# Estrutura de Dados

Ordenação (Parte 1 / 3)

# Sumário

# 1. Definição

Ordenação

- Ato de colocar os **elementos** em uma **ordem predefinida** conforme algum **critério** comum **relacionado** com esses elementos.

- **Operação fundamental** em computação:
  - Muitos programas utilizam-na como **etapa intermediária**.

- **Melhor algoritmo** de ordenação **depende**:
  - Número de **elementos** a serem ordenados.
  - **Extensão** em que os elementos já estão ordenados.
  - **Restrições**: dispositivo de armazenamento (memória principal).

# 1. Definição

- **Problema**: Rearranjar um *array* em **ordem crescente.**
  - A[1 . . n] é crescente se A[1] ≤ · · · ≤ A[n]

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 22 | 33 | 33 | 33 | 44 | 55 | 11 | 99 | 22 | 55 | 77 |

**Não está** na ordem crescente

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 22 | 22 | 33 | 33 | 33 | 44 | 55 | 55 | 77 | 99 |

**Está** na ordem crescente

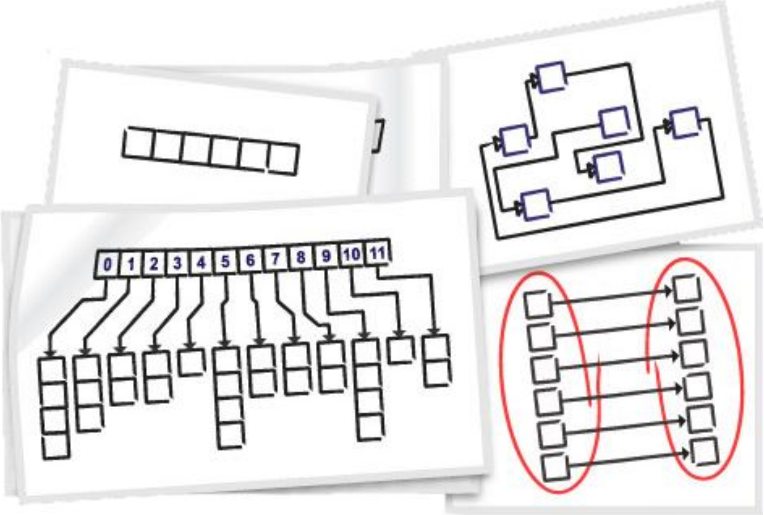
# 1. Definição

Ordenação

- Muitos **algoritmos conhecidos** com complexidade **O(*n log n*):**
  - *Mergesort*
  - *Heapsort*
  - *Quicksort*

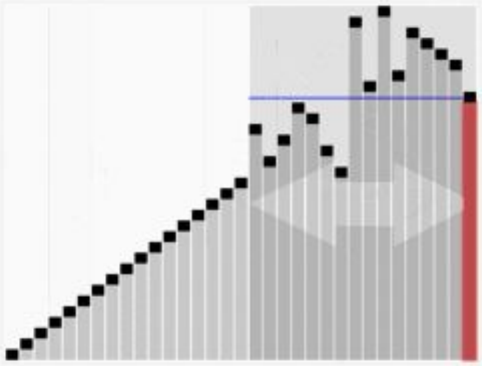
- No entanto, **algoritmos** com complexidade assintótica pior – tipicamente **O($n^2$)** – podem ser **mais eficientes** para ***n* pequeno**:
  - *Bublesort*
  - *Insertion Sort*

# 2. Métodos Gerais




- Existem **três métodos** (paradigmas, técnicas) **básicos** que podem ser usados para **ordenar** elementos de um **vetor**.
  1. Por troca
  2. Por seleção
  3. Por inserção

# 3. Algoritmos simples



- Existem **muitos algoritmos** implementados segundo cada um dos **três métodos** (paradigmas, técnicas) básicos vistos anteriormente.

- Cada um deles tem seus próprios méritos.

- Para **exemplificar** os algoritmos de ordenação, será apresentado **um algoritmo simples** para **cada** um dos métodos explicados.

# 3.1. Ordenação por troca

*Bublesort* (Ordenação bolha)

6 5 3 1 8 7 2 4

- O algoritmo ***bublesort*** funciona da seguinte forma:
  - **Compare** cada par de **elementos adjacentes** do *array*.
  - **Trocá-los** se eles estão na ordem errada.
  - **Repetir** as **etapas acima** para o restante dos elementos do *array* (começando na segunda posição e avançando uma posição de cada vez)

# 3.1. Ordenação por troca

## *Bublesort* (Ordenação bolha)

**Exemplo:** A= {10,9,7,6} e n = 4

```
proc bolha(Array A[0..n-1], n) {
   para i ← 0 ate n-2 fazer
      para j ← 0 ate (n–i)-2 fazer {
  se (A[j] > A[j+1]) então { /* Compara elementos adj.*/
     aux ← A[j];
     A[j] ← A[j+1];          /* Troca de valores */
     A[j+1] ← aux;           /* Troca de valores */
        }
      }
   }
}
```

***Bublesort*** (versão crescente)

# 3.1. Ordenação por troca

**Dado** V=[10,9,7,6] e n=4 como entrada para a função *bublesort*, quais **trocas** o algoritmo realiza? (utilize a versão crescente)

| i | j | V[0] | V[1] | V[2] | V[3] | Troca |
|---|---|---|---|---|---|---|
| 0 | 0 | **10** | 9 | 7 | 6 | V[0] com v[1] |
| 0 | 1 | 9 | **10** | 7 | 6 | V[1] com v[2] |
| 0 | 2 | 9 | 7 | **10** | 6 | V[2] com v[3] |
| | | 9 | 7 | 6 | **10** | Fim i=0 |
| 1 | 0 | **9** | 7 | 6 | 10 | V[0] com v[1] |
| 1 | 1 | 7 | **9** | 6 | 10 | V[1] com v[2] |
| | | 7 | 6 | **9** | 10 | Fim i=1 |
| 2 | 0 | **7** | 6 | 9 | 10 | V[0] com v[1] |
| | | 6 | **7** | 9 | 10 | Fim i=2 |

# 3.1. Ordenação por troca

## Análise de Complexidade - *Bublesort*

```
proc bolha(Array A[0..n-1], n) {
    para i ← 0 ate n-2 fazer
        para j ← 0 ate (n–i)-2 fazer {
    se (A[j] > A[j+1]) então {
        aux ← A[j];
        A[j] ← A[j+1];
        A[j+1] ← aux;
            }
          }
      }
}
```

| i | j | Val. de J |
|---|---|---|
| n-2 | 0 | 1 |
| n-3 | 0,1 | 2 |
| ... | ... | ... |
| 0 | 0,1,...,n-2 | n-1 |

quantidade de valores de j

$$S_n = \frac{\boxed{n} \cdot (a_1 + a_n)}{2}$$

$$S_n = \frac{(n-1)\,n}{2}$$

$$S_n = \frac{n^2 - n}{2}$$

# 3.1. Ordenação por troca

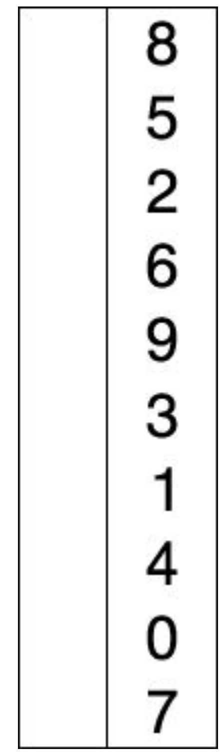
Análise de Complexidade - *Bublesort*

- Complexidade **de tempo** (pior caso):
  - T(n) = **O($n^2$)**
- Complexidade **de espaço** (pior caso):
  - S(n) = **O(n)**

# 3.2. Ordenação por seleção

*Selection Sort*

- O algoritmo ***Selection Sort*** funciona da seguinte forma:
  - **Encontre** o **valor mínimo** entre os elementos do *array*.
  - **Troque-o** com o valor na **primeira posição**.
  - **Repetir** as **etapas acima** para o restante dos elementos do array (começando na segunda posição e avançando uma posição de cada vez)



# 3.2. Ordenação por seleção

*Selection Sort* - **Exemplo:** A= {10,9,7,6} e n = 4

```
01. proc selecao(Array A[0..n-1], n) {
02.  para i ← 0 ate n-2 fazer {
03.   c ← i;
04.   t ← A[i];
05.   para j ← i+1 ate n-1 fazer {
06.     se (A[j] < t) então {
07.       c ← j;
08.       t ← A[j];
09.     }
10.   }
11.   A[c] ← A[i];
12.   A[i] ← t;
13.  }
14. }
```

| i | j | Val. de J |
|---|---|---|
| n-2 | n-1 | 1 |
| n-3 | n-2, n-1 | 2 |
| ... | ... | ... |
| 1 | 2,...,n-1 | n-2 |
| 0 | 1,...,n-1 | n-1 |

**Passos:**

1 – **Seleção** do elemento **$t_1$** (linhas 03-04)

2 – **Identificação** do **menor elemento – $t_2$**(linhas 05-10)

3 – Troca dos elementos **$t_1$** com **$t_2$** (linhas 11-12)

# 3.2. Ordenação por seleção

*Selection Sort*

- Complexidade **de tempo** (pior caso):
  - T(n) = $\mathbf{O(\boldsymbol{n}^2)}$

- Complexidade **de espaço** (pior caso):
  - S(n) = $\mathbf{O(\boldsymbol{n})}$

# 3.3. Ordenação por inserção

*Insertion Sort*

6 5 3 1 8 7 2 4

- O algoritmo ***insertion sort*** funciona da seguinte forma:

1. **Selecione** o **primeiro elemento** do *array*.

2. **Compare** com os **elementos à esquerda** e **insira** o elemento na **ordem correta.**

3. **Repetir** as **etapas acima** para o restante dos elementos do *array* (começando na segunda posição e avançando uma posição de cada vez)

# 3.3. Ordenação por inserção

Insertion Sort

6 5 3 1 8 7 2 4

**Exemplo:** Tente você para A= {10,9,7,6}

```
01. proc insert(Array A[0..n-1], n) {
02.  para i ← 1 ate n-1 fazer {
03.    t ← A[i];
04.    j ← i-1;
05.    enquanto (j ≥ 0 e t < A[j]) fazer {
06.      A[j+1] ← A[j];
07.      j ← j -1;
08.    }
09.    A[j+1] ← t;
10.  }
11. }
```

**Passos:**

1 – **Seleção** do elemento **t** (linha 03)

2 – **j** é o índice do **elemento anterior** (linha 04)

3 – **Comparação** do elemento **t** com todos os **elementos anteriores**. (linha 05-08)

4 – **Deslocamento** de elementos para direita do array (linha 06)

5 – **Inclusão** do elemento **t** na **posição correta**. (linha 09)

# 3.3. Ordenação por inserção

*Insertion Sort*

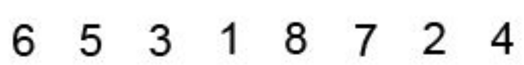
- Complexidade **de tempo** (pior caso):
  - T(n) = $\mathbf{O}(\boldsymbol{n}^2)$

- • Complexidade **de espaço** (pior caso):
  - S(n) = $\mathbf{O}(\boldsymbol{n})$

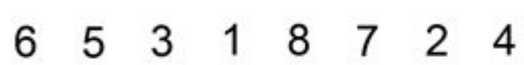
# 3.4. Resumo

- Segue abaixo um **resumo** dos algoritmos simples apresentados:

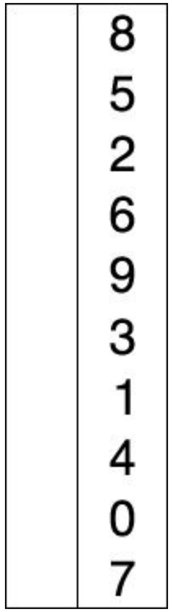
| Algoritmo | Paradigma | Tempo | Espaço |
|---|---|---|---|
| Bublesort | Troca | $O(n^2)$ | O(n) |
| Selection Sort | Seleção | $O(n^2)$ | O(n) |
| Insertion Sort | Inserção | $O(n^2)$ | O(n) |




*Bublesort*




*Insertion Sort*

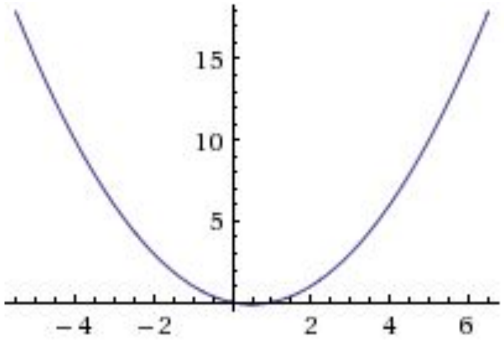



*Selection Sort*

# 4. Algoritmos rápidos

- Os **algoritmos simples** possuem a seguinte **limitação**:
  - São **processados** numa grandeza de tempo **O(*n*$^2$)**

- Para um grande número de dados (***n* grande**)
  - Ordenação será **lenta**
  - **Pode**, em certos casos, **ser inviável**
  - n=10, tempo=100; n=100, tempo=1000.




- Alternativamente serão apresentados dois **algoritmos** efetivamente **mais rápidos** para realizar a **ordenação**:
  - ***Mergesort***
  - ***Quicksort***

# Estrutura de Dados

Ordenação (Parte 2 / 3)

# 4.1. Mergesort

- O ***Mergesort*** é um algoritmo desenvolvido sob um **método** (técnica, princípio, paradigma) **diferente** dos vistos para os **algoritmos simples**.

- Denominado **dividir para conquistar**:

    - **Dividir** um **problema maior recursivamente** em **problemas menores** até que o problema possa ser resolvido diretamente.

    - **A solução do problema** inicial é então dada por meio da **combinação** dos resultados de todos os **problemas menores** computados.

# 4.1. Mergesort

- Para o caso do algoritmo **Mergesort**, podemos entender os **três passos** úteis do método **dividir para conquistar** assim:

  **1) Dividir a tarefa** em pequenas **subtarefas**.

  **2) Conquistar: resolver cada subtarefa** aplicando o **algoritmo recursivamente** a cada uma.

  **3) Combinar** as **soluções** das **subtarefas** construindo assim a **solução do problema** como um todo.

- Os algoritmos do tipo **dividir para conquistar** são **tipicamente recursivos**.

- Na **análise** de **algoritmos recursivos**
  - Os **limites de complexidade** precisam ser determinados resolvendo **recorrências**.

# 4.1. Mergesort

- Em termos algorítmicos, **considere** um ***array* A[1..n]**.

6 5 3 1 8 7 2 4

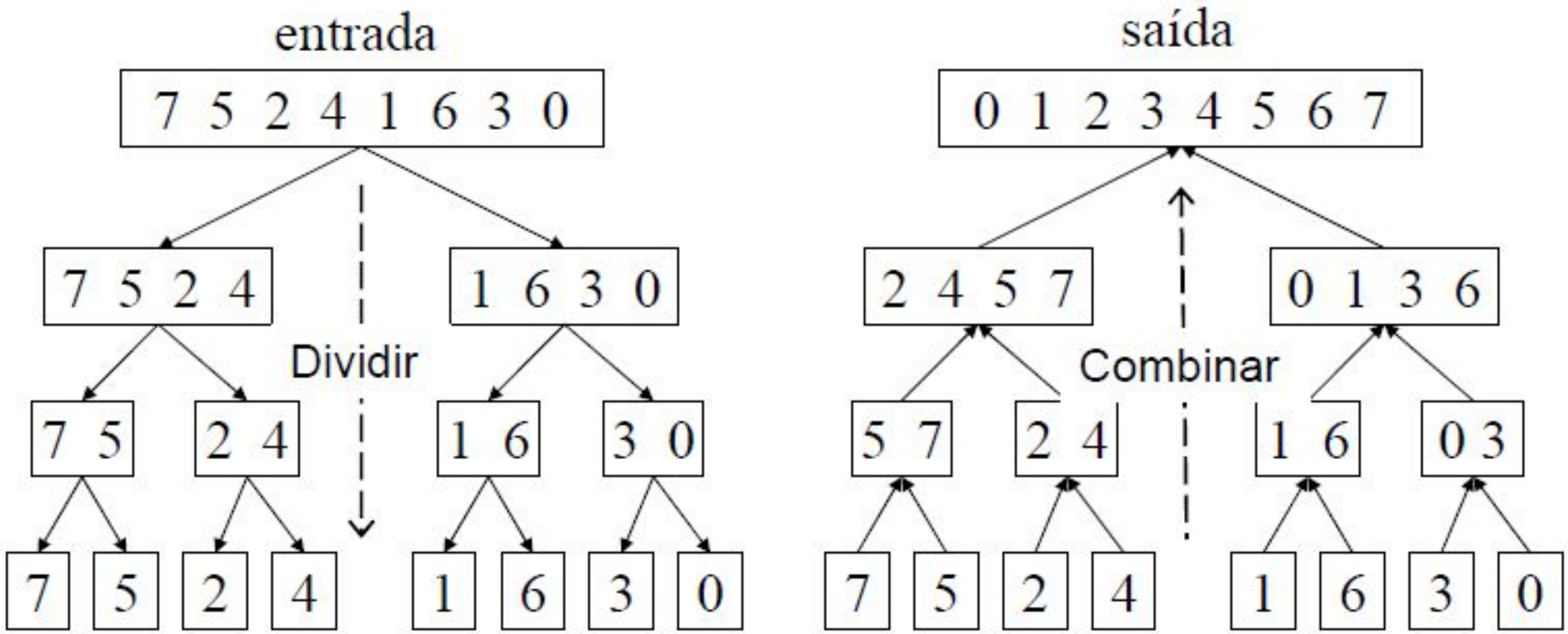
- O algoritmo consiste das seguintes fases:

  **1) Dividir** A em 2 sub-coleções de tamanho ≈ n/2
  **2) Conquistar**: ordenar cada sub-coleção chamando MergeSort recursivamente
  **3) Combinar** as sub-coleções ordenadas formando uma única coleção ordenada

- **Se** uma **sub-coleção** tem apenas **um elemento**, ela já está **ordenada** e configura o **caso base** do algoritmo

# 4.1. Mergesort

Exemplo gráfico



# 4.1. Mergesort

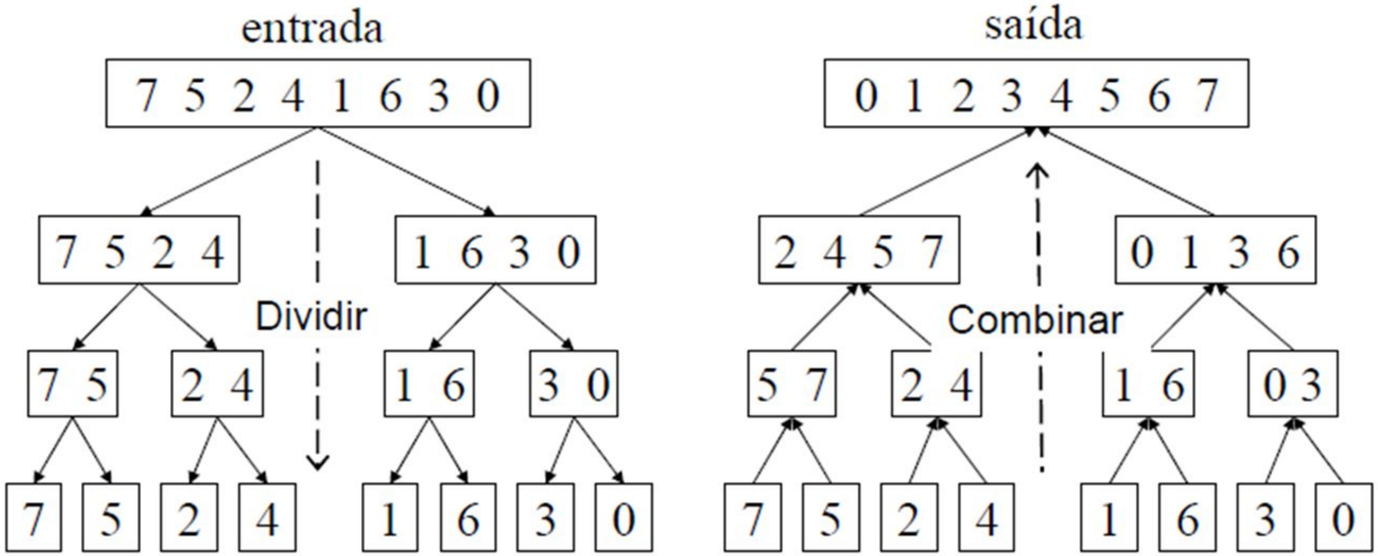
```
proc MergeSort (A [], i, n) {
 se n > 1 então {
   m ← n div 2;
   MergeSort(A, i, m);              /* Dividir e Conq.*/
   MergeSort(A, i + m, n – m);      /* Dividir e Conq.*/
   Merge(A, i, i + m, n);           /* Combinar */
 }
}
```

- A rotina **MergeSort** ordena as posições ***i, i+1, … , i+n–1*** do ***array* A[]**

- A rotina **Merge** será apresentada no próximo slide.

**Note** que:
**i**: primeira posição do *array* a ser ordenado
**n**: número de elementos a ordenar
**div** retorna a parte inteira do quociente obtido na divisão. Já **mod** retorna o resto da divisão inteira. **Ex:** 11 div 4 = 2 e 11 mod 4 = 3

# 4.1. Mergesort




- **Funcionamento** do procedimento **Merge**:
  - Na etapa de **combinar** o procedimento merge é **chamado para cada um dos níveis** da árvore, de baixo para cima.

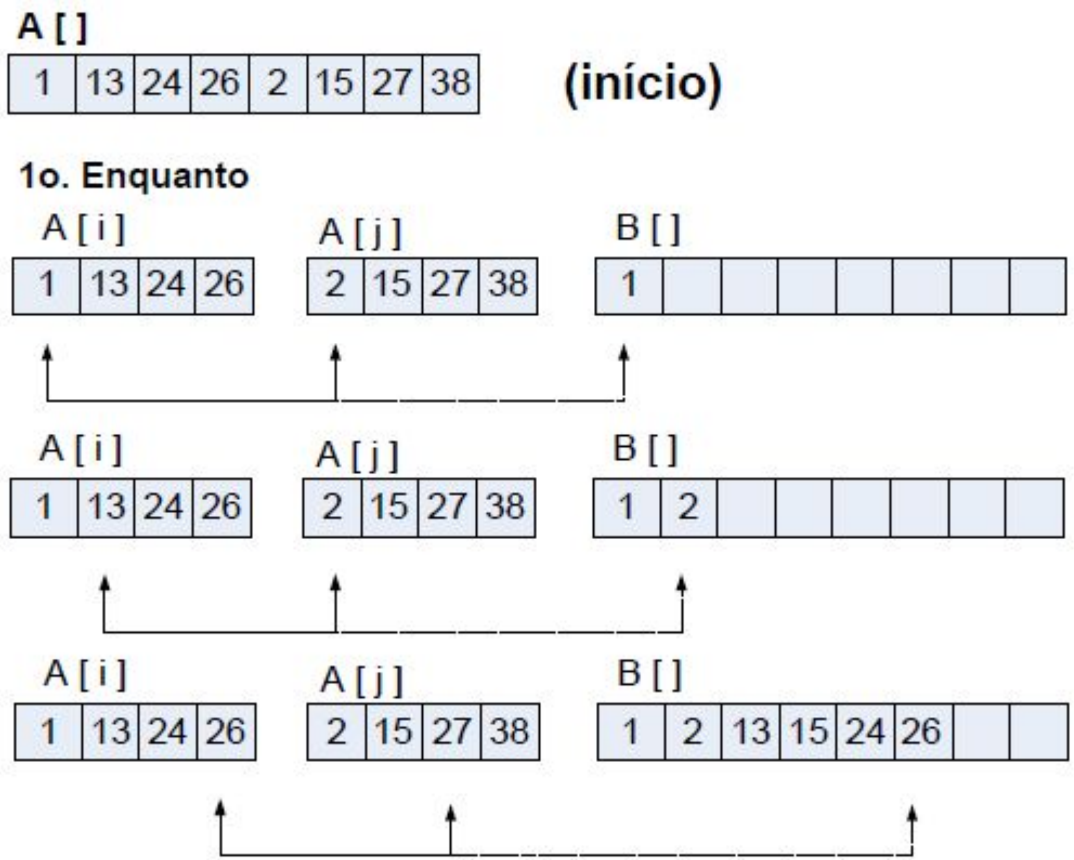
# 4.1. Mergesort

```
proc Merge (A [], i₁, i₂, n) {
   array B [];
   i ← i₁; j ← i₂; k ← i₁;
1  enquanto (i < i₂) e (j < i₁ + n) fazer
{
   se (A[i] ≤ A[j]) então {
      B[k] ← A [i];
      i ← i + 1;
   } senão {
      B[k] ← A[j];
      j ← j + 1;
   }
   k ← k + 1
  }
2  enquanto (i < i₂) fazer {
      B[k] ← A[i];
      i ← i + 1; k ← k + 1;
   }
3  para (i ← i₁) ate (j–1) fazer {
      A[i] ← B[i];
   }
}
```

**Note** que:
- **A** é **dividido em 2** partes.
- **1º enquanto**: **compara** as **2 partes de A**, elemento a elemento, **escolhe o menor elemento** e **copia em B**.
- **2º enquanto**: **copia em B** os **elementos remanescentes** (maiores) de A, se estiverem do lado esquerdo
- **para**: copia de volta em A o que for necessário

# 4.1. Mergesort

Exemplo Numérico - Merge



# 4.1. Mergesort

Exemplo Numérico - Merge

**A [ ]**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 13 | 24 | 26 | 2 | 15 | 27 | 38 |

**(início)**

**1o. Enquanto (após)**

A [ i ]

| | | | |
|---|---|---|---|
| 1 | 13 | 24 | 26 |

A [ j ]

| | | | |
|---|---|---|---|
| 2 | 15 | 27 | 38 |

B [ ]

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 13 | 15 | 24 | 26 | | |

**2o. enquanto**

Nada é copiado para B, pois todos os elementos do lado esquerdo já foram copiados.

**para**

**A [ ]**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 13 | 24 | 26 | 2 | 15 | 27 | 38 |

←

**B [ ]**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 13 | 15 | 24 | 26 | | |

**A [ ]**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 13 | 15 | 24 | 26 | 27 | 38 |

**(fim)**

# 4.1. Mergesort

- **Mergesort** é um algoritmo de ordenação **estável**.
  - **Preserva a ordem** de elementos de **chaves iguais**.
  - Importante, **por exemplo**, se os elementos já estão ordenados segundo alguma **chave secundária**.

- O array auxiliar B não pode ser evitado sem piorar a performance do algoritmo.

- Devido ao ***overhead*** da **recursão** pode-se pensar alternativamente no emprego de algoritmos mais simples
  - **Bubblesort:** quando os ***arrays*** forem **pequenos** (algumas dezenas de elementos).

# 4.1. Mergesort

Alguns algoritmos de ordenação **instáveis**:

- Quicksort
- Heapsort
- **Selection *sort***
- Shellsort

Alguns algoritmos de ordenação **estáveis**:

- **Bubblesort**
- Cocktail sort
- Insertion Sort
- **Mergesort**
- Bucket sort
- Counting sort

- **Alguns algoritmos** são **estáveis** a partir de sua **concepção original**.

- É possível implementar **estabilidade artificialmente** em certos algoritmos.
  - **Requer** um **custo adicional** (aplicar-se uma comparação adicional para verificar se a ordem original dos elementos associados foi mantida).

# 5. Conclusão


*"Sempre faço o que não consigo fazer para aprender o que não sei!”*

*Pablo Picasso*

# Estrutura de Dados

Ordenação (Parte 3 / 3)

# 4.2 Mergesort - Análise

**Análise** da rotina Merge:

proc Merge (A [], $i_1$, $i_2$, n)

- **Cada chamada mescla** (intercala) um total de **n** elementos.
- O **total** de **iterações** dos **2 primeiros laços** (juntos) **e** do **terceiro laço não pode exceder n**:
    - **2 primeiros:** Cada iteração copia exatamente um elemento de A para B **(n)**
    - **3 laço:** Cada iteração copia exatamente um elemento de B para A **(n)**
- Vemos então que, no máximo, **2n iterações** são executadas no total e, **portanto, o algoritmo é O(n)**

```
proc Merge (A [], i1, i2, n) {
   array B [];
   i ← i1; j ← i2; k ← i1;
1  enquanto (i < i2) e (j < i1 + n) fazer {
    se (A[i] ≤ A[j]) então {
       B[k] ← A [i];
       i ← i + 1;
    } senão {
       B[k] ← A[j];
       j ← j + 1;
    }
    k ← k + 1
 }
2 enquanto (i < i2) fazer {
    B[k] ← A[i];
    i ← i + 1; k ← k + 1;
 }
3 para (i ← i1) ate (j–1) fazer {
    A[i] ← B[i];
 }
}
```

# 4.2 Mergesort - Análise

- **Análise** da rotina **MergeSort**: `proc MergeSort(A[],i,n)`

- **Admitamos** que **T(n)** represente a **complexidade** de tempo de **pior caso** (número de passos, comparações, etc) de **MergeSort**.

```
proc MergeSort (A[], i, n) {
 se n > 1 então {
   m ← n div 2;
   MergeSort(A, i, m);
   MergeSort(A, i + m, n – m);
   Merge(A, i, i + m, n);
 }
}
```

- Para **n = 1**, o **tempo** é **constante** - T(1) = 1

- Para **n > 1**, a rotina chama:
  - **A si mesma**, recursivamente em 2 momentos:
    - Cada uma com ≈ n/2 elementos
  - **Merge**, que executa **n operações**
  - **Portanto**, **para n >1**

$$T(n) = T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + T\left(\left\lceil\frac{n}{2}\right\rceil\right) + n$$

# 4.2 Mergesort - Análise

- Vimos que a analise do algoritmo **Mergesort** resultou numa **formula recorrente**:

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

- Para **resolver tais problemas**, podem ser empregadas **muitas técnicas**. Vamos ver **apenas algumas:**
  1. Indução: palpite + verificação por indução
  2. Iteração
  3. Teorema mestre (simplificado)
  4. Árvore de recursão

# 4.2 Mergesort – Análise por Indução

- Vejamos como ***T* (*n*)** se comporta para **alguns valores de *n***

**Note que:**
$T(1) = 1$
$T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n$

$\boldsymbol{T(1)} = 1$
$\boldsymbol{T(2)} = T(1) + T(1) + 2 = 1 + 1 + 2 = 4$
$\boldsymbol{T(3)} = T(2) + T(1) + 3 = 4 + 1 + 3 = 8$
$\boldsymbol{T(4)} = T(2) + T(2) + 4 = 4 + 4 + 4 = 12$
…
$\boldsymbol{T(8)} = T(4) + T(4) + 8 = 12 + 12 + 8 = 32$
…
$\boldsymbol{T(16)} = T(8) + T(8) + 16 = 32 + 32 + 16 = 80$
…
$\boldsymbol{T(32)} = T(16) + T(16) + 32 = 80 + 80 + 32 = 192$

# 4.2 Mergesort – Análise por Indução

- Considerando **valores de n iguais a potência de 2**, pode-se observar um **comportamento** que acredita ser possível identificar um padrão**.**

**Note que:**
$T(1) = 1$
$T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n$

***T*(1)** = 1
***T*(2)** = *T*(1) + *T*(1) + 2 = 1 + 1 + 2 = **4**
***T*(3)** = *T*(2) + *T*(1) + 3 = 4 + 1 + 3 = 8
***T*(4)** = *T*(2) + *T*(2) + 4 = 4 + 4 + 4 = **12**
…
***T*(8)** = *T*(4) + *T*(4) + 8 = 12 + 12 + 8 = **32**
…
***T*(16)** = *T*(8) + *T*(8) + 16 = 32 + 32 + 16 = **80**
…
***T*(32)** = *T*(16) + *T*(16) + 32 = 80 + 80 + 32 = **192**

# 4.2 Mergesort – Análise por Indução

- **De fato,** observamos o seguinte quando consideramos o **valor de** $\frac{T(n)}{n}$**:**
  - $T(1) / 1 = 1$
  - $T(2) / 2 = 2$
  - $T(4) / 4 = 3$
  - $T(8) / 8 = 4$
  - $T(16) / 16 = 5$
  - …
  - $T(2^k) / 2^k = k+1$

- **Ou seja**, para **potências de 2:**
  - $n = 2^k$
  - $\frac{T(n)}{n} = k + 1$
  - $T(n) = n(k + 1)$

$$T(n) = (n\, log_2\, n) + n$$

**Note** que:
$n = 2^k$
então: $k = \log_2 n$

# 4.2 Mergesort – Análise por Indução

- **Temos** então um **palpite** de que para qualquer valor de **n** que seja **potência de 2**.

$$T(n) = (n\, log_2\, n) + n$$

- O que **sabemos** de **fato** é:

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

- **Estamos afirmando** que o **palpite é solução** para a **recorrência**, mas isso precisa ser **provado**.

# 4.2 Mergesort – Análise por Indução

- Primeiro vamos nos **livrar** dos **pisos e tetos** da equação:

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

- **Vamos considerar** como valores de ***n*** apenas **potências de 2**.
- Essa consideração é **possível pois** o **algoritmo não se comporta** significativamente **diferente** quando ***n* não é potência de 2**. Dessa forma **temos**:

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

- Vamos **provar por indução** que, para ***n* = 1 ou** qualquer valor de ***n*** que seja **potência de 2**, temos:

# 4.2 Mergesort – Análise por Indução

- **Palpite** a ser provado: $T(n) = (n\, log_2\, n) + n$

- **Caso base:** n=1
  - $T(1) = (log_2\, 1) + 1$
  - $T(1) = 1$ – **correto** (equivalente ao obtido na expressão original)

- **Hipótese:** o palpite vale para **n/2**, pois **se n é potência de 2, n/2 também é**. Podemos inclusive admitir que é verdade para qualquer potência de 2 **n' < n**
  - $T\left(\frac{n}{2}\right) = \left(\frac{n}{2} log_2 \frac{n}{2}\right) + \frac{n}{2}$

# 4.2 Mergesort – Análise por Indução

- **Caso geral:** substitui-se o resultado da hipótese na **expressão geral** inicial:
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$
  - $T(n) = 2(\left(\frac{n}{2} log_2 \frac{n}{2}\right) + \frac{n}{2}) + n$
  - $T(n) = n\ log_2 \frac{n}{2} + 2n$
  - $T(n) = n(log_2 n\ - log_2 2) + 2n$
  - $T(n) = n(log_2 n\ - 1) + 2n$
  - $T(n) = n(log_2 n)\ - n + 2n$
  - $T(n) = n(log_2 n) +\ n$

- **Conclusão:** o resultado da **expressão geral coincide** com o **palpite**. Está **provado**.

# 4.2 Mergesort – Análise por Iteração

- **Nem sempre**, no entanto, **temos intuição** suficiente sobre o funcionamento do algoritmo para dar um **palpite correto**.
- O método da **iteração** permite que se **reconheça um padrão sem** necessidade de dar um **palpite** (um chute).
- **Quando funciona**, a **solução** do problema da recorrência é obtida resolvendo-se um **somatório.**
- O **método** consiste esquematicamente de:
  - Algumas iterações do caso geral são expandidas até se encontrar uma **lei de formação**.
  - O **somatório resultante** é resolvido substituindo-se os termos recorrentes **por fórmulas** envolvendo apenas o(s) **caso(s) base**.

# 4.2 Mergesort – Análise por Iteração

- Retomando então a **expressão** geral inicial: **T(n) = 2T(n/2) + n**, vamos agora fazer **substituições recorrentemente**.

- No primeiro instante vamos utilizar - **T(n/2) = 2T(n/4) + n/2** e, depois, seguir recorrentemente:

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n \\
&= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n \\
&= 8(2T(n/16) + n/8) + 3n = 16T(n/16) + 4n \\
&= \ldots \\
&= 2^k T(n/(2^k)) + kn
\end{aligned}
$$

# 4.2 Mergesort – Análise por Iteração

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n \\
&= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n \\
&= 8(2T(n/16) + n/8) + 3n = 16T(n/16) + 4n \\
&= \dots \\
&= 2^k T(n/(2^k)) + kn
\end{aligned}
$$

- Lembramos que, **no limite**, temos que chegar no **caso base** da recursão, ou seja, **T(1)**.

- Para termos a **fórmula acima** em termos de **T(1)**, é preciso que **$n/(2^k)$** venha a convergir para **1**, e isso só acontece se **$2^k=n$**, ou seja, **$k=\log_2 n$**

# 4.2 Mergesort – Análise por Iteração

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n \\
&= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n \\
&= 8(2T(n/16) + n/8) + 3n = 16T(n/16) + 4n \\
&= \ldots \\
&= 2^k T(n/(2^k)) + kn
\end{aligned}
$$

- Para **n=1**, temos (lembre que: **k=log$_2$n**):

$$
\begin{aligned}
T(n) &= 2^{\log_2 n} T(n/(2^{\log_2 n})) + (\log_2 n)n \\
&= n^{\log_2 2} T(1) + n \log_2 n \\
&= n + n \log_2 n
\end{aligned}
$$

- Chegamos a mesma fórmula **sem chutar**.

- $\log(ab) = \log a + \log b$
- $\log(a - b) = \log a - \log b$
- $\log a^b = b \log a$
- $\log_a x = \dfrac{\log_b x}{\log_b a}$
- $a^{\log_b x} = x^{\log_b a}$

# 4.2 Mergesort – Análise por Árvore de Recursão

- **Árvore de recursão:**
  - **Maneira gráfica** de **visualizar** a estrutura de **chamadas recursivas** do algoritmo.

- **Cada nó** da árvore é uma **instância** (chamada recursiva).

- **Cada nó** é **rotulado** com o **tempo gasto** apenas nas **operações locais.**
  - **Sem contar** as chamadas **recursivas**.

**Mergesort com** chamadas recursivas

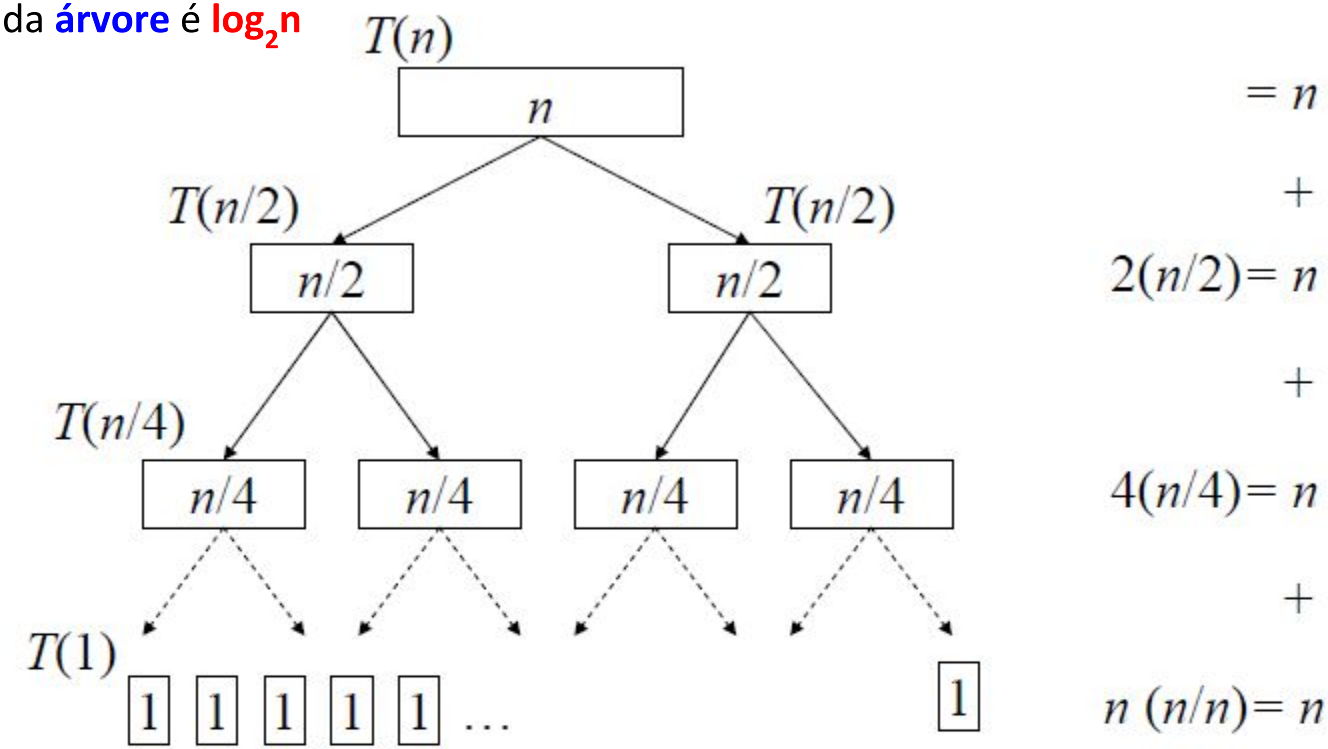
$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ 2T(n/2) + n & \text{se } n > 1. \end{cases}$$

**Mergesort sem** chamadas recursivas

$$T(n) = \begin{cases} 1 & \text{se } n = 1, \\ n & \text{se } n > 1. \end{cases}$$

# 4.2 Mergesort – Análise por Árvore de Recursão

**Altura** da **árvore** é **$\log_2 n$**




***$T(n) = n \log_2 n$***

# Mergesort – Análise pelo Teorema Mestre

- Podemos observar que as **fórmulas de recorrência** provenientes de algoritmos do tipo **Dividir para Conquistar** são muito **semelhantes.**

- Tais algoritmos tendem a **dividir o problema** em **a partes iguais**, cada uma de tamanho **b vezes menor** que o **problema original.**

- Quando, **além disso**, o trabalho executado em **cada instância** da **recursão** é uma **potência de n**, **existe um teorema** que nos dá diretamente a complexidade assintótica do algoritmo.

- Em outras palavras, o **Teorema Mestre (simplificado)** pode resolver recorrências cujo caso geral é da forma:

$$T(n) = aT\,(n/b) + n^k$$

# Mergesort – Análise pelo Teorema Mestre

- Teorema Mestre (simplificado)

- Dadas as constantes **a ≥ 1** e **b ≥ 1** e uma recorrência da forma :

$$T(n) = aT(n/b) + n^k$$

- Então:
  - **Caso 1:** se $a > b^k$ então: $T(n) \in \Theta(n \log_b a)$
  - **Caso 2:** se $a = b^k$ *então:* $T(n) \in \Theta(n^k \log_2 n)$
  - **Caso 3:** se $a < b^k$ então: $T(n) \in \Theta(n^k)$

**Lembre** que:

$$a^p . a^q = a^{p+q}$$

$$a^p . a^{-q} = a^{p-q} = \frac{a^p}{a^q}$$

- Assumimos que **n** é uma **potência de b** e que o **caso base T(1)** tem **complexidade constante**.

# Mergesort – Análise pelo Teorema Mestre

- Teorema Mestre (simplificado)

- **MergeSort - $T(n) = 2T(n/2) + n^1$**
  - **a=2, b=2, k=1**
  - **Caso 2** se aplica, então: $T(n) \in \Theta(n \log_2 n)$

- Outros exemplos:
- **$T(n) = 3T(n/2) + n^2$**
  - **a=3, b=2, k=2**
  - **Caso 3** se aplica, então: $T(n) \in \Theta(n^2)$

- **$T(n) = 2T(n/2) + n \log_2 n$**
  - Teorema mestre **não se aplica**
  - Pode ser resolvido **por iteração**

# 4.3 Quicksort

- Quicksort é um **algoritmo** de **dividir e conquistar** que **divide um *array*** em duas partes (2 *subarrays*).
  - Os elementos "menores" e os elementos "maiores" – e então **recursivamente ordena** os 2 ***subarrays***.

- As **etapas** são as seguintes:
  - **Escolher** um elemento do *array* chamado **pivô**.
  - **Reordenar** o ***array*** para que todos os **elementos** com valores **menores** que o **pivô** esteja **antes** do pivô e os **elementos maiores**, **após** o pivô.
  - Após esta separação, o **pivô** está na sua **posição final** e **haverá duas listas não ordenadas**.
  - **Recursivamente ordenar** o ***array*** com os **valores menores e** o ***array*** com os **valores maiores**.

# 4.3 Quicksort

- O *QuickSort* é **provavelmente** o **algoritmo mais usado** na prática para **ordenar** *arrays*.
  - Sua complexidade de **caso médio** é $\boldsymbol{\Theta(n \log_2 n)}$
  - Sua complexidade de **pior caso** é $\mathbf{O(n^2)}$
  - A **probabilidade** do **pior caso** acontecer fica **menor à medida** que **n cresce.**

- O **passo crucial** do algoritmo é **escolher** um elemento do *array* para servir de **pivô**.

# 4.3 Quicksort

- O ***QuickSort*** pode se tornar um algoritmo de complexidade de **pior caso** $\mathbf{O(n \log_2 n)}$ **se** escolhermos sempre a **mediana** como **pivô.**

  - **Algoritmo** que seleciona a **mediana** dos elementos: **Θ(n)**.

  - **Entretanto**, o algoritmo (Quicksort) acaba tendo um **desempenho ruim** na **prática**.

  - Quando **n cresce**, a adição de tempo para **escolher** a **mediana** torna-se **intolerável**.

# 4.3 Quicksort

- **Seja** uma **amostra** (uma população) **ordenada** de tamanho n, a **mediana** é **um número** (uma medida) **que separa a metade inferior** (1/2 da amostra terá valores inferiores) **da amostra da metade superior** (1/2 da amostra terá valores superiores ou iguais à mediana).

- **Como calcular** a **mediana?**

- Considerando uma amostra ordenada de tamanho n, se n for **ímpar**, a mediana será o **elemento central**.

    - Se n for **par**, a mediana será o resultado da **média simples** entre os elementos **n/2 e (n/2 +1)**.

# 4.3 Quicksort

- **Exemplos de medianas:**
  - 1, 3, 5, 7, 9 A **mediana** é 5 (igual à média)
  - 1, 2, 4, 10, 13 A **mediana** é 4 (enquanto a média é 6)
  - 1, 2, 4, 7, 9, 10 Amostra de tamanho par. A **mediana** é (4+7)/2, que é 5.5 (enquanto a média é 6.6).

- O ***QuickSort*** utiliza **como base** um **algoritmo** de **particionamento**.
  - Particionar um *array* em torno de um **pivô x** significa **dividi-lo** em **3 segmentos** contíguos contendo, respectivamente, todos os **elementos** **< x**, **= x**, e **> x**.

# 4.3 Quicksort

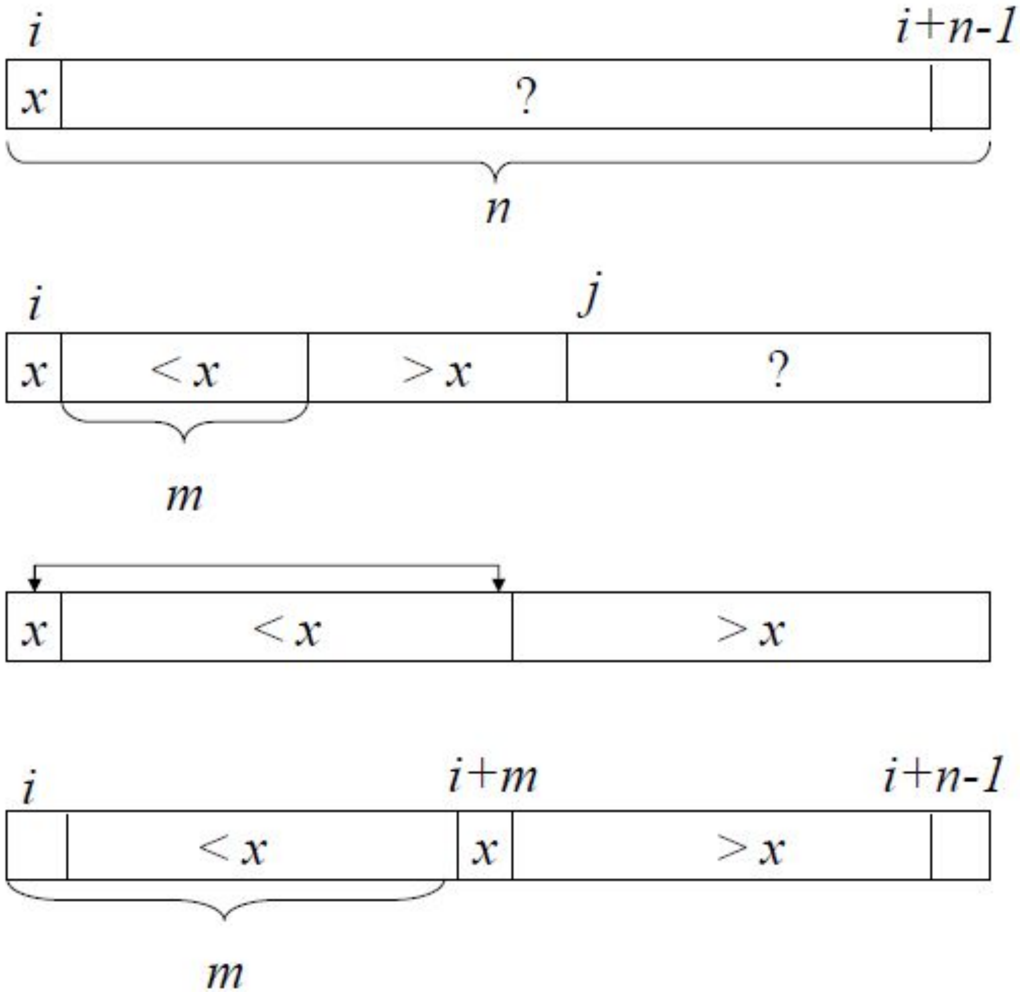
- **Vamos definir** uma **função partição** que **divide** um ***array*** **A** de n elementos indexados **a partir** do índice **i**, isto é:
  - **A[i], A[i+1] … A[i + n – 1]**
  - Como **pivô**, escolheremos **x = valor inicial** do vetor = **A[0]**.
  - A função retorna **m**, o número de **elementos menores que *x***
  - **Assumimos** também que o *array* tem **todos** os seus **elementos distintos**, logo, ao **final da chamada**, os segmentos são:

A[i] … A[i + m – 1] (elementos menores que **x**)

A[i + m] (elemento igual a **x = pivô**)

A[i + m + 1] … A[i + n – 1] (elementos maiores que **x**)

# 4.3 Quicksort - Partição




**Ideia:**
Tem-se: $A[i..i+n-1]$
**1** – pivô = $A[i]$
**2** – $m = 0$
**3** - $j = i+1$
**4** - $A[j] < A[i]$ ?
**5** - Sim
$m = m+1$
trocar $A[j]$ com $A[i+m]$
**7** - Não: faz nada.
**8** - Incrementar $j$ e repetir processo (**4**) ate $j = i+n-1$.
**9** - trocar $A[i]$ com $A[i+m]$
**10** - pivô = $A[i+m]$

# 4.3 Quicksort - Partição

**Inicio**

| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| 4 | 7 | 3 | 2 | 1 | 9 | 10 |

m = 0; i = 5; n = 7; i+n-1 = 11

**Exemplo:**

$j \leftarrow i + 1$

A[j] = A[6] = 7 < A[i] = A[5] = 4 : Não

**Exemplo:**

$j \leftarrow i + 2 = 7$

A[j] = A[7] = 3 < A[i] = A[5] = 4 : Sim

$m \leftarrow m+1 = 1$

Trocar A[7] com A[i+m] = A[6]

| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| 4 | 3 | 7 | 2 | 1 | 9 | 10 |

m = 1; i = 5; n = 7; i+n-1 = 11

**Ao final:**

| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| 1 | 3 | 2 | 4 | 7 | 9 | 10 |

m = 3; i = 5; n = 7; i+n-1 = 11

**Ideia:**

Tem-se: $A[i..i+n-1]$

**1** – pivô (x) = $A[i]$

**2** – $m = 0$

**3** - $j = i+1$

**4** - $A[j] < A[i]$ ?

**5** - Sim

$m = m+1$

trocar $A[j]$ com $A[i+m]$

**7** - Não: faz nada.

**8** - Incrementar $j$ e repetir processo (**4**) ate $j = i+n-1$.

**9** - trocar $A[i]$ com $A[i+m]$

**10** - pivô = $A[i+m]$

# 4.3 Quicksort - Partição

```
proc partição(i, n, A[i..i+n–1]) {
  m ← 0
  para j ← i+1 ate (i + n – 1) fazer {
   se (A [j] < A [i]) entao {
     m ← m + 1
     trocar A[j] com A[i+m];
   }
  }
  trocar A[i] com A[i+m];
  retornar m
}
```

**Note que:**
**i**: índice do pivô.
**n**: número de elementos do array.
**A[ i..i+n-1]**: array.

# 4.3 Quicksort

- É fácil ver que o algoritmo **partição** tem **complexidade** de pior caso **$\Theta(n)$**.

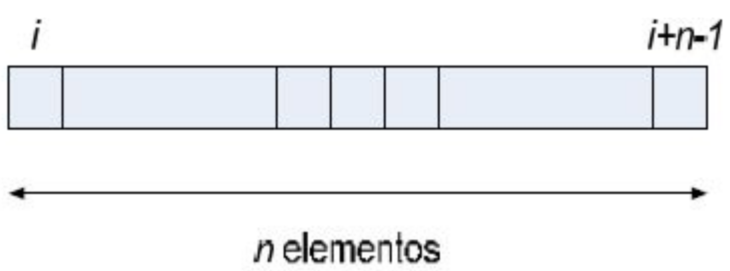
- Se quisermos escolher **outro** elemento como **pivô**, o de **índice k** digamos, **basta trocar A[i] com A[k]** antes de **chamar** o algoritmo **partição**.

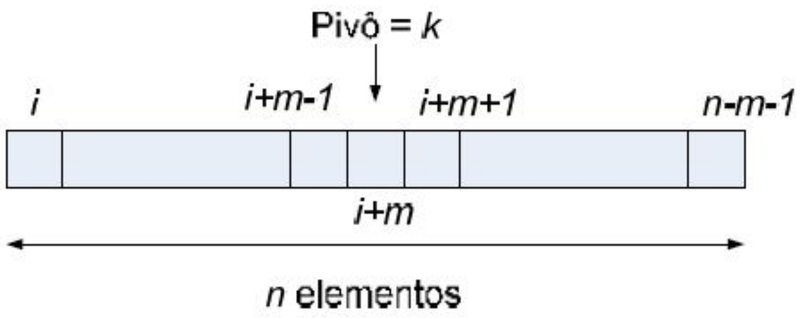
- **Na prática**, utiliza-se um **pivô randômico**.
  - Prova-se que isso garante complexidade de **caso médio $\Theta(n \log n)$** para o Quicksort.
  - Já a complexidade de **pior caso**, como já mencionamos, é **$O(n^2)$**.

- O **Quicksort** funciona **particionando** o array **recursivamente até** que todos os segmentos tenham **tamanho $\leq 1$**.

# 4.3 Quicksort - Algoritmo

```
proc QuickSort (i, n, A [i .. i + n – 1]) {
  se (n > 1) entao {
    escolha um valor (randômico) k entre i e i + n – 1
    trocar A[k] com A[i]
    m ← partição (i, n, A)
    QuickSort (i, m, A)
    QuickSort (i+m+1, n–m–1, A)
  }
}
```



# 4.3 Quicksort - Análise

- A **análise** de **pior caso** do algoritmo **resulta** na **recorrência:**

$$T(n) = \begin{cases} 1 & \text{se } n \leq 1, \\ T(m) + T(n-m-1) + n & \text{se } n > 1. \end{cases}$$

- Para o **pior caso**, **a pergunta é**:
  - Qual **valor** de **m maximiza T(n)**?

- Nesse caso, **a resposta é**:
  - Valores de **m** iguais a **0 ou n – 1** fazem **T(n) = O($n^2$)**

# 4.3 Quicksort - Análise

- Se **m = 0**, a recorrência se reduz a:

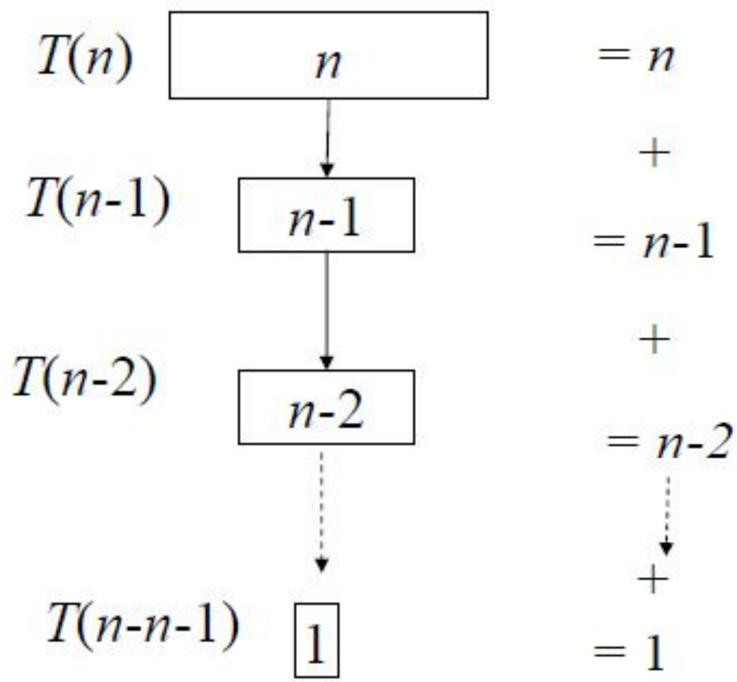
$$T(n) = \begin{cases} 1 & \text{se } n \leq 1, \\ T(0) + T(n-1) + n & \text{se } n > 1. \end{cases}$$

- Como trata-se de **pior caso**, vamos desprezar o valor constante **T(0) = 1**.

- Vamos agora utilizar a **árvore de recursão** para fazer essa **análise**. Podemos então reescrever a recorrência como:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 1, \\ n & \text{se } n > 1. \end{cases}$$

# 4.3 Quicksort - Análise




**Simplificadamente:** cada nível da árvore tem um trabalho **O(n)**, considerando que há **n níveis**, então a complexidade total é **$T(n) = O(n^2)$**.

# 4.3 Quicksort - Análise

- Se **m = n-1**, a **recorrência** se reduz a:

$$T(n) = \begin{cases} 1 & \text{se } n \le 1, \\ 2T(n-1) + n & \text{se } n > 1. \end{cases}$$

- Vamos agora utilizar a **árvore de recursão** para fazer essa **análise**. Podemos então reescrever a **recorrência** como:

$$T(n) = \begin{cases} 1 & \text{se } n \le 1, \\ n & \text{se } n > 1. \end{cases}$$

- Note que a **análise** torna-se **idêntica** a que acabamos de realizar considerando **m = 0**. Portanto:

**$T(n) = O(n^2)$**

# 4.3 Quicksort - Análise

- Se **m = n/2**, a **recorrência** torna-se:

$$T(n) = \begin{cases} 1 & \text{se } n \le 1, \\ T(n/2) + T(n/2) + n & \text{se } n > 1. \end{cases}$$

- Veja que é **igual** a recorrência do **Mergesort** . Portanto a complexidade de **pior caso** passa a ser:

**T(n) = O(n log n)**

# 4.3 Quicksort - Análise

- **Conclusão:** para ter essa **complexidade de pior caso** é **preciso** escolher um **pivô** que faça **m = n/2**, isto quer dizer, que o pivô deve ser a **própria mediana do array**.
  - Note que: o algoritmo a ser utilizado para **determinar** a **mediana deve ser** no **máximo O(n)**, pois é a complexidade de tempo do algoritmo partição, o qual estabelece o trabalho efetuado em cada nível da árvore de recursão.

- **Na prática**, contudo, como já mencionado, utiliza-se um **pivô randômico**. Isso por que o tempo adicional (na prática) para **identificar a mediana** torna-se **intolerável** conforme **n cresce**.

# 5. Conclusão

*"Que ninguém se engane, só se consegue a simplicidade através de muito trabalho."*

*Clarice Lispector*