap sort

Para uma estrutura ser considerada um *heap* de máximo, duas regras devem ser seguidas.

1^a. Os “pais” sempre devem ser maiores que os filhos.

2^a. A árvore deve sempre estar completa, exceto pela última linha, na parte da direita.



Heap sort

Heap é uma estrutura de dados **útil** quando é necessário **remover repetidamente o elemento com a prioridade mais alta (ou mais baixa)**, uma vez que tal elemento fica sempre na raiz.





Heap sort

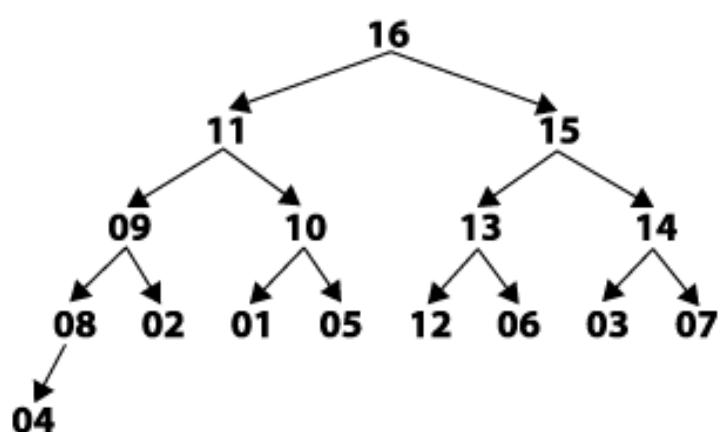
Cada elemento do *heap* é um **NÓ**.

Para cada **NÓ** i , tem-se que:

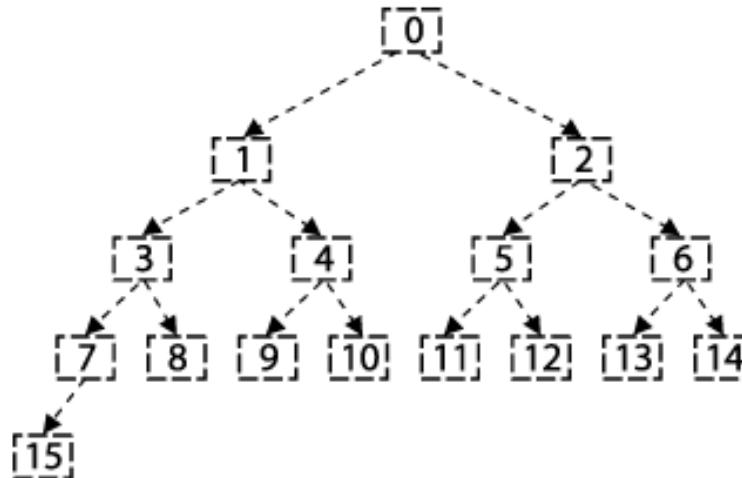
- O filho esquerdo de i é **$2*i + 1$**
- O filho direito de i é **$2*i + 2$**
- O pai de i é **$(i-1)/2$**
- Para todo $i = 2, 3, \dots, m$ (m é a quantidade de elementos do *heap*), tem-se que: $\text{Vetor}[(i-1)/2] \geq \text{Vetor}[i]$ (o pai sempre será maior que os filhos).



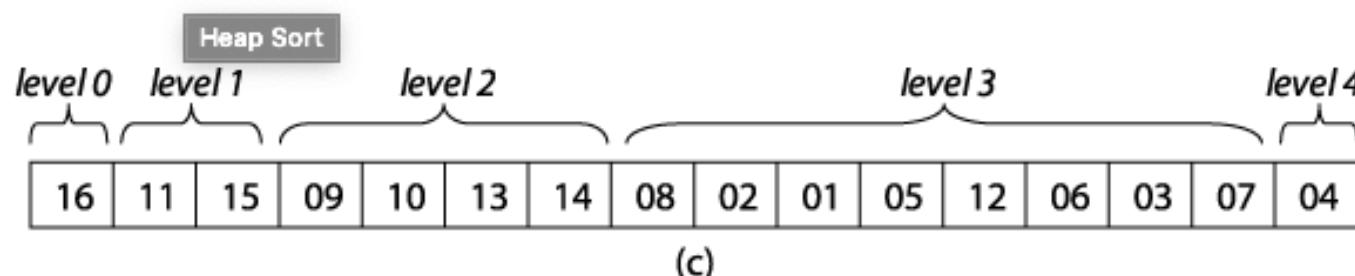
Heap sort



(a)



(b)



(c)

Figure 4-15. (a) Sample heap of 16 unique elements; (b) labels of these elements; (c) heap stored in an array





Heap sort

- O algoritmo **resolve o problema da ordenação usando um max-heap.**
- A rotina **Constrói-Max-Heap** transforma $A[1..n]$ em um max-heap e a rotina **Corrigie-Descendo** transforma o vetor $A[1..m-1]$ em max-heap supondo que as subárvores com raízes 2 e 3 são max-heaps.

Heapsort (A, n)

```
1 Constrói-Max-Heap (A, n)
2 para m := n, n-1, ..., 2
3   A[1] := A[m]
4 Corrigie-Descendo (A, m-1, 1)
```

Constrói-Max-Heap (A, n)

```
1 para i :=  $\lfloor n/2 \rfloor$  decrescendo até 1
2 Corrigie-Descendo (A, n, i)
```

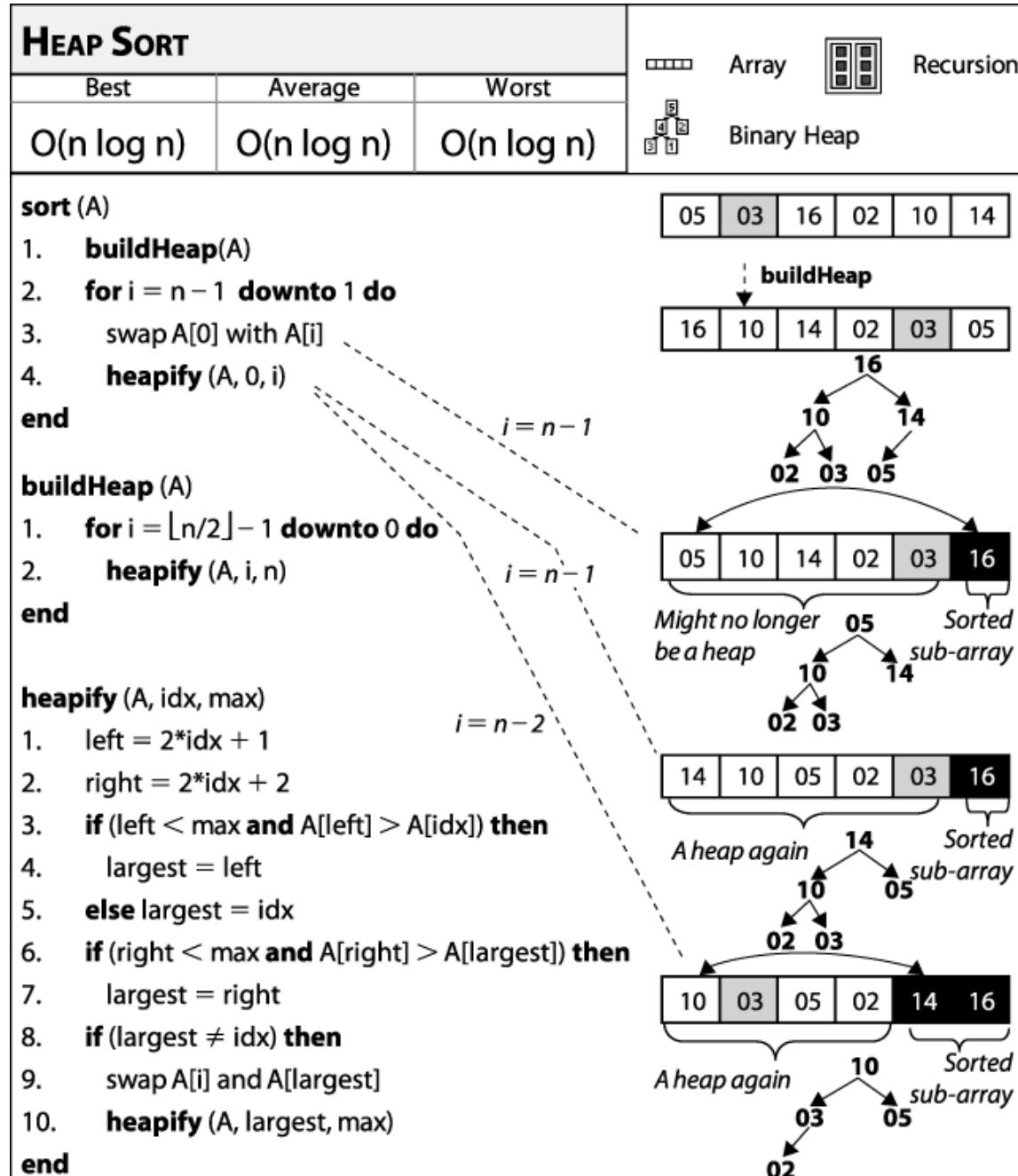
Corrigie-Descendo (A, n, i)

```
1 j := i
2 enquanto  $2j \leq n$ 
3   f :=  $2j$ 
4   se  $f < n$  e  $A[f] < A[f+1]$ 
5     f := f+1
6   se  $A[j] \geq A[f]$ 
7     j := n
8   senão troque  $A[j] := A[f]$ 
9     j := f
```



Heap sort

Em inglês, o algoritmo é conhecido como **HEAPIFY**, ou **SIEVE**, ou **Fix-Down**, ou **SHAKE-DOWN**. Em português, poderíamos usar o nome **PENEIRA**, mas vou preferir **CORRIGE-DESCENDO**, porque é mais descriptivo.



-
-

Heap sort

```
void constroiHeap (int m, int v[]) {  
    for (int k = 1; k < m; ++k) {  
        // v[1..k] é um heap  
        int f = k+1;  
        while (f > 1 && v[f/2] < v[f]) {  
            troca (v[f/2], v[f]);  
            f /= 2;  
        }  
    }  
}
```





Heap sort

```
void peneira (int m, int v[]) {  
    int p = 1, f = 2, x = v[1];  
    while (f <= m) {  
        if (f < m && v[f] < v[f+1]) ++f;  
        if (x >= v[f]) break;  
        v[p] = v[f]; p = f, f = 2*p;  
    }  
    v[p] = x;  
}
```



-
-

Heap sort

```
void heapsort (int n, int v[]) {  
    constroiHeap (n, v);  
    for (int m = n; m >= 2; --m) {  
        troca (v[1], v[m]);  
        peneira (m-1, v);  
    }  
}
```





Bibliografia

BARNES, David J.; KOLLING, Michael. **Programação orientada a objetos com Java.** 4. ed. São Paulo: Prentice Hall - Br, 2009.

Aditya Y. Bhargava. **Entendendo Algoritmos.** Novatec Editora. ISBN 9788575226629.

Algoritmos. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliford Stein. Campus.

Algorithms. Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani. McGraw Hill.

Concrete Mathematics: A Foundation for Computer Science (2nd Edition). Ronald L. Graham, Donald E. Knuth, Oren Patashnik. Addison Wesley.

DEITEL, Paul; DEITEL, Harvey. **Java: como programar.** 10. ed. São Paulo: Pearson, 2017. ISBN 9788543004792. Disponível em: <https://ifsp.bv3.digitalpages.com.br/users/publications/9788543004792>. Acesso em: 14 jun. 2019.

Helder da Rocha.. **Gerência de memória em Java.** Argonavis: 2005.

Sierra, Kathy. **Use A Cabeça Java.** S.l: Alta Books, 2009. ISBN: 8576081733





**INSTITUTO
FEDERAL**

São Paulo

Câmpus
São Paulo

Obrigado!

Gustavo Fortunato Puga

gustavo.puga@ifsp.edu.br

