MC-202
Ordenação

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

2º semestre/2023

Ordenação

Queremos ordenar um vetor

3	7	1	6	5	2	4	0	8	9

$\downarrow$

0	1	2	3	4	5	6	7	8	9

Nos códigos vamos ordenar vetores de `int`

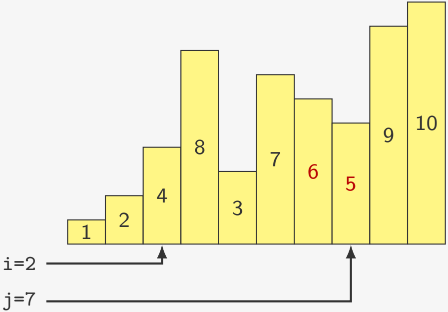
- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
  - O valor usado para a ordenação é a *chave* de ordenação
  - Podemos até desempatar por outros campos

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

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



Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
void bubblesort_v2(int *v, int n) {
  int i, j, trocou = 1;
  for (i = 0; i < n - 1 && trocou; i++){
    trocou = 0;
    for (j = n - 1; j > i; j--)
      if (v[j] < v[j-1]) {
        troca(&v[j-1], &v[j]);
        trocou = 1;
      }
  }
}
```

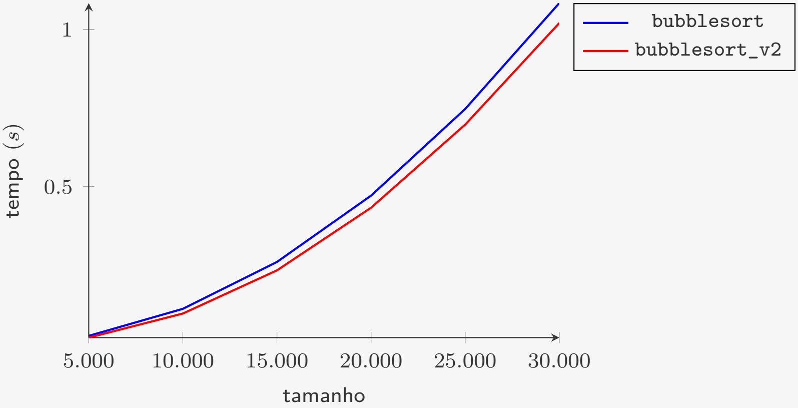
No pior caso toda comparação gera uma troca:

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

No caso médio:

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

Gráfico de comparação do BubbleSort



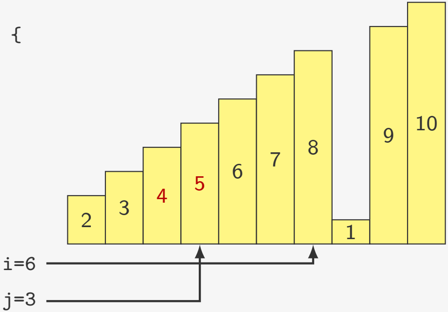
A segunda versão é um pouco mais rápida

Ordenação por Inserção

Ideia:

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

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



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;`

```c
void insertionsort_v2(int *v, int n) {
  int i, j, t;
  for (i = 1; i < n; i++) {
    t = v[i];
    for (j = i; j > 0 && t < v[j-1]; j--)
      v[j] = v[j-1];
    v[j] = t;
  }
}
```

Ordenação por Inserção — Análise

```
void insertionsort_v2(int *v, int n) {
  int i, j, t;
  for (i = 1; i < n; i++) {
    t = v[i];
    for (j = i; j > 0 && t < v[j-1]; j--)
      v[j] = v[j-1];
    v[j] = t;
  }
}
```

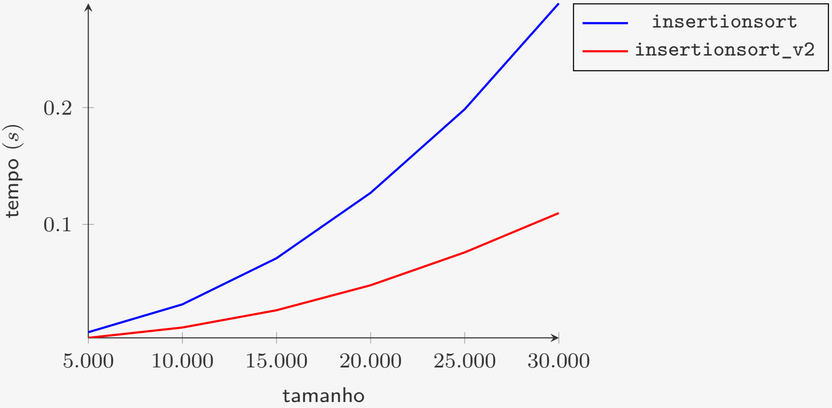
No caso pior caso:

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

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

Mas as otimizações levaram a um ganho na performance

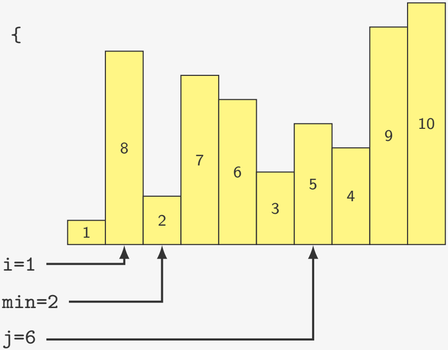
- menos do que a metade do tempo para n grande

Ordenação por Seleção

Ideia:

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

```c
void selectionsort(int *v, int n) {
  int i, j, min;
  for (i = 0; i < n - 1; i++) {
    min = i;
    for (j = i+1; j < n; j++)
      if (v[j] < v[min])
        min = j;
    troca(&v[i], &v[min]);
  }
}
```

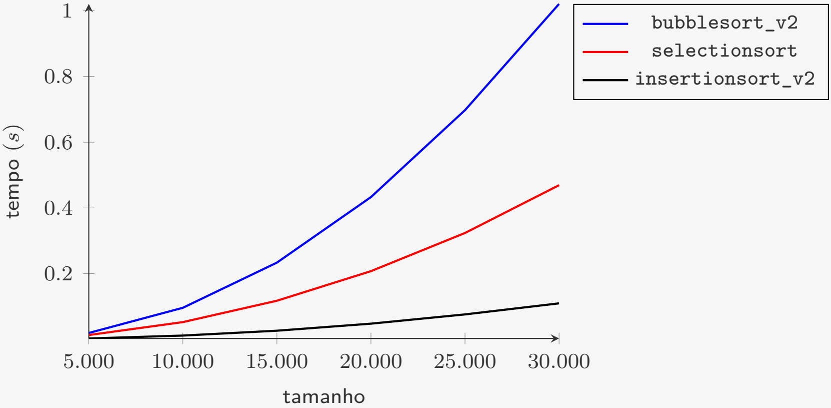


Ordenação por Seleção

```c
void selectionsort(int *v, int n) {
  int i, j, min;
  for (i = 0; i < n - 1; i++) {
    min = i;
    for (j = i+1; j < n; j++)
      if (v[j] < v[min])
        min = j;
    troca(&v[i], &v[min]);
  }
}
```

- 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
  - 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...

```
int selection_invertido(int *v, int n) {
  int i, j, max;
  for (i = n - 1; i > 0; i--) {
    max = i;
    for (j = i-1; j >= 0; j--)
      if (v[j] > v[max])
        max = j;
    troca(&v[i], &v[max]);
  }
}
```

Rescrevendo...

Usamos uma função que acha o elemento máximo do vetor

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

E reescrevemos o SelectionSort

```c
int selection_invertido_v2(int *v, int n) {
  int i, j, max;
  for (i = n - 1; i > 0; i--) {
    max = extrai_maximo(v, i + 1);
    troca(&v[i], &v[max]);
  }
}
```

Tempo do SelectionSort

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

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

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

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

Ordenação usando Fila de Prioridades

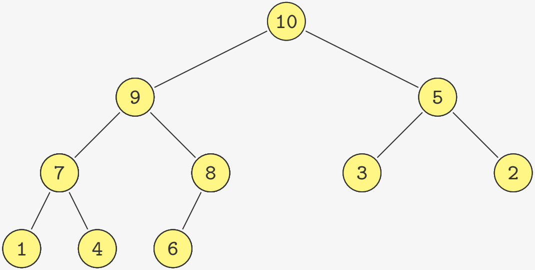
```
void fpsort(Item *v, int n) {
  int i;
  p_fp fprio = criar_fprio(n);
  for (i = 0; i < n; i++)
    insere(fprio, v[i]);
  for (i = n - 1; i >= 0; i--)
    v[i] = extrai_maximo(fprio);
  destroi(fprio);
}
```

Tempo: $\mathrm{O}(n \lg n)$

- 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

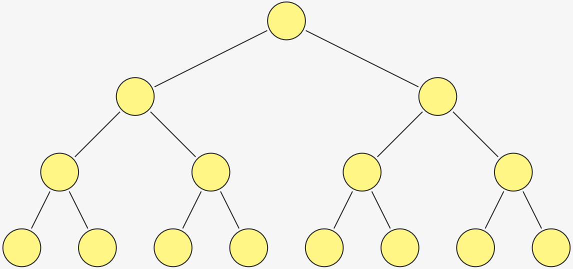
Transformando um vetor em um heap

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

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

Ou seja,

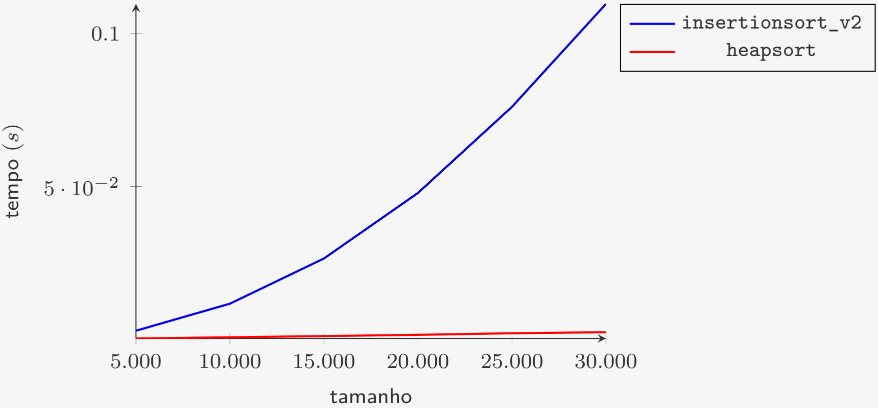
$$c \cdot 2^k \sum_{h=1}^{k-1} \frac{h}{2^h} \leq c \cdot 2^k \cdot 2 = \mathrm{O}(2^k) = \mathrm{O}(n)$$

Heapsort

```c
void desce_no_heap(int *heap, int n, int k) {
  int maior_filho;
  if (F_ESQ(k) < n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < n &&
        heap[F_ESQ(k)] < heap[F_DIR(k)])
      maior_filho = F_DIR(k);
    if (heap[k] < heap[maior_filho]) {
      troca(&heap[k], &heap[maior_filho]);
      desce_no_heap(heap, n, maior_filho);
    }
  }
}

void heapsort(int *v, int n) {
  int k;
  for (k = n / 2; k >= 0; k--) /* transforma em heap */
    desce_no_heap(v, n, k);
  while (n > 1) { /* extrai o máximo */
    troca(&v[0], &v[n - 1]);
    n--;
    desce_no_heap(v, n, 0);
  }
}
```

Vale a pena um algoritmo $\mathrm{O}(n \lg n)$?



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 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
  - 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

```c
void bubblesort_v2(int *v, int n) {
  int i, j, trocou = 1;
  for (i = 0; i < n - 1 && trocou; i++){
    trocou = 0;
    for (j = n - 1; j > i; j--)
      if (v[j] < v[j-1]) {
        troca(&v[j-1], &v[j]);
        trocou = 1;
      }
  }
}
```

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

```c
void insertionsort_v2(int *v, int n) {
  int i, j, t;
  for (i = 1; i < n; i++) {
    t = v[i];
    for (j = i; j > 0 && t < v[j-1]; j--)
      v[j] = v[j-1];
    v[j] = t;
  }
}
```

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).