Programação Procedimental

Ordenação (parte 1)

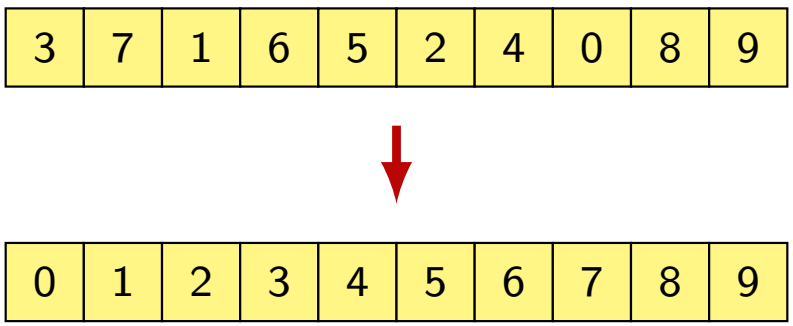
Aula 13

Prof. Felipe A. Louza

Roteiro

Ordenação

Queremos ordenar um vetor

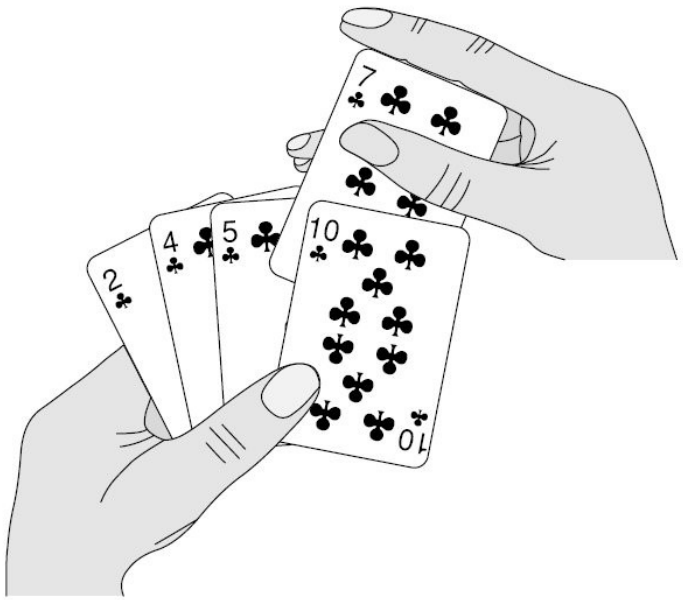


Nos códigos vamos ordenar vetores de `int`

- 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

Ordenação por Seleção

Ordenação por Seleção:

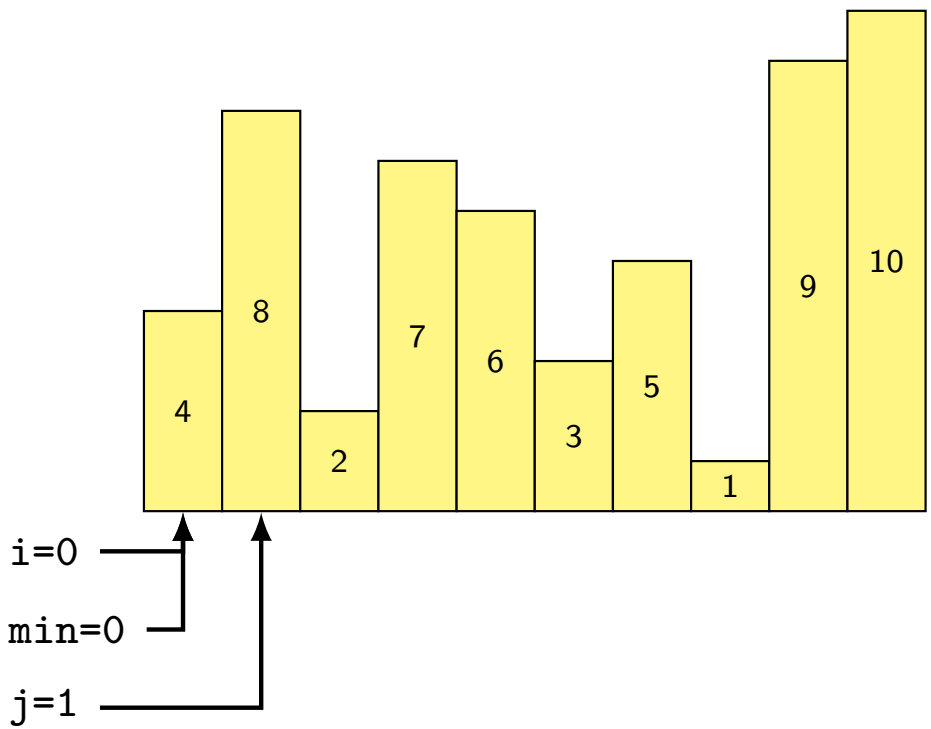


Ordenação por Seleção

Ideia:

- Trocar `v[0]` com o mínimo de `v[0]`, `v[1]`, ..., `v[n - 1]`
- Trocar `v[1]` com o mínimo de `v[1]`, ..., `v[n - 1]`
- ...
- Trocar `v[i]` com o mínimo de `v[i]`, ..., `v[n-1]`

```
1 void selection_sort(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 selection_sort(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:
  $(n-1)+(n-2)+\cdots+1 = n(n-1)/2 = \mathrm{O}(n^2)$
- número de trocas: $n-1 = \mathrm{O}(n)$
  - Muito bom quando trocas são muito caras

Ordenação por Seleção

Existem variações do Selection Sort

- Busca invertida pelo maior valor
- Custo computacional também é $\mathrm{O}(n^2)$

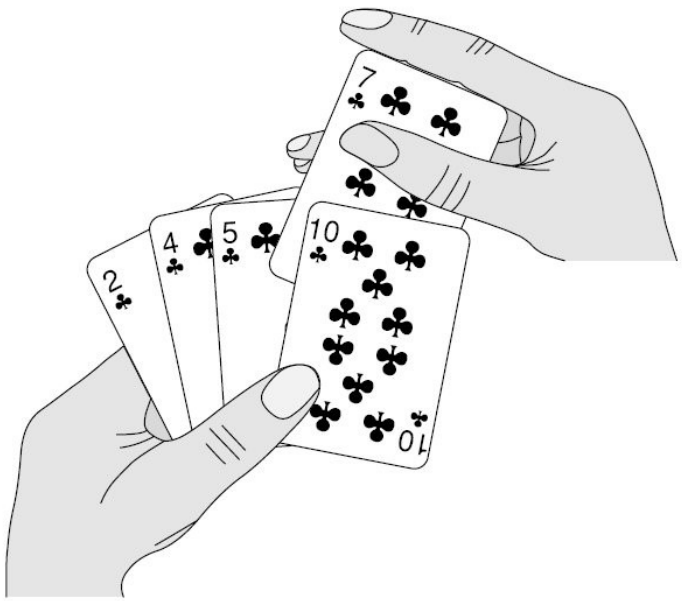
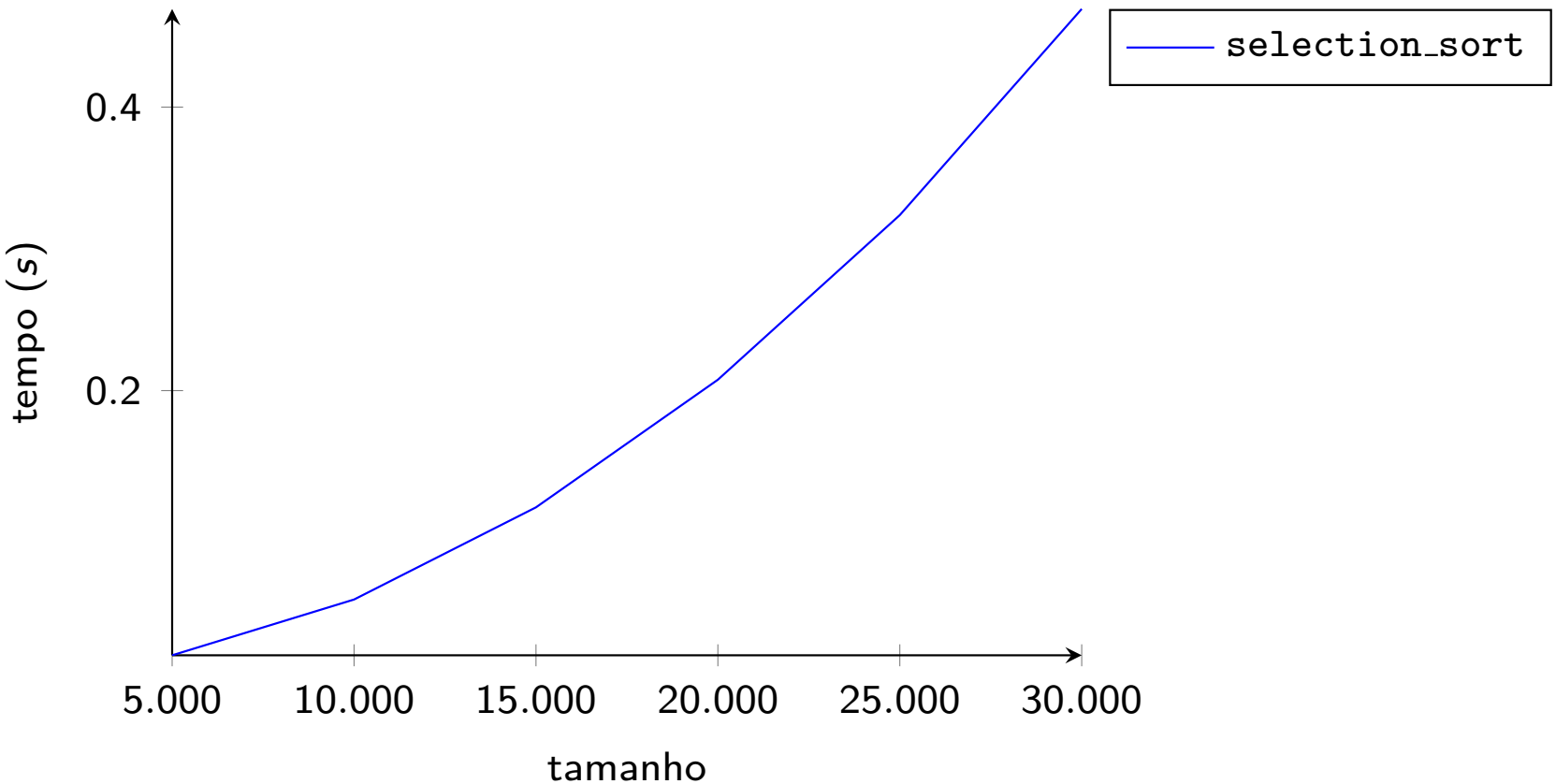


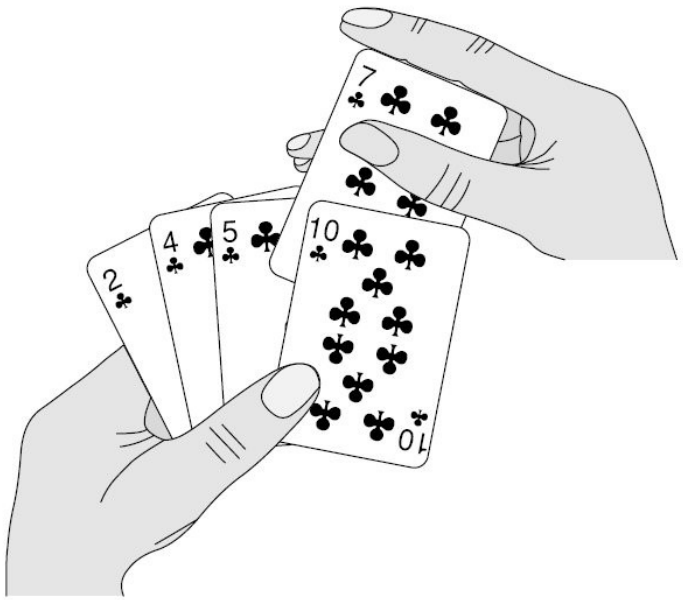
Gráfico de comparação dos algoritmos



Roteiro

Ordenação por Inserção

Ordenação por Inserção:

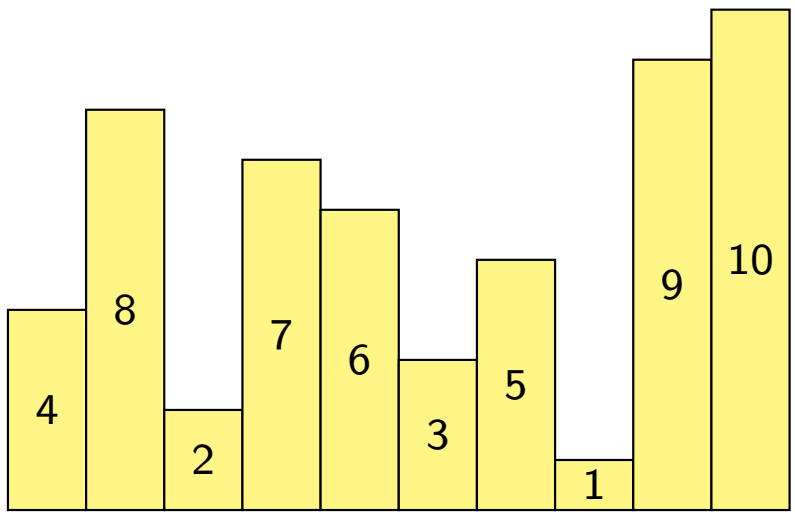


Ordenação por Inserção

Ideia:

- Se já temos `v[0], v[1], ..., v[i-1]` ordenado
- Inserimos `v[i]` na posição correta
- Ficamos com `v[0], v[1], ..., 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 }
```



i

j

Insertion-Sort

Análise de pior caso:

```
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 }
```

- O 2º elemento é inserido com 1 comparação.
- O 3º elemento é inserido com 2 comparações.
- ...
- O nº elemento é encontrado com $n - 1$ comparações.
- Custo assintótico:

$$T(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \mathrm{O}(n^2)$$

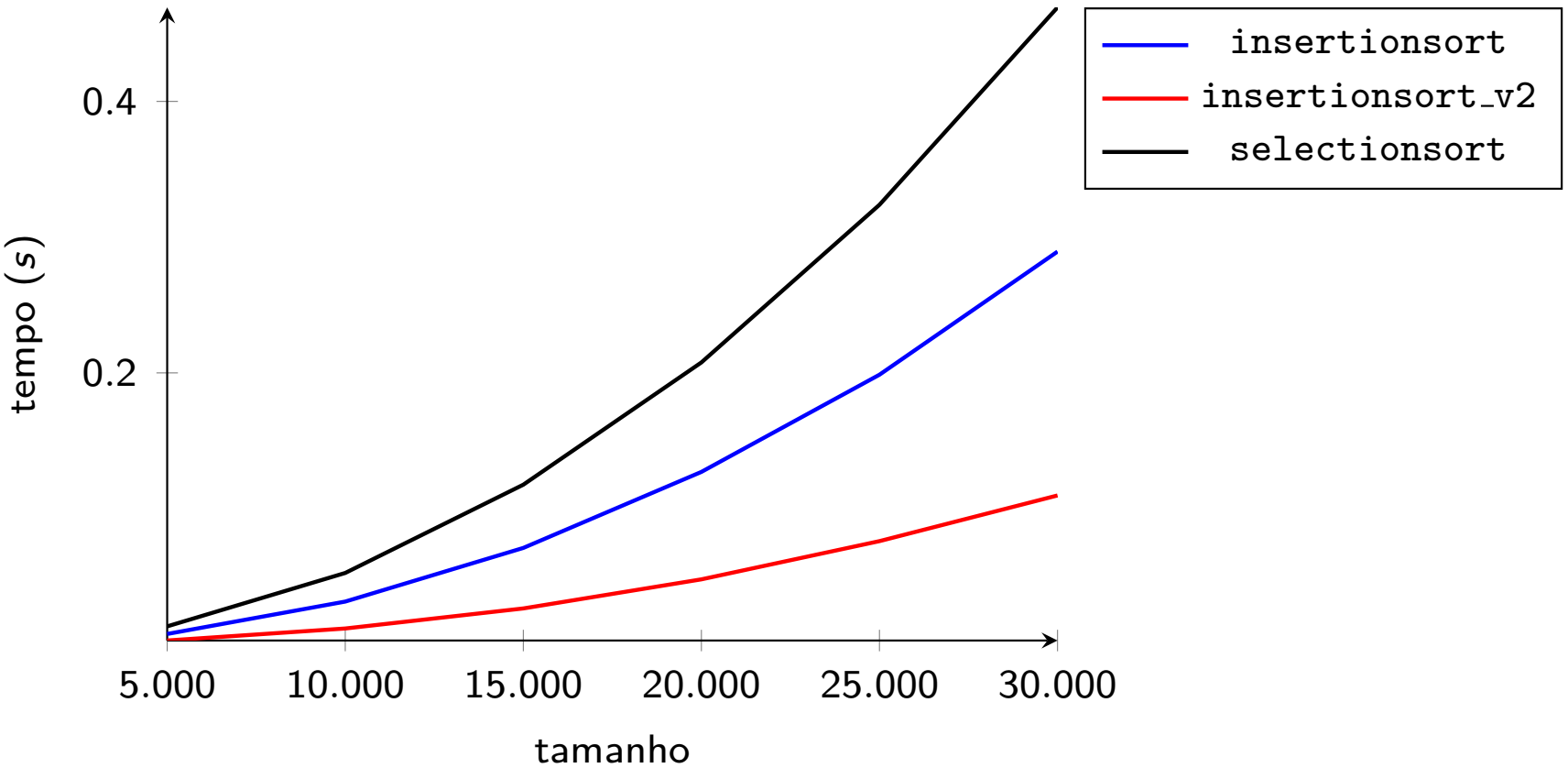
Ordenação por Inserção - Otimizações

Existem variações do Insertion Sort

1. Quando o elemento já está na sua posição correta não é necessário mais percorrer o vetor testando se `v[j] < v[j-1]`
2. Quando 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[i]; 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 }
```

Gráfico de comparação do InsertionSort



A complexidade teórica do algoritmo não melhorou

- continua $\mathrm{O}(n^2)$

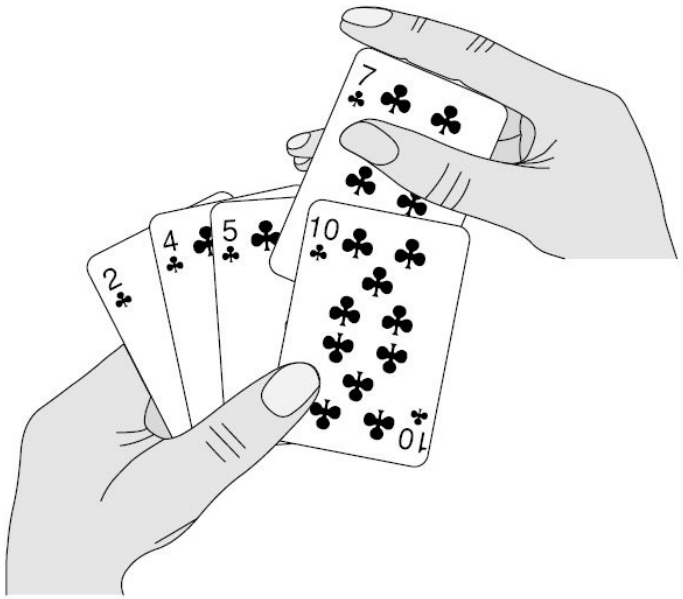
Mas as otimizações levaram a um ganho na performance

- menos do que a metade do tempo para n grande

Roteiro

BubbleSort

BubbleSort:

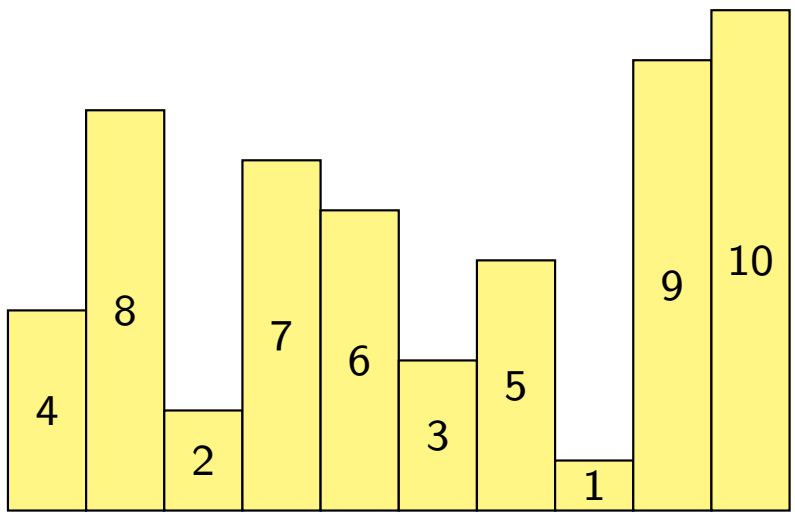


BubbleSort

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 à esquerda

```
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 }
```



i

j

Parando quando não há mais trocas

Existem variações do Bubble Sort

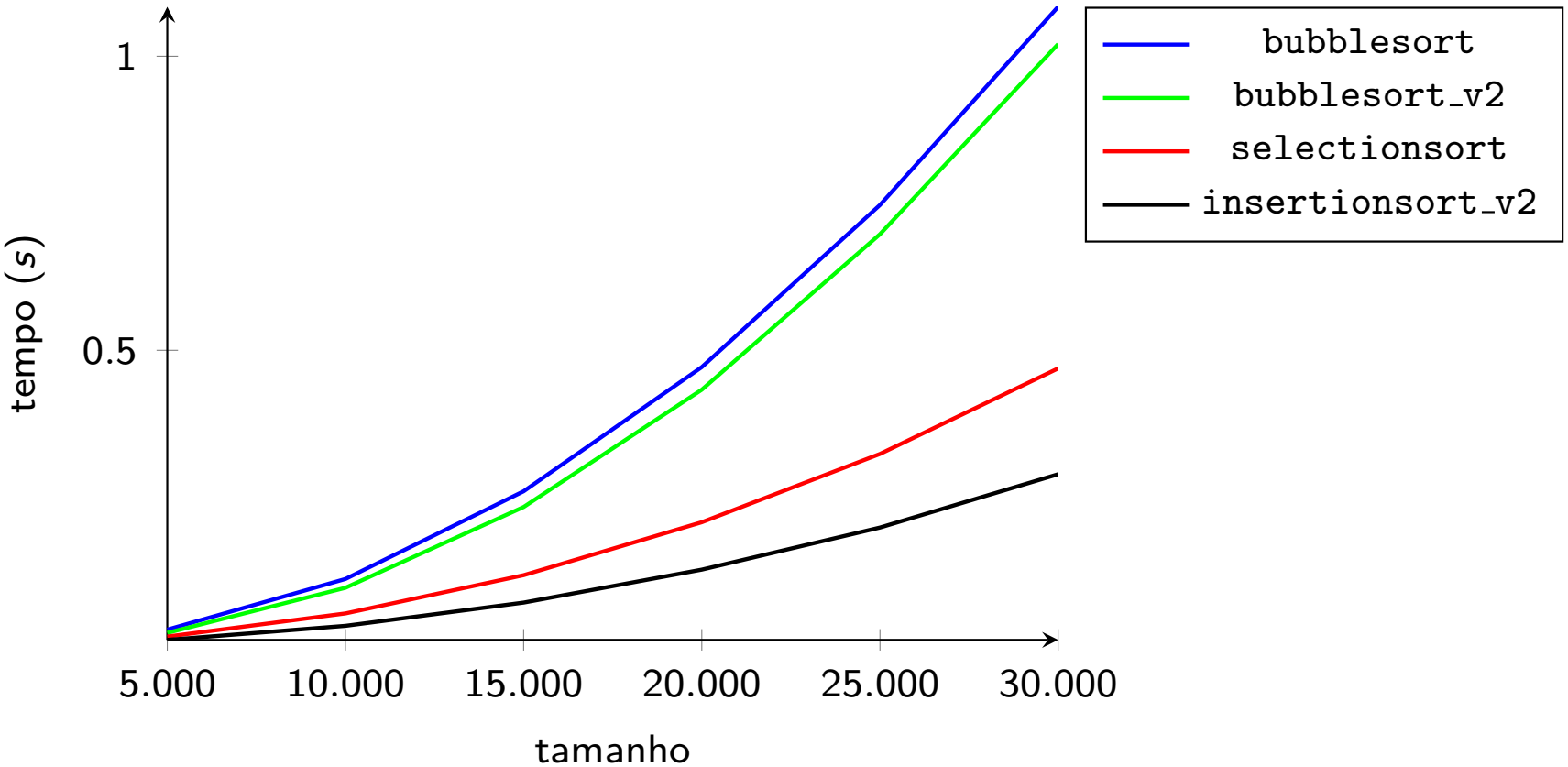
- Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
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: $n(n-1)/2 = \mathrm{O}(n^2)$
- trocas: $n(n-1)/2 = \mathrm{O}(n^2)$

Gráfico de comparação do BubbleSort



A segunda versão do BubbleSort é um pouco mais rápida, ainda assim, mais lenta do que os outros algroithmos

Conclusão

Vimos três algoritmos $\mathrm{O}(n^2)$:

- `bubblesort`: na pratica 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

Vamos ver algoritmos melhores..

- `MergeSort`: $\mathrm{O}(n \lg n)$
- `QuickSort`: $\mathrm{O}(n^2)$

Fim

Dúvidas?

Roteiro

Referências

1. Materiais adaptados dos slides dos Profs. Rafael Schouery e Lehilton L. C. Pedrosa, da UNICAMP.