

# MC-202

## Ordenação

Lehilton Pedrosa  
lehilton@ic.unicamp.br

Universidade Estadual de Campinas

Segundo semestre de 2024

# Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
  - O valor usado para a ordenação é a *chave* de ordenação
  - Podemos até desempatar por outros campos

# BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {
2     int i, j;
3     for (i = 0; i < n - 1; i++)
4         for (j = n - 1; j > i; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j-1], &v[j]);
7 }
```

## Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int n) {
2     int i, j, trocou = 1;
3     for (i = 0; i < n - 1 && trocou; i++){
4         trocou = 0;
5         for (j = n - 1; j > i; j--){
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

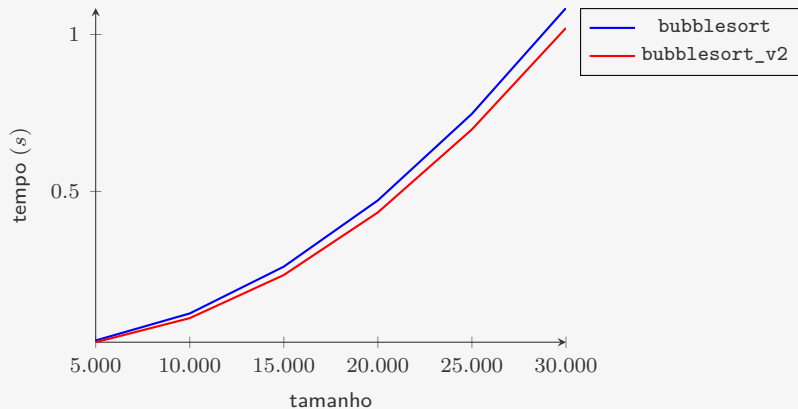
No pior caso toda comparação gera uma troca:

- comparações:  $n(n-1)/2 = O(n^2)$
- trocas:  $n(n-1)/2 = O(n^2)$

No caso médio:

- comparações:  $\approx n^2/2 = O(n^2)$
- trocas:  $\approx n^2/2 = O(n^2)$

## Gráfico de comparação do BubbleSort



A segunda versão é um pouco mais rápida

# Ordenação por Inserção

Ideia:

- Se já temos  $v[0], v[1], \dots, v[i-1]$  ordenado
- Inserimos  $v[i]$  na posição correta
  - fazemos algo similar ao BubbleSort
- Ficamos com  $v[0], v[1], \dots, v[i]$  ordenado

```
1 void insertionsort(int *v, int n) {
2     int i, j;
3     for (i = 1; i < n; i++)
4         for (j = i; j > 0; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j], &v[j-1]);
7 }
```

## Ordenação por Inserção — Otimizações

Quando o elemento já está na sua posição correta não é necessário mais percorrer o vetor testando se  $v[j] < v[j-1]$

Se trocamos  $v[j]$  com  $v[j-1]$  e  $v[j-1]$  com  $v[j-2]$

- fazemos 3 atribuições para cada troca = 6 atribuições
- é melhor fazer:

```
t = v[j]; v[j] = v[j-1]; v[j-1] = v[j-2]; v[j-2] = t;
```

```
1 void insertionsort_v2(int *v, int n) {
2     int i, j, t;
3     for (i = 1; i < n; i++) {
4         t = v[i];
5         for (j = i; j > 0 && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

# Ordenação por Inserção — Análise

```
1 void insertionsort_v2(int *v, int n) {
2     int i, j, t;
3     for (i = 1; i < n; i++) {
4         t = v[i];
5         for (j = i; j > 0 && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

No caso pior caso:

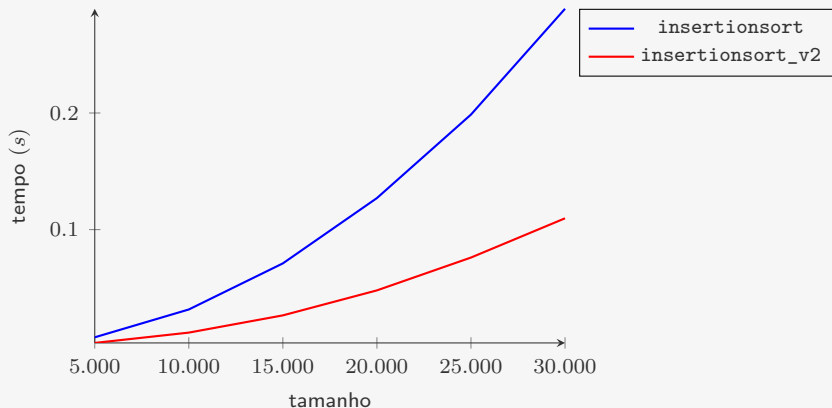
- comparações:  $\approx n^2/2 = O(n^2)$
- atribuições (ao invés de trocas):  $\approx n^2/2 = O(n^2)$

No caso médio é metade disso:

- cada elemento anda metade do prefixo do vetor em média



## Gráfico de comparação do InsertionSort



A complexidade teórica do algoritmo não melhorou

- continua  $O(n^2)$

Mas as otimizações levaram a um ganho na performance

- menos do que a metade do tempo para  $n$  grande

# Ordenação por Seleção

Ideia:

- Trocar  $v[0]$  com o mínimo de  $v[0], v[1], \dots, v[n-1]$
- Trocar  $v[1]$  com o mínimo de  $v[1], v[2], \dots, v[n-1]$
- ...
- Trocar  $v[i]$  com o mínimo de  $v[i], v[i+1], \dots, v[n-1]$

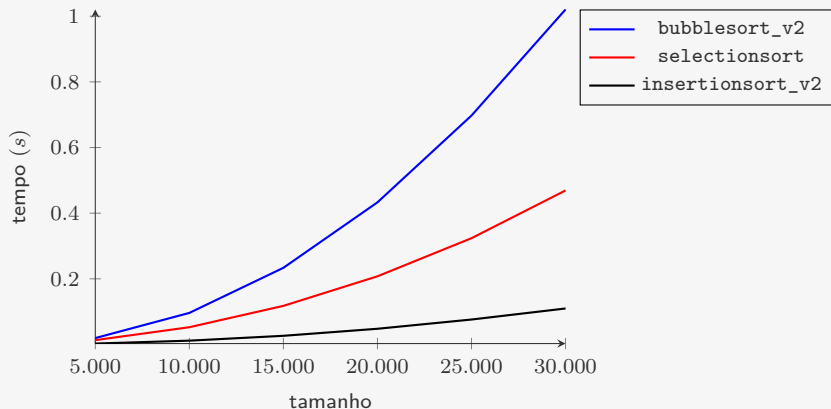
```
1 void selectionsort(int *v, int n) {
2     int i, j, min;
3     for (i = 0; i < n - 1; i++) {
4         min = i;
5         for (j = i+1; j < n; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

# Ordenação por Seleção

```
1 void selectionsort(int *v, int n) {
2     int i, j, min;
3     for (i = 0; i < n - 1; i++) {
4         min = i;
5         for (j = i+1; j < n; j++)
6             if (v[j] < v[min])
7                 min = j;
8         troca(&v[i], &v[min]);
9     }
10 }
```

- número de comparações:  
 $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$
- número de trocas:  $n - 1 = O(n)$ 
  - Muito bom quando trocas são muito caras
  - Porém, talvez seja melhor usar ponteiros nesse caso

## Gráfico de comparação do três algoritmos



Para  $n = 30.000$ :

- selectionsort leva 4,2 o tempo do insertionsort\_v2
- bubblesort\_v2 leva 9,3 o tempo do insertionsort\_v2

# Voltando ao SelectionSort

Versão do SelectionSort que

- coloca o elemento máximo na posição  $v[n - 1]$
- coloca o segundo maior elemento na posição  $v[n - 2]$
- etc...

```
1 int selection_invertido(int *v, int n) {
2     int i, j, max;
3     for (i = n - 1; i > 0; i--) {
4         max = i;
5         for (j = i-1; j >= 0; j--)
6             if (v[j] > v[max])
7                 max = j;
8         troca(&v[i], &v[max]);
9     }
10 }
```

## Rescrevendo...

Usamos uma função que acha o elemento máximo do vetor

```
1 int extrai_maximo(int *v, int n) {
2     int max = n - 1;
3     for (j = n - 2; j >= 0; j--)
4         if (v[j] > v[max])
5             max = j;
6     return max;
7 }
```

E reescrevemos o SelectionSort

```
1 int selection_invertido_v2(int *v, int n) {
2     int i, j, max;
3     for (i = n - 1; i > 0; i--) {
4         max = extrai_maximo(v, i + 1);
5         troca(&v[i], &v[max]);
6     }
7 }
```

## Tempo do SelectionSort

```
1 int selection_invertido_v2(int *v, int n) {
2     int i, j, max;
3     for (i = n - 1; i > 0; i--) {
4         max = extrai_maximo(v, i + 1);
5         troca(&v[i], &v[max]);
6     }
7 }
```

O tempo do `selection_invertido_v2` é:

- o tempo de chamar `extrai_maximo(v, i + 1)`
  - com `i` variando de `n - 1` a `1`
- mais o tempo de fazer `n - 1` trocas

$T(k)$ : tempo de extrair o máximo de um vetor com  $k$  elementos

Para ordenar  $n$  elementos, o SelectionSort gasta tempo

$$n-1 + \sum_{k=2}^n T(k) = n-1 + \sum_{k=2}^n c \cdot k = n-1 + c \cdot \frac{(n+2)(n-1)}{2} = O(n^2)$$

Mas, com heap, podemos extrair o máximo em  $O(\lg k)$

# Ordenação usando Fila de Prioridades

```
1 void fpsort(Item *v, int n) {
2     int i;
3     p_fp fprio = criar_fprio(n);
4     for (i = 0; i < n; i++)
5         insere(fprio, v[i]);
6     for (i = n - 1; i >= 0; i--)
7         v[i] = extrai_maximo(fprio);
8     destroi(fprio);
9 }
```

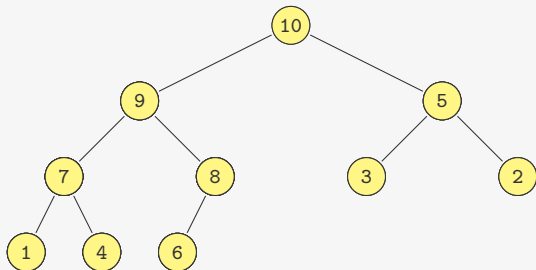
Tempo:  $O(n \lg n)$

- Estamos usando espaço adicional, mas não precisamos...
- Perdemos tempo para copiar do vetor para o heap
- Podemos transformar um vetor em um heap rapidamente
  - Mais rápido do que fazer  $n$  inserções



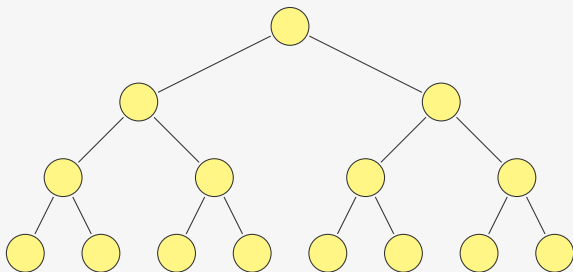
## Transformando um vetor em um heap

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



Quanto tempo demora?

## Tempo da construção para $n = 2^k - 1$



- Temos  $2^{k-1}$  heaps de altura 1
- Temos  $2^{k-2}$  heaps de altura 2
- Temos  $2^{k-h}$  heaps de altura  $h$
- Cada heap de altura  $h$  consome tempo  $c \cdot h$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h} = c \cdot 2^k \sum_{h=1}^{k-1} \frac{h}{2^h}$$

## Tempo da construção para $n = 2^k - 1$

$$\sum_{h=1}^{k-1} c \cdot h \cdot 2^{k-h} = c \cdot 2^k \sum_{h=1}^{k-1} \frac{h}{2^h}$$

Note que

$$\begin{aligned} \sum_{h=1}^{k-1} \frac{h}{2^h} &= \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} = 1 - \frac{1}{2^{k-1}} < 1 \\ &+ \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} = \frac{1}{2} - \frac{1}{2^{k-1}} < \frac{1}{2} \\ &+ \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} = \frac{1}{4} - \frac{1}{2^{k-1}} < \frac{1}{4} \\ &\cdots + \frac{1}{2^{k-1}} = \frac{1}{2^{r-1}} - \frac{1}{2^{k-1}} < \frac{1}{2^{r-1}} \end{aligned}$$

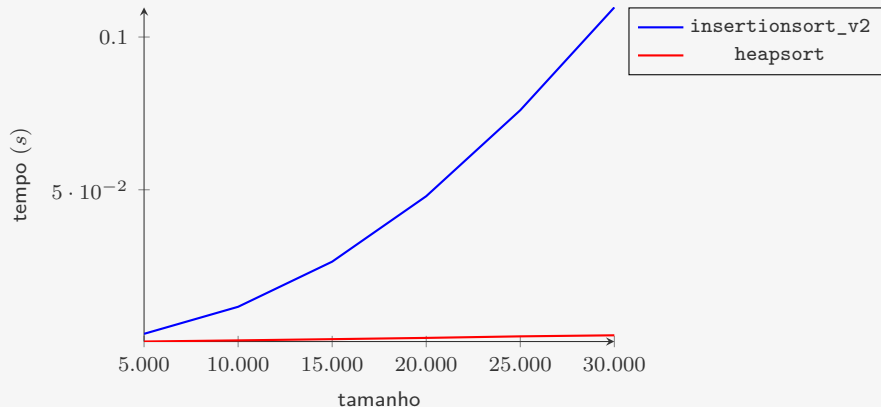
Ou seja,

$$c \cdot 2^k \sum_{h=1}^{k-1} \frac{h}{2^h} \leq c \cdot 2^k \cdot 2 = O(2^k) = O(n)$$

# Heapsort

```
1 void desce_no_heap(int *heap, int n, int k) {
2     int maior_filho;
3     if (F_ESQ(k) < n) {
4         maior_filho = F_ESQ(k);
5         if (F_DIR(k) < n &&
6             heap[F_ESQ(k)] < heap[F_DIR(k)])
7             maior_filho = F_DIR(k);
8         if (heap[k] < heap[maior_filho]) {
9             troca(&heap[k], &heap[maior_filho]);
10            desce_no_heap(heap, n, maior_filho);
11        }
12    }
13 }
14
15 void heapsort(int *v, int n) {
16     int k;
17     for (k = n / 2; k >= 0; k--) /* transforma em heap */
18         desce_no_heap(v, n, k);
19     while (n > 1) { /* extrai o máximo */
20         troca(&v[0], &v[n - 1]);
21         n--;
22         desce_no_heap(v, n, 0);
23     }
24 }
```

## Vale a pena um algoritmo $O(n \lg n)$ ?



Para  $n = 30.000$ :

- **heapsort** leva em média **0.002369s**
- **insertionsort\_v2** leva em média **0.109704s**
  - **46,3** vezes o tempo do **heapsort**

# Conclusão

Vimos três algoritmos  $O(n^2)$ :

- **bubblesort**: na prática é o pior dos três, raramente usado
- **selectionsort**: bom quando comparações são muito mais baratas que trocas
- **insertionsort**: o melhor dos três na prática
  - Vimos otimizações do código que melhoraram os resultados empíricos
  - Durante o curso, não focaremos em otimizações como essas...

E vimos um algoritmo melhor **assintoticamente**

- **heapsort** é  $O(n \lg n)$
- Melhor do que qualquer algoritmo  $O(n^2)$ 
  - Mesmo na versão mais otimizada

## Exercício

```
1 void bubblesort_v2(int *v, int n) {
2     int i, j, trocou = 1;
3     for (i = 0; i < n - 1 && trocou; i++){
4         trocou = 0;
5         for (j = n - 1; j > i; j--){
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10    }
11 }
```

Quando ocorre o pior caso do `bubblesort_v2`?

Quando ocorre o melhor caso do `bubblesort_v2`?

- Quantas comparações são feitas no melhor caso?
- Quantas trocas são feitas no melhor caso?

## Exercício

```
1 void insertionsort_v2(int *v, int n) {
2     int i, j, t;
3     for (i = 1; i < n; i++) {
4         t = v[i];
5         for (j = i; j > 0 && t < v[j-1]; j--)
6             v[j] = v[j-1];
7         v[j] = t;
8     }
9 }
```

Quando ocorre o pior caso do `insertionsort_v2`?

Quando ocorre o melhor caso do `insertionsort_v2`?

- Quantas comparações são feitas no melhor caso?
- Quantas atribuições são feitas no melhor caso?



## Exercício

Em `sobe_no_heap` trocamos `k` com `PAI(k)`, `PAI(k)` com `PAI(PAI(k))` e assim por diante. Algo similar acontece com `desce_no_heap`. Modifique as versões iterativas das duas funções para diminuir o número de atribuições (como feito no `InsertionSort`).

# Solução

```
1 void sobe_no_heap(p_fp fprio, int k) {
2     Item aux = fprio->v[k];
3     while (k > 0 && fprio->v[PAI(k)].chave < aux.chave) {
4         fprio->v[k] = fprio->v[PAI(k)];
5         k = PAI(k);
6     }
7     fprio->v[k] = aux;
8 }
```

# Solução

```
9 void desce_no_heap(p_fp fprio, int k) {
10     int maior_filho, continuar = 1;
11     Item aux = fprio->v[k];
12     while (F_ESQ(k) < fprio->n && continuar) {
13         continuar = 0;
14         maior_filho = F_ESQ(k);
15         if (F_DIR(k) < fprio->n &&
16             fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
17             maior_filho = F_DIR(k);
18         if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
19             fprio->v[k] = fprio->v[maior_filho];
20             k = maior_filho;
21             continuar = 1;
22         }
23     }
24     fprio->v[k] = aux;
25 }
```

Dúvidas?