

# AED2 - Aula 10

## Ordenação por intercalação (mergesort)

### O problema da ordenação

Definição de ordenação (crescente)

- Um vetor  $v[0 .. n - 1]$  está ordenado se  $v[0] \leq v[1] \leq v[2] \leq \dots \leq v[n-1]$

Definindo o problema da ordenação:

- Dado um vetor  $v$  de tamanho  $n$ ,
  - permutar os elementos de  $v$  até ele ficar ordenado.

Exemplo:

- Entrada


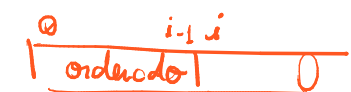

<del>77</del>	<del>55</del>	<del>11</del>	<del>44</del>	<del>33</del>	<del>22</del>	<del>88</del>	<del>66</del>
---------------	---------------	---------------	---------------	---------------	---------------	---------------	---------------

- Saída

11	22	33	44	55	66	77	88
----	----	----	----	----	----	----	----

### Relembrando

Vimos algoritmos iterativos simples com tempo  $O(n^2)$  no pior caso:

- insertionSort, 
- selectionSort, 
- bubbleSort. 

Também vimos o heapSort, que usa a estrutura de dados Heap

- para melhorar o selectionSort e atingir tempo  $O(n \log n)$ .

### Na aula de hoje

Veremos três tópicos extremamente importantes:

- Técnica de projeto de algoritmos divisão-e-conquista.
  - Permite obter algoritmos nada óbvios e muito eficientes.
- Algoritmo de ordenação mergeSort,
  - clássico e ainda relevante.
- Árvore de recursão para analisar eficiência de algoritmo recursivo.
  - Pode ser generalizada no Teorema Mestre.

## Projeto de algoritmos por divisão e conquista

- **Dividir**: o problema é dividido em subproblemas menores do mesmo tipo.
- **Conquistar**: os subproblemas são resolvidos recursivamente, sendo que os subproblemas pequenos são caso base.
- **Combinar**: as soluções dos subproblemas são combinadas numa solução do problema original.

## Ideia do mergesort

- **Dividir** o vetor a ser ordenado em dois subvetores,
  - cada um com metade do tamanho original.
- **Ordenar** cada subvetor recursivamente,
  - sendo que subvetores com 0 ou 1 elementos já estão ordenados.
- Intercalar (**merge**) os subvetores ordenados resultantes.

Exemplo:

77	55	11	44	33	22	88	66
----	----	----	----	----	----	----	----

- Dividir em dois subvetores.



- Conquistar recursivamente (lembrar dos casos base).



- Combinar por intercalação (merge).



## Algoritmo mergeSort recursivo

// ordena o vetor  $v[p \dots r-1]$

```
void mergeSortR(int v[], int p, int r) {
```

```
    int m;
```

```
    if (r - p > 1) { // se subvetor corrente tem mais de 1 elemento
```

```
        m = (p + r) / 2; //  $m = p + (r - p) / 2$ 
```

```
        mergeSortR(v, p, m);
```

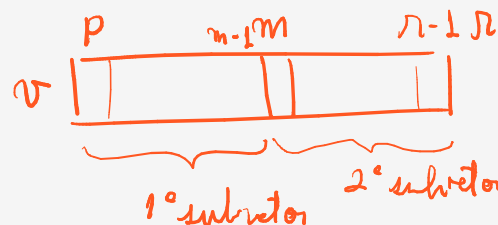
```
        mergeSortR(v, m, r);
```

```
        intercala1(v, p, m, r);
```

```
    }
```

```
}
```

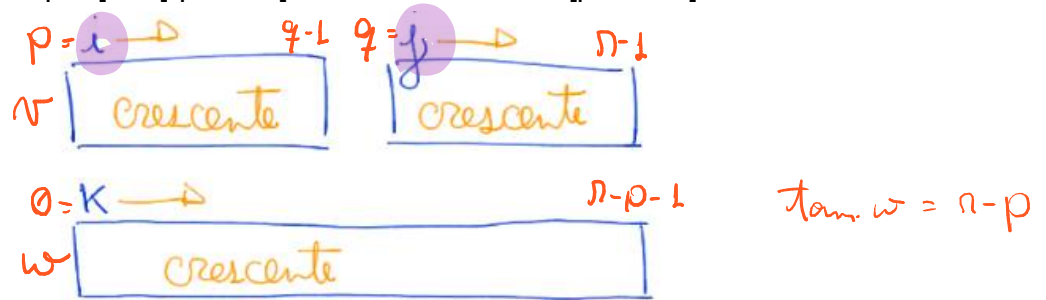
$$m = \frac{p+r}{2} = \left(\frac{p-p}{2}\right) + \frac{r}{2} = p + \frac{(r-p)}{2}$$



- A questão central é
  - como fazer a intercalação (merge) com eficiência linear?
- Note que, se intercala levar tempo  $O(n^2)$ , não há esperança
  - desse algoritmo ser mais eficiente que os algoritmos elementares.

## Problema da intercalação

- Dados  $v[p \dots q - 1]$  e  $v[q \dots r - 1]$  ordenados, obter  $v[p \dots r - 1]$  ordenado.



Ideia: percorrer cada subvetor, colocando no vetor auxiliar

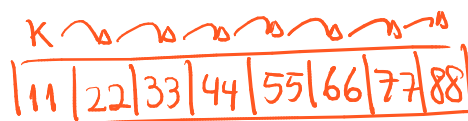
- o menor dentre os elementos correntes de cada subvetor.

Exemplo:

- Vetor  $v$



- Vetor  $w$



Algoritmo de intercalação de subvetores ordenados

```
// primeiro subvetor em  $v[p \dots q-1]$ , segundo subvetor em  $v[q \dots r-1]$ 
void intercala1(int v[], int p, int q, int r) {
    int i, j, k, tam;
    i = p; j = q; k = 0; tam = r - p;
    int *w = malloc(tam * sizeof(int));
    while (i < q && j < r) { // não chegou a fim de nenhum subvetor
        if (v[i] <= v[j]) w[k++] = v[i++]; // cópia do 1º subvetor
        else /* v[i] > v[j] */ w[k++] = v[j++]; // cópia do 2º subvetor
    }
    while (i < q) w[k++] = v[i++];
    while (j < r) w[k++] = v[j++];
    for (k = 0; k < tam; k++) {
        v[p + k] = w[k];
    }
    free(w);
}
```

Invariantes e correteude do intercala1: no início de cada iteração temos

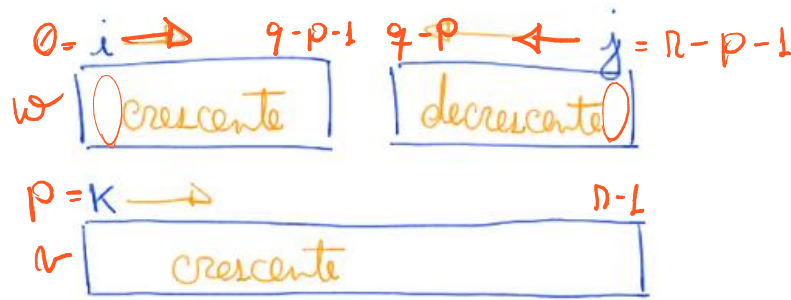
- $w[0 \dots k - 1]$  contém os elementos de  $v[p \dots i - 1]$  e  $v[q \dots j - 1]$ ,
- $w[0 \dots k - 1]$  está ordenado,
- $w[0 \dots k - 1] \leq v[i \dots q - 1]$ ,
- $w[0 \dots k - 1] \leq v[j \dots r - 1]$ .

Eficiência de tempo do intercala:

- O número de operações é linear no tamanho do subvetor sendo intercalado,
  - ou seja,  $O(r - p) = O(\text{tam})$
- Para verificar isso, note que em cada iteração, de qualquer laço,
  - $i$  ou  $j$  aumenta de 1.
- Como  $i$  começa em  $p$  e termina em  $q$  e  $j$  começa em  $q$  e termina em  $r$ ,
  - temos  $(q - p) + (r - q) = q - p + r - q = r - p = \text{tam}$

Curiosidade:

- Sedgewick propõe uma versão interessante do algoritmo de intercalação,
  - chamado intercalação com sentinelas.



```
// primeiro subvetor em v[p .. q-1], segundo subvetor em v[q .. r-1]
```

```
void intercala2(int v[], int p, int q, int r) {
```

```
    int i, j, k, *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 - 1) - (j - q)] = v[j];
```

```
    i = 0;
```

```
    j = r - p - 1;
```

```
    for (k = p; k < r; ++k)
```

```
        if (w[i] <= w[j])
```

```
            v[k] = w[i++];
```

```
        else
```

```
            v[k] = w[j--];
```

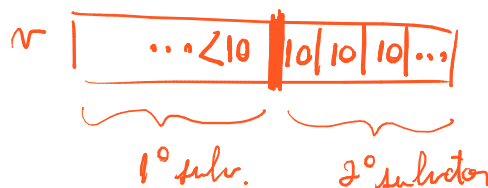
```
    free(w);
```

```
}
```

cópia do 1º subvetor  
cópia do 2º subvetor invertido

cópia de volta p/ r em ordem

- Você consegue entender por que a função anterior funciona?
  - Por que ela não precisa de laços para copiar as sobras do primeiro ou segundo subvetor?
- Quais os invariantes do seu laço principal?
- Ela é estável? I.e., elementos iguais tem sua ordem relativa preservada?
  - Considere o caso em que existe repetição do maior elemento
    - e essas repetições estão no final do 2º subvetor.



Relembrando o código do mergeSort recursivo:

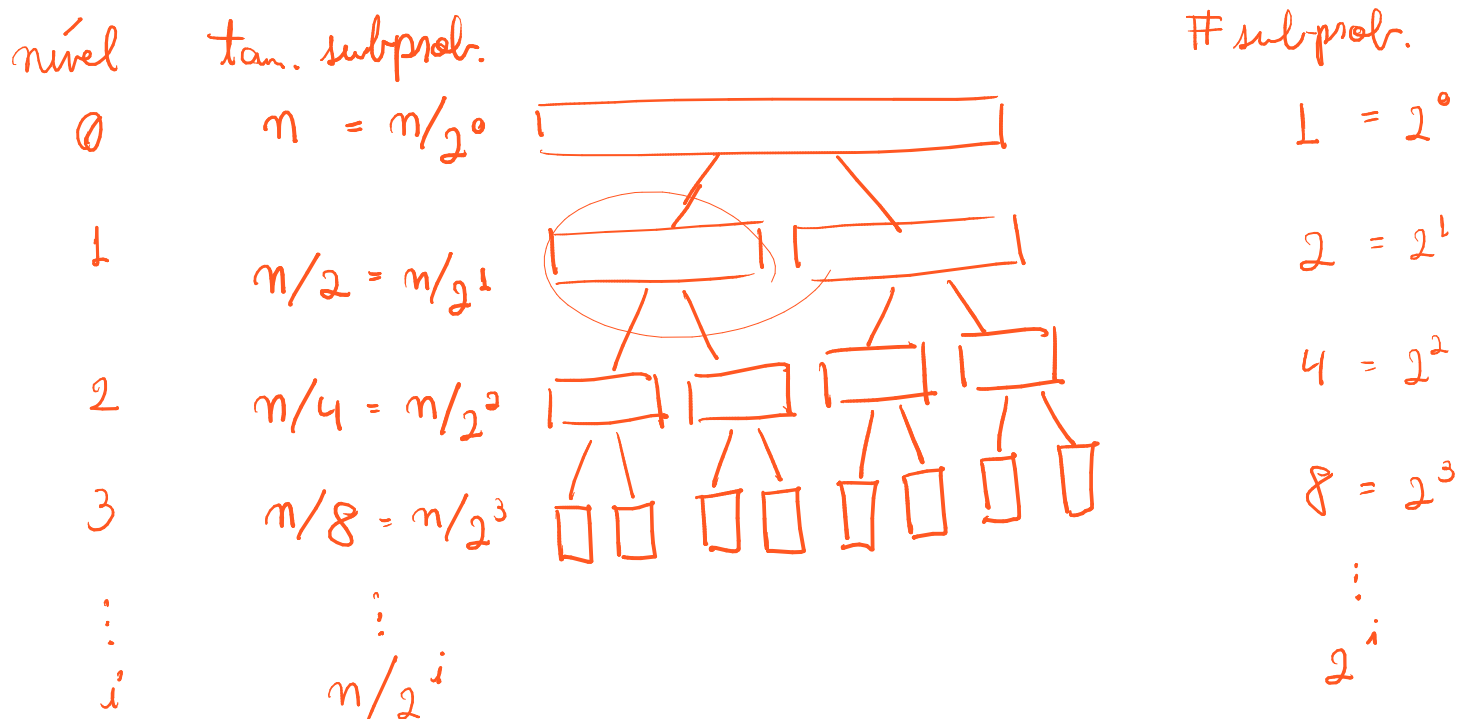
```
// ordena o vetor v[p .. r-1]
void mergeSortR(int v[], int p, int r) {
    int m;
    if (r - p > 1) {
        m = (p + r) / 2; // m = p + (r - p) / 2;
        mergeSortR(v, p, m); -1°
        mergeSortR(v, m, r); -2°
        intercala1(v, p, m, r);
    }
}
```

Corretude do mergeSort (por indução matemática):

- Pelo caso base  $p \geq r - 1$  sabemos que nosso algoritmo devolve
  - subvetores ordenados quando estes tem tamanho menor ou igual a 1.
- Supondo que nosso algoritmo ordena um subvetor de tamanho  $n/2$ ,
  - verificamos que ele ordena um vetor de tamanho  $n$ ,
    - uma vez que a função intercala funciona corretamente.
- Note que é necessário provar, usando invariantes, a corretude dessa função.

Eficiência de tempo do mergeSort:

- Usamos uma árvore (binária) de recursão na análise.



$$\frac{n}{2^h} = 1 \Rightarrow 2^h = n \Rightarrow h = \lg n = \log_2 n$$

tam. subprob. nível i =  $\frac{n}{2^i}$       # subprob. nível i =  $2^i$       # níveis =  $1 + \lg n$

tam. subprob. nível  $i = \frac{n}{2^i}$     # subprob. nível  $i = 2^i$     # níveis =  $1 + \lg n$

- Curiosidade: outra maneira de chegar ao # de níveis igual a  $\log_2 n + 1$ , é:
  - no nível 0 temos  $n$  elementos no vetor,
  - $\log_2 n$  é o número de vezes que podemos dividir  $n$  por 2
    - antes dele se tornar menor ou igual a 1 (caso base).

Questão chave: qual o trabalho (não recursivo) realizado por nível da árvore?

- Uma chamada do mergeSort realiza basicamente um teste,
  - seguido de duas chamadas recursivas e uma chamada de intercala.
- Como intercala é uma função com eficiência linear, i.e.,  $O(\text{tam. subvetor sendo intercalado})$ 
  - o trabalho não recursivo realizado por mergeSort
    - num vetor de tamanho  $m$  é  $c * m$ , para alguma constante  $c$ .
- Assim, o trabalho realizado por problema do nível  $i$  é  $c * \frac{n}{2^i}$
- Portanto, o trabalho realizado por nível da árvore é dado
  - pelo número de subproblemas por nível vezes
    - o trabalho não recursivo realizado por subproblema, i.e.,

$$\text{Trab}(i) = \# \text{subprob}(i) \cdot c \cdot \frac{n}{2^i} = 2^i \cdot c \cdot \frac{n}{2^i} = c n = O(n)$$

- Por fim, o trabalho total é dado pela soma no número de níveis da árvore
  - do trabalho realizado por nível desta,

$$\left[ \begin{aligned} \text{i.e., } \sum_{j=0}^{\log_2 n} c * n &= c * n \sum_{j=0}^{\log_2 n} 1 \\ &= c * n * (1 + \log_2 n) = cn \log_2 n + cn = O(n \log n) \end{aligned} \right]$$

$$\text{Trab. Total} = \sum_{j=0}^{\log_2 n} c * n = c * n (1 + \lg n) = O(n \cdot \lg n)$$

- Numa comparação rápida, para  $n = 10^6$  e  $10^9$  temos:

$n$	$10^6$	$10^9$
$\lg n$	20	30
$n \lg n$	$2 \cdot 10^7$	$3 \cdot 10^{10}$
$n^2$	$10^{12}$	$10^{18}$

- Supondo um computador de 10 GHz (i.e.,  $10^{10}$  operações por segundo),
  - quanto tempo um algoritmo de ordenação  $O(n \log n)$  leva para
    - ordenar vetores de  $10^6$  e  $10^9$  elementos, respectivamente?
- Responda às questões anteriores para um algoritmo de ordenação  $O(n^2)$ .

Estabilidade:

- Ordenação do mergeSort é estável, desde que a intercalação seja.
  - Por que? Mostre que isso vale usando indução.

Eficiência de espaço:

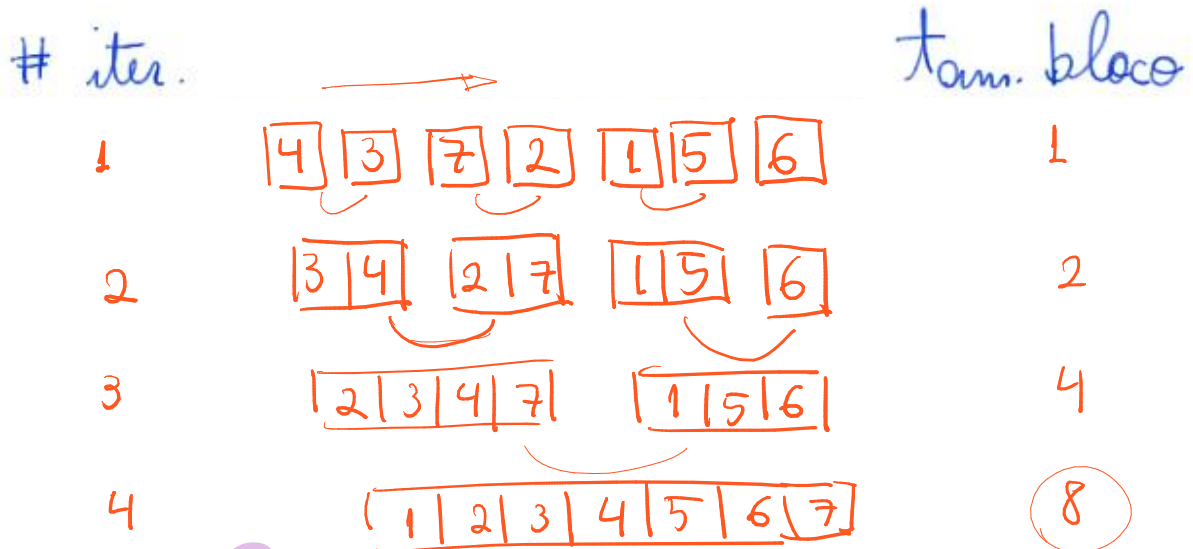
- Ordenação não é **in place**, pois usa a rotina intercala
  - que precisa de vetor auxiliar (e portanto memória)
    - proporcional ao tamanho dos vetores sendo intercalados.

Curiosidade:

- Podemos usar o algoritmo insertionSort como caso base do mergeSort.
- Isso é interessante porque o insertionSort tem constante menor que o mergeSort, sendo por isso mais rápido quando  $n$  é pequeno.

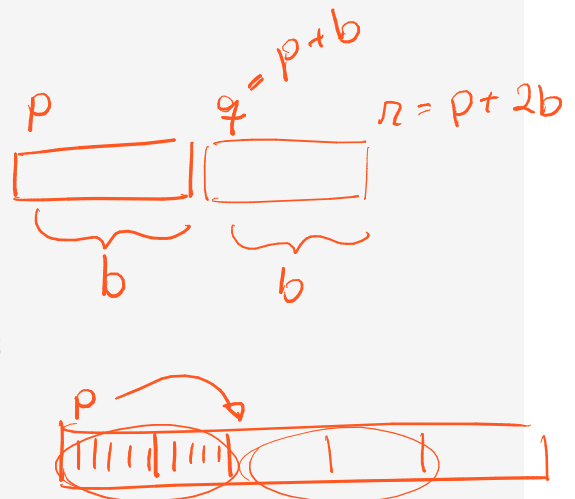
Bônus: algoritmo mergeSort iterativo,

- que em cada iteração percorre o vetor
  - intercalando pares de blocos de tamanho  $b$ .



- Note que,  $b$  dobra de tamanho a cada iteração.

```
void mergeSortI(int v[], int n) {
    int b = 1;
    while (b < n) {
        int p = 0;
        while (p + b < n) {
            int r = p + 2 * b;
            if (r > n) r = n;
            intercala1(v, p, p + b, r);
            p = p + 2 * b;
        }
        b = 2 * b;
    }
}
```



Animação:

- Visualization and Comparison of Sorting Algorithms - [www.youtube.com/watch?v=ZZuD6iUe3Pc](http://www.youtube.com/watch?v=ZZuD6iUe3Pc)