SCC-601– Introdução à Ciência da Computação II

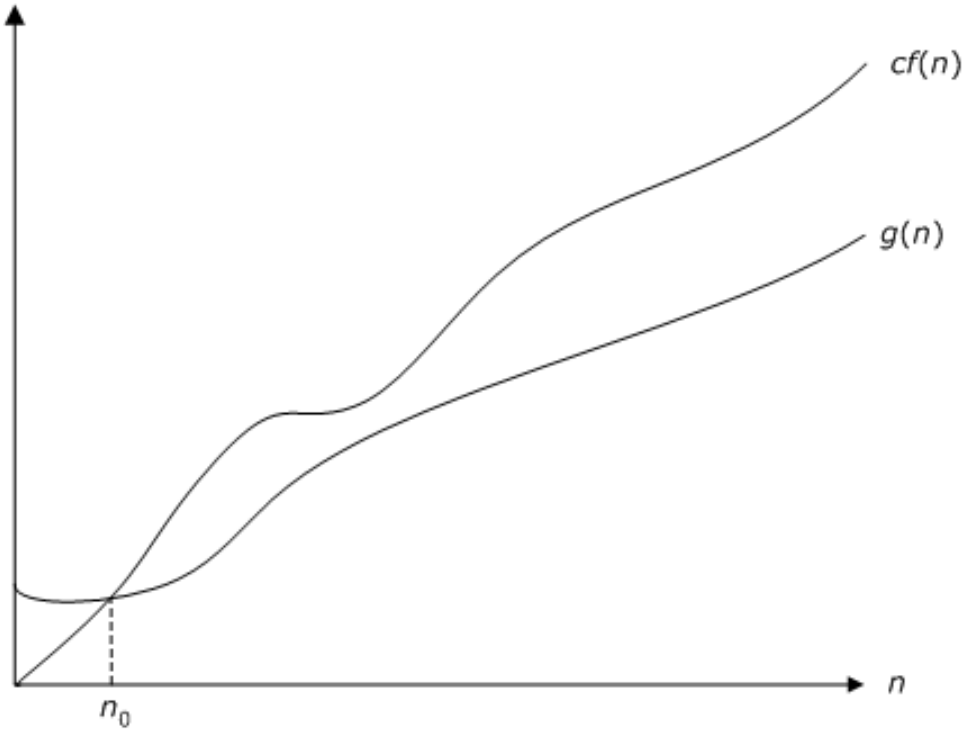
Ordenação e Complexidade – Parte 2

Lucas Antiqueira

Notação Big-O

$g(n) = O(f(n))$ (g é da ordem de f) se existirem constantes c e n_0 tal que $g(n) \leq c*f(n)$ quando $n \geq n_0$

Ou seja, a taxa de crescimento de g(n) é menor ou igual à taxa de f(n) a partir de n_0



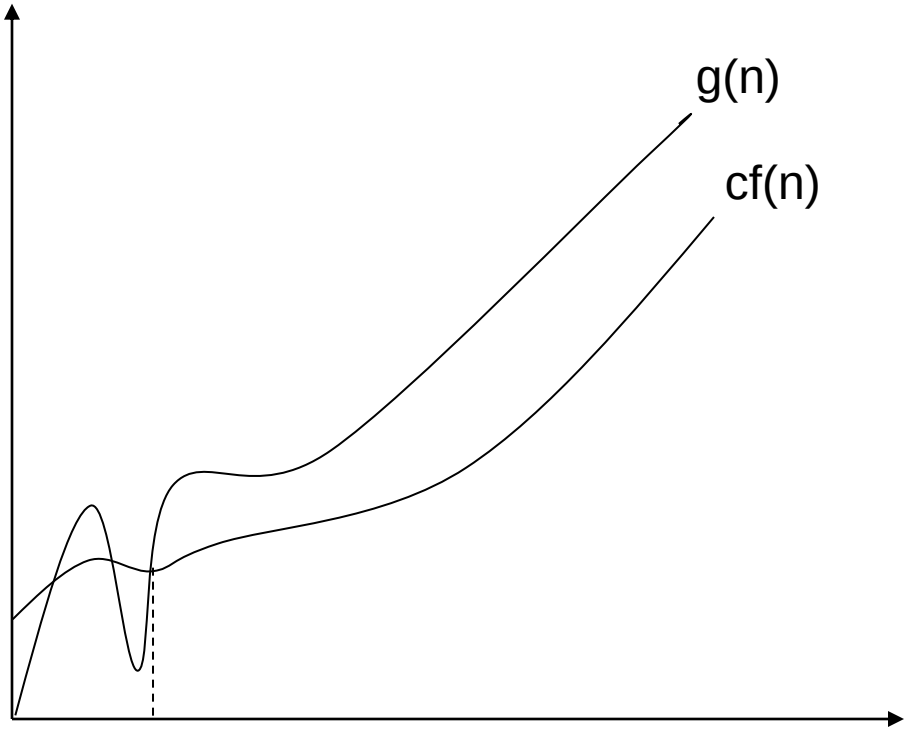
Mais notações?

?

Notação Big-Omega

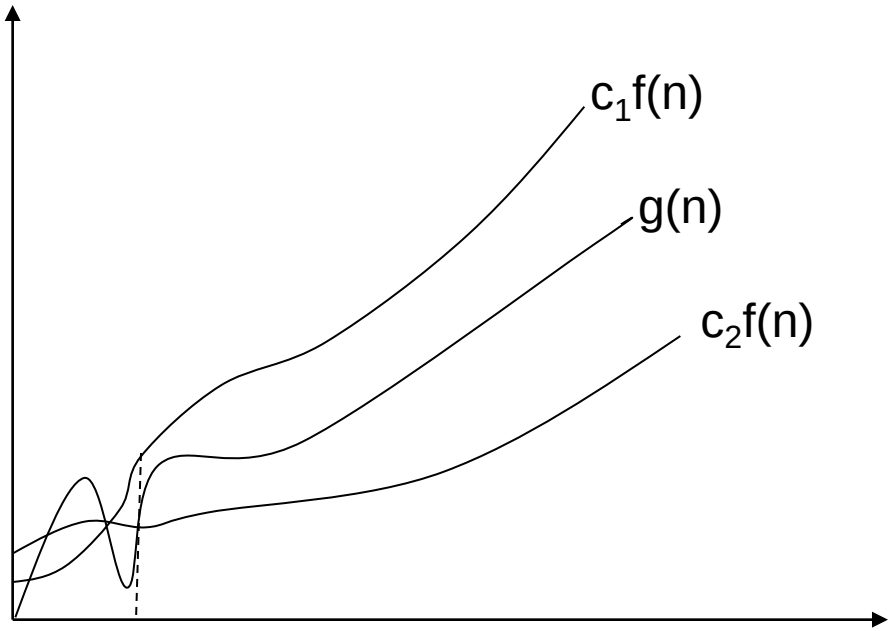
$g(n) = \Omega(f(n))$ se existirem constantes c e n_0 tal que $g(n) \geq c*f(n)$ quando $n \geq n_0$

Ou seja, a taxa de crescimento de g(n) é maior ou igual à taxa de f(n) a partir de n_0



Notação Big-Theta

$g(n) = \Theta(f(n))$ se e somente se $g(n) = O(f(n))$ e $g(n) = \Omega(f(n))$



Exercício

- Um algoritmo tradicional e muito utilizado é da ordem de $n^{1,5}$, enquanto um algoritmo novo proposto recentemente é da ordem de $n \log_2 n$
  - $g_1(n)=n^{1,5}$
  - $g_2(n)=n \log_2 n$

- Qual algoritmo você adotaria?

Exercício

- Solução:

$g_1(n)$	?	$g_2(n)$		
$n^{1,5}$	?	$n \log_2 n$		
$n^{1,5}/n$	?	$(n \log_2 n)/n$		
$n^{0,5}$	?	$\log_2 n$		
$(n^{0,5})^2$	?	$(\log_2 n)^2$		
n	$\geq$	$(\log_2 n)^2$	,	para $n \geq n_0$

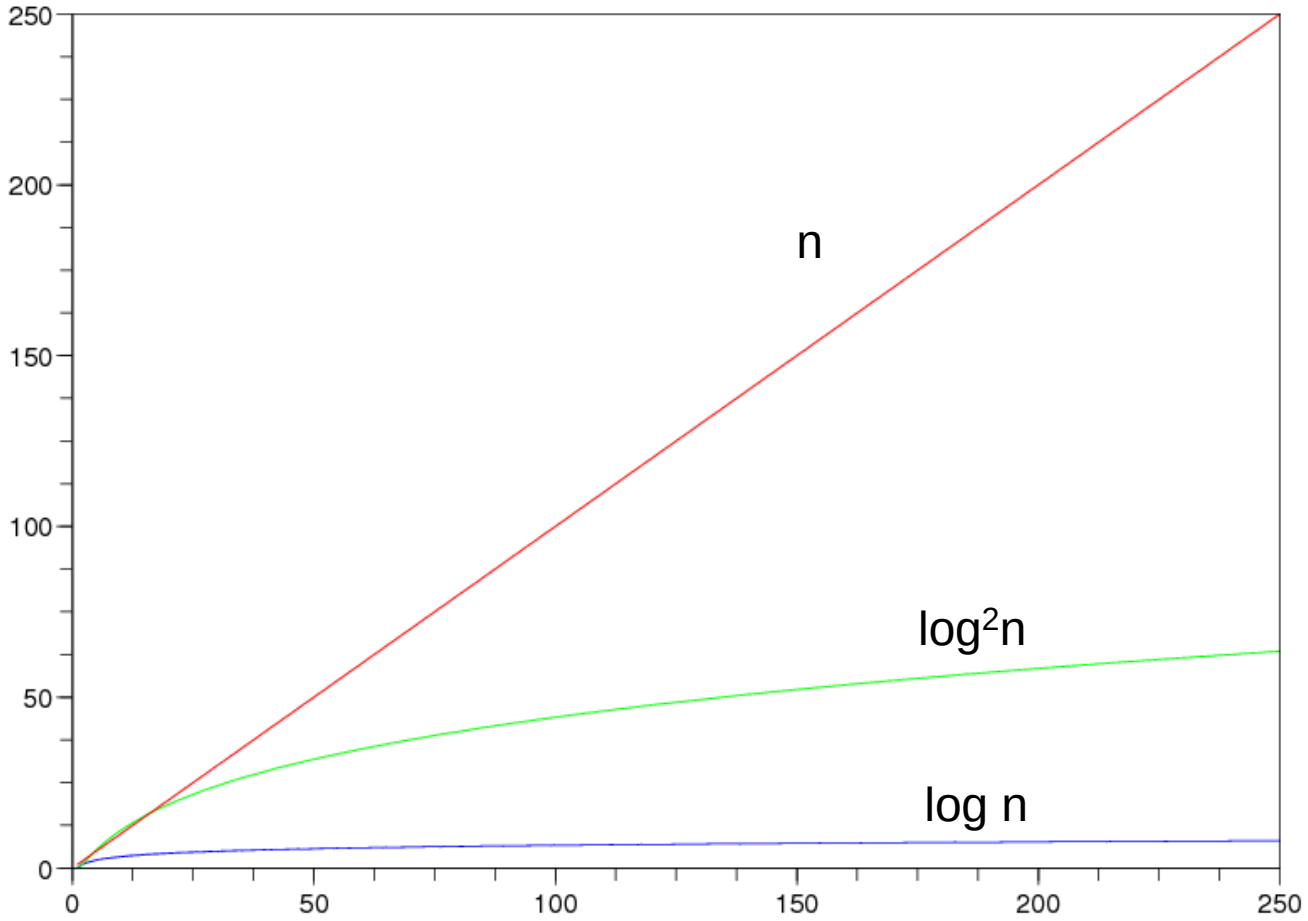
Como n cresce mais rapidamente do que qualquer potência de log, temos que o algoritmo novo é mais eficiente.

Taxas de crescimento

- As mais comuns:

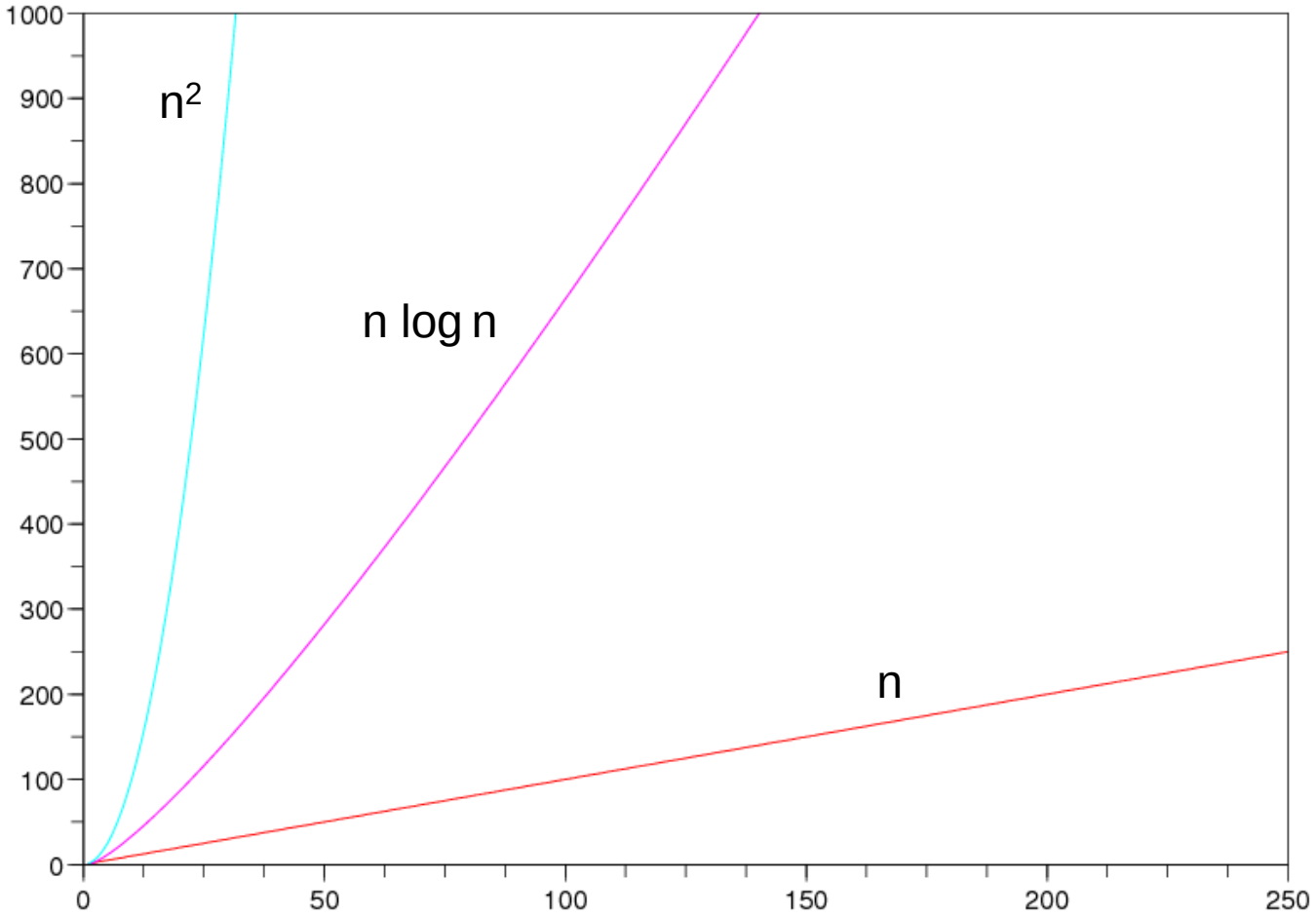
c
$\log n$
$\log^2 n$
n
$n \log n$
n^2
n^3
2^n
a^n
$n!$

Taxas de crescimento

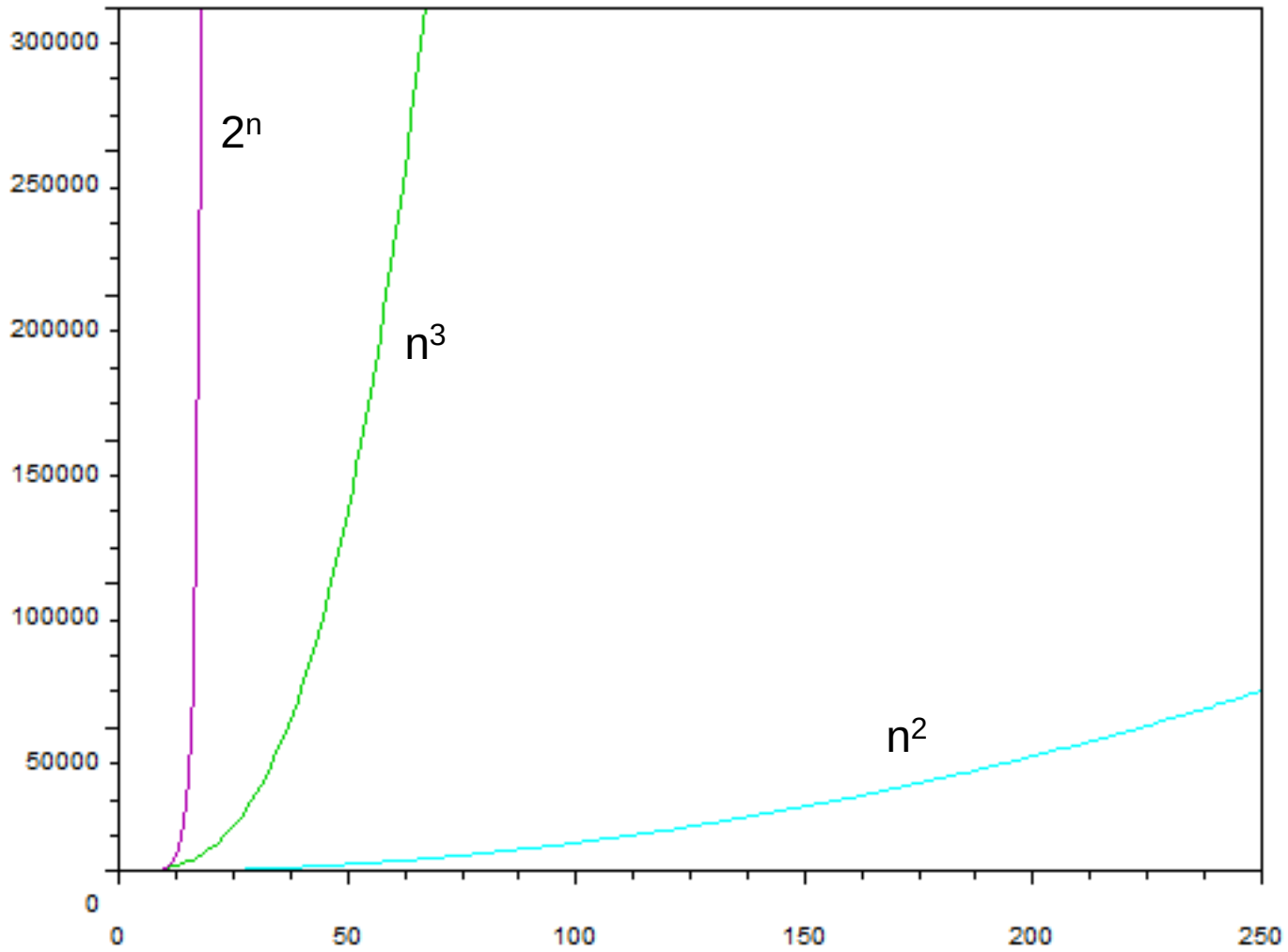


Obs.: log base 2

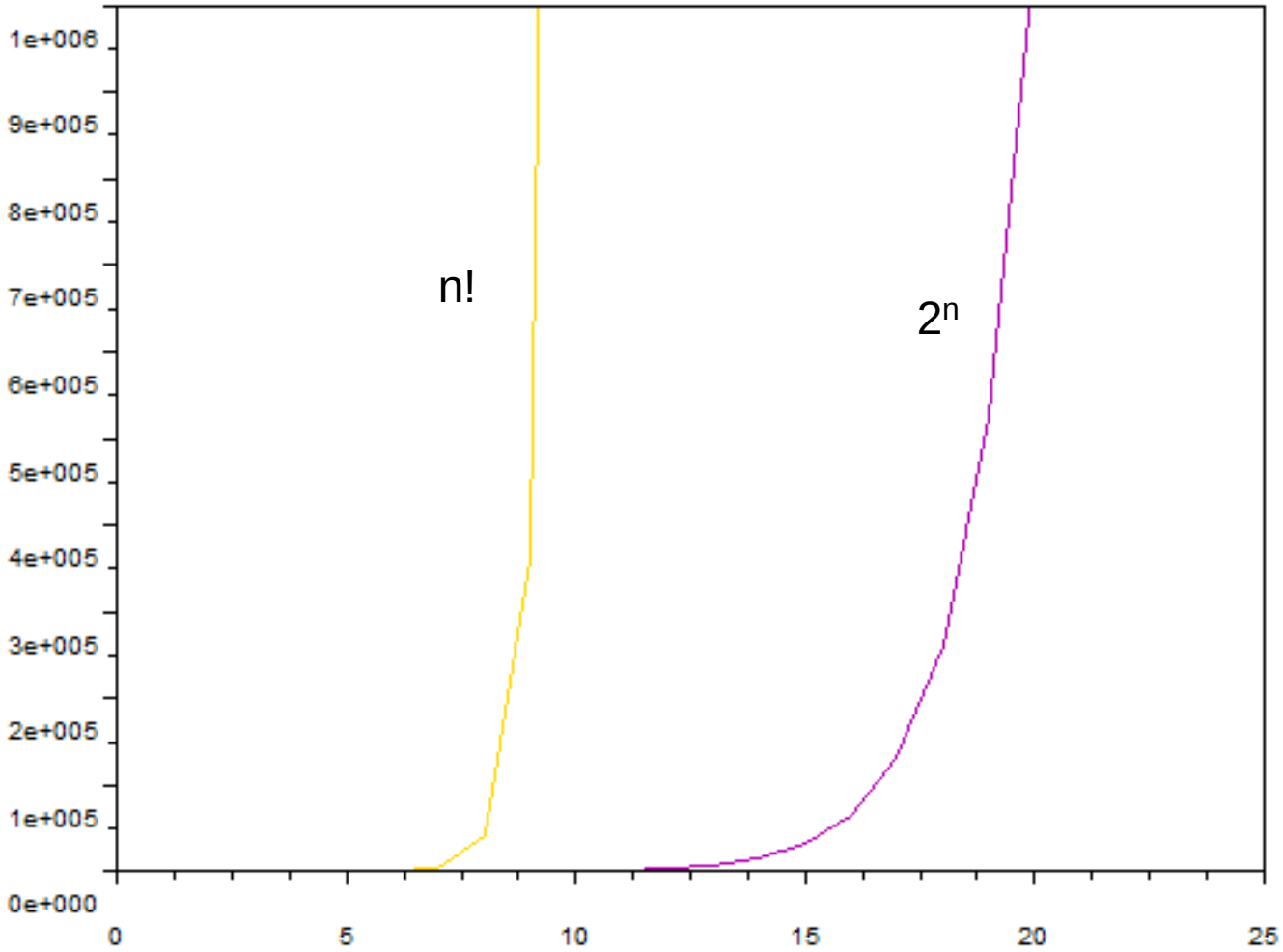
Taxas de crescimento



Taxas de crescimento



Taxas de crescimento

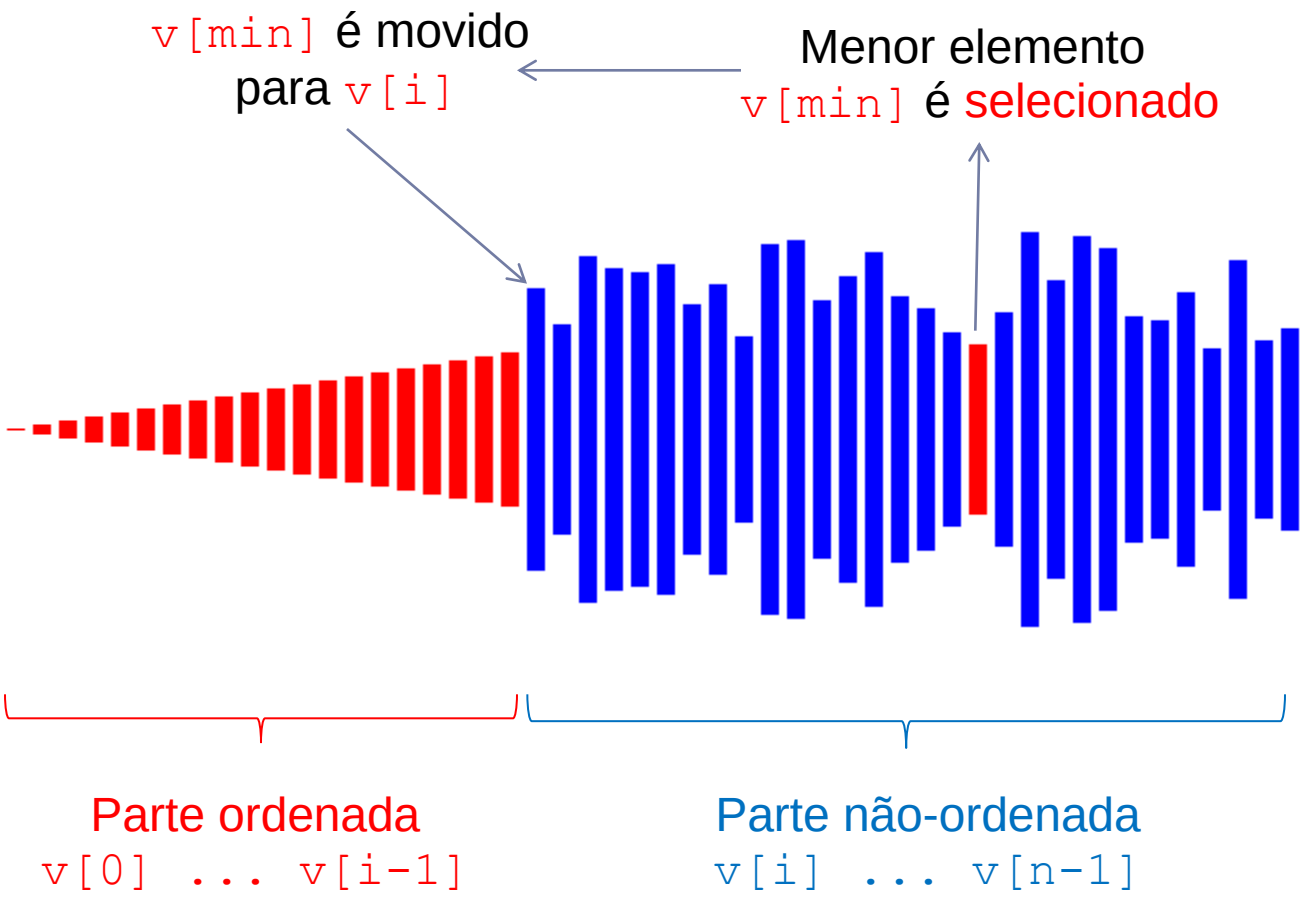


Ordenação: Seleção Direta

Seleção Direta

- Método
  1. Selecionar o elemento que apresenta o menor valor
  2. Trocar o elemento de lugar com o primeiro elemento da parte não-ordenada
  3. Repetir as operações 1 e 2, envolvendo agora apenas os n-1 elementos restantes, depois os n-2 elementos, etc, até restar somente um elemento, o maior deles

Seleção Direta



- Qualquer elemento da parte ordenada é **menor** que qualquer elemento da parte não-ordenada

Seleção Direta

- v = (44 , 55 , 12 , 42 , 94 , 18 , 06 , 67)

  - 44 55 12 42 94 18 06 67
  - | 06 55 12 42 94 18 44 67
  - 06 | 12 55 42 94 18 44 67
  - 06 12 | 18 42 94 55 44 67
  - 06 12 18 | 42 94 55 44 67
  - 06 12 18 42 | 44 55 94 67
  - 06 12 18 42 44 | 55 94 67
  - 06 12 18 42 44 55 | 67 94
  - 06 12 18 42 44 55 67 | 94

Seleção Direta

- No i-ésimo passo, o elemento com o menor valor entre v[i], ..., v[n-1] é selecionado e trocado com v[i]
- Como resultado, após i passos, os elementos v[0], ... , v[i-1] estão ordenados

Seleção Direta

```
void selection_sort(int v[], int n) {
  int i, j, aux;
  for (i = 0; i < n - 1; i++)
       for (j = i + 1; j < n; j++)
              if (v[j] < v[i]) {
                     aux = v[j];
                     v[j] = v[i];
                     v[i] = aux;
              }
}
```

Contagem de operações

```
void selection_sort(int v[], int n) {
  int i, j, aux;
  for (i = 0; i < n - 1; i++)
       for (j = i + 1; j < n; j++)
            if (v[j] < v[i]) {
                 aux = v[j];
                 v[j] = v[i];
                 v[i] = aux;
            }
}
```

Podemos contar somente as operações dominantes. Neste caso, é o teste (if) dentro do loop mais interno.

Contagem de operações

```
void selection_sort(int v[], int n) {
  int i, j, aux;
  for (i = 0; i < n - 1; i++)
      for (j = i + 1; j < n; j++)
          if (v[j] < v[i]) {
              aux = v[j];
              v[j] = v[i];
              v[i] = aux;
          }
}
```

Portanto, vamos contar o número de vezes que o loop mais interno é executado.

Contagem de operações

i=0	→	j varia de 1 a n-1	(n-1) iterações
i=1	→	j varia de 2 a n-1	(n-2) iterações
i=2	→	j varia de 3 a n-1	(n-3) iterações
...			
i=n-3	→	j varia de n-2 a n-1	(2) iterações
i=n-2	→	j varia de n-1 a n-1	(1) iteração

Contagem de operações

i=0	→	j varia de 1 a n-1	(n-1) iterações
i=1	→	j varia de 2 a n-1	(n-2) iterações
i=2	→	j varia de 3 a n-1	(n-3) iterações
...			
i=n-3	→	j varia de n-2 a n-1	(2) iterações
i=n-2	→	j varia de n-1 a n-1	(1) iteração

1 + 2 + ... + (n-2) + (n-1) = (1+n-1)*(n-1)/2 =

= n(n-1)/2 = $(n^2-n)/2$ comparações

Contagem de operações

i=0	→	j varia de 1 a n-1	(n-1) iterações
i=1	→	j varia de 2 a n-1	(n-2) iterações
i=2	→	j varia de 3 a n-1	(n-3) iterações
...			
i=n-3	→	j varia de n-2 a n-1	(2) iterações
i=n-2	→	j varia de n-1 a n-1	(1) iteração

$1 + 2 + ... + (n-2) + (n-1) = (1+n-1)*(n-1)/2 =$

$= n(n-1)/2 = (n^2-n)/2$ comparações

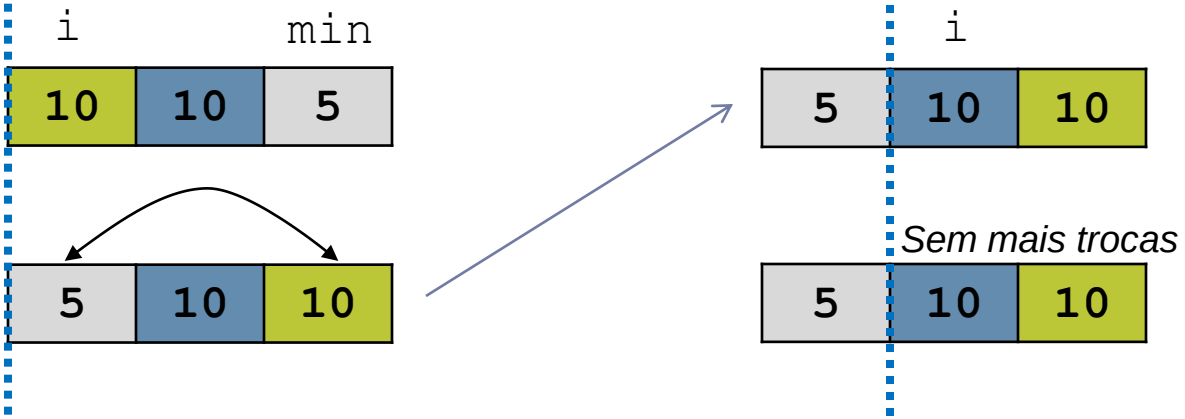
Complexidade $O(n^2)$

Seleção Direta

- Não existe diferença se a entrada está completamente ordenada ou desordenada (todas as comparações são sempre efetuadas)
  - Melhor caso: $O(n^2)$
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n^2)$
- Complexidade de espaço: $O(n)$
- Algoritmo de inserção é mais eficiente quando o vetor está quase ordenado

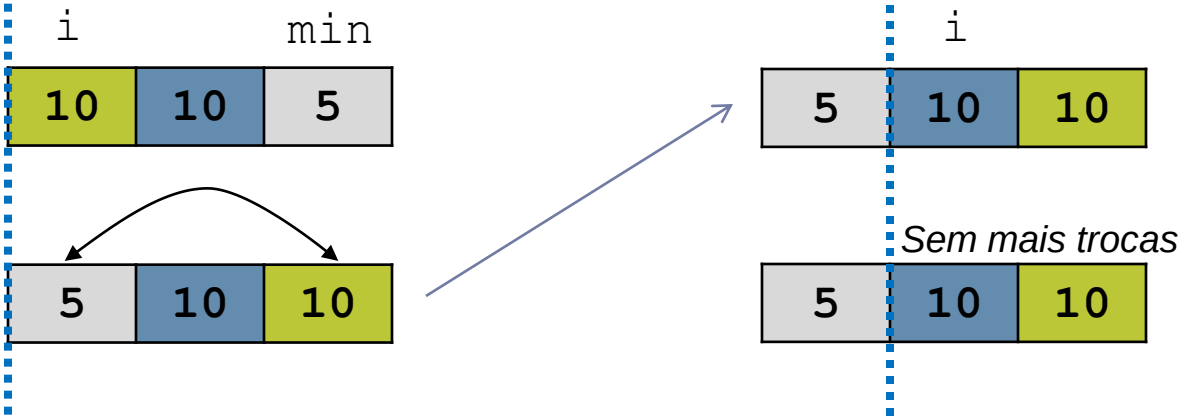
Algoritmo de seleção

O algoritmo de seleção é estável?



Algoritmo de seleção

O algoritmo de seleção é estável?



Não é estável

Seleção direta

- É possível ainda otimizar o algoritmo de seleção
- Note que, no loop mais interno, sempre que um elemento mínimo é encontrado, uma troca é efetuada
- Podemos realizar a troca somente ao final do loop mais interno, quando soubermos exatamente qual o menor elemento da parte não-ordenada

Seleção direta

```
void selection_sort_improved(int v[], int n) {
  int i, j, min, aux;
  for (i = 0; i < n - 1; i++) {
      min = i;
      for (j = i + 1; j < n; j++)
          if (v[j] < v[min])
              min = j;
      aux = v[min];
      v[min] = v[i];
      v[i] = aux;
  }
}
```

Troca somente uma vez por iteração do loop externo

Seleção direta

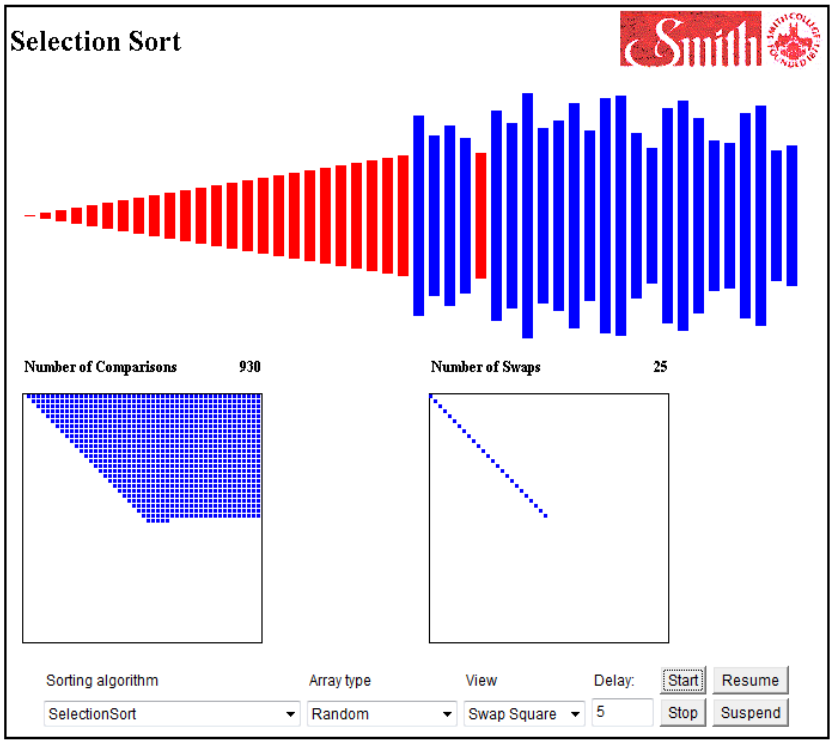
```
void selection_sort_improved(int v[], int n) {
  int i, j, min, aux;
  for (i = 0; i < n - 1; i++) {
      min = i;
      for (j = i + 1; j < n; j++)
          if (v[j] < v[min])
              min = j;
      aux = v[min];
      v[min] = v[i];
      v[i] = aux;
  }
}
```

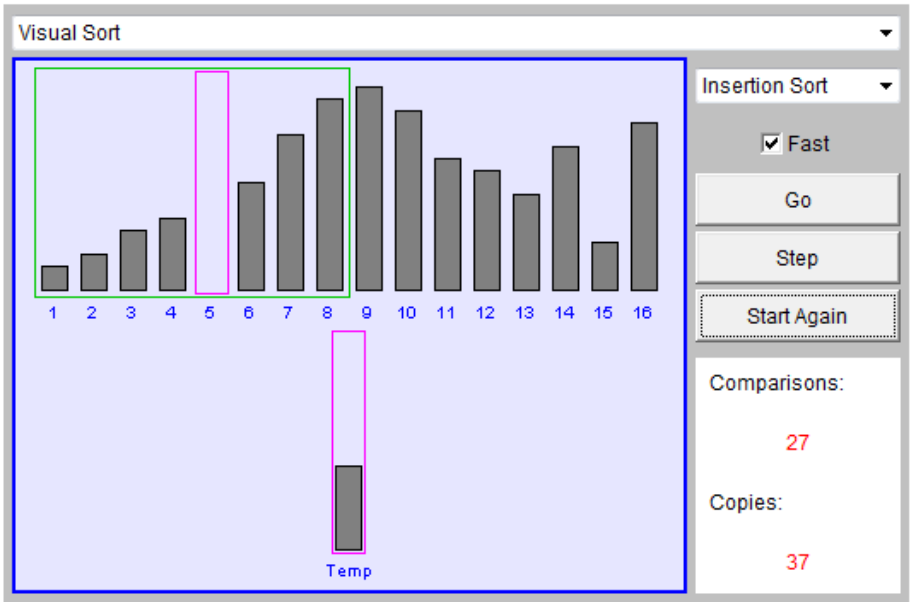
Troca somente uma vez por iteração do loop externo

A complexidade, considerando o número de comparações, ainda é $O(n^2)$

Visualização

- http://maven.smith.edu/~thiebaut/java/sort/demo.html
- http://math.hws.edu/TMCM/java/xSortLab/





Bubble-sort

Bubble-sort

- Método
  - Percorrer o vetor várias vezes
  - A cada iteração, comparar cada elemento com seu sucessor (vetor[i] com vetor[i+1]) e trocá-los de posição caso estejam na ordem incorreta

Bubble-sort: um passo

- v = (25 , 57 , 48 , 37 , 12 , 92 , 86 , 33)

v[0] com v[1] (25 com 57) não ocorre permutação

v[1] com v[2] (57 com 48) ocorre permutação

v = (25 , 48 , 57 , 37 , 12 , 92 , 86 , 33)

v[2] com v[3] (57 com 37) ocorre permutação

v = (25 , 48 , 37 , 57 , 12 , 92 , 86 , 33)

v[3] com v[4] (57 com 12) ocorre permutação

v = (25 , 48 , 37 , 12 , 57 , 92 , 86 , 33)

v[4] com v[5] (57 com 92) não ocorre permutação

v[5] com v[6] (92 com 86) ocorre permutação

v = (25 , 48 , 37 , 12 , 57 , 86 , 92 , 33)

v[6] com v[7] (92 com 33) ocorre permutação

v = (25 , 48 , 37 , 12 , 57 , 86 , 33 , 92)

Bubble-sort

- Depois do primeiro passo
  - v = (24 , 48 , 37 , 12 , 57 , 86 , 33 , 92)
  - Ainda não está completamente ordenado (garante-se somente que o maior elemento está na última posição)
- Para um vetor de n elementos, são necessárias n-1 iterações
- A cada iteração, os elementos vão sendo movidos na direção de suas posições corretas
  - Por que se chama método das bolhas (bubble)?

Bubble-sort

```
void bubble_sort(int v[], int n) {
  int i, j, aux;
  for (i = n - 1; i >= 0; i--)
       for (j = 0; j < i; j++)
              if (v[j] > v[j+1]) {
                     aux = v[j];
                     v[j] = v[j+1];
                     v[j+1] = aux;
              }
}
```

Complexidade

```
void bubble_sort(int v[], int n) {
  int i, j, aux;
  for (i = n - 1; i > 0; i--)
       for (j = 0; j < i; j++)
             if (v[j] > v[j+1]) {
                   aux = v[j];
                   v[j] = v[j+1];
                   v[j+1] = aux;
             }
}
```

Contagem de testes

Complexidade

i=n-1	→	j de 0 a n-2	→n-1 iterações
i=n-2	→	j de 0 a n-3	→n-2 iterações
...			
i=2	→	j de 0 a 1	→2 iterações
i=1	→	j de 0 a 0	→1 iteração

Complexidade

i=n-1	→	j de 0 a n-2	→n-1 iterações
i=n-2	→	j de 0 a n-3	→n-2 iterações
...			
i=2	→	j de 0 a 1	→2 iterações
i=1	→	j de 0 a 0	→1 iteração

- $1+2+...+n-1 = (1+n-1)(n-1)/2=$
  $= n(n-1)/2 = (n^2-n)/2$

Complexidade

i=n-1	→	j de 0 a n-2	→n-1 iterações
i=n-2	→	j de 0 a n-3	→n-2 iterações
...			
i=2	→	j de 0 a 1	→2 iterações
i=1	→	j de 0 a 0	→1 iteração

- $1+2+...+n-1 = (1+n-1)(n-1)/2 = n(n-1)/2 = (n^2-n)/2$

- Complexidade $O(n^2)$

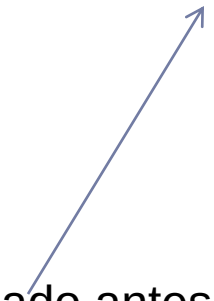
Bubble-sort

- Não existe diferença se a entrada está completamente ordenada ou desordenada (todas as comparações são sempre efetuadas)
  - Melhor caso: $O(n^2)$
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n^2)$
- Complexidade de espaço: $O(n)$

Bubble-sort

- Que melhorias podem ser feitas?

passo 0 (vetor original)	25 57 48 37 12 92 86 33
passo 1	25 48 37 12 57 86 33 92
passo 2	25 37 12 48 57 33 86 92
passo 3	25 12 37 48 33 57 86 92
passo 4	12 25 37 33 48 57 86 92
passo 5	12 25 33 37 48 57 86 92
passo 6	12 25 33 37 48 57 86 92
passo 7	12 25 33 37 48 57 86 92



Já está ordenado antes do fim. Deveríamos ser capazes de detectar esse momento e parar o algoritmo.

Bubble-sort aprimorado

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

Bubble-sort aprimorado

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

Complexidade no melhor caso?

Bubble-sort aprimorado

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

Complexidade no melhor caso é O(n)

Bubble-sort aprimorado

- Exercício: mostre que a complexidade no melhor caso é O(n)

- E a complexidade média?
  - Se considerarmos que o loop mais interno é executado, na média, metade das vezes que no pior caso, a complexidade no caso médio é $O(n^2)$
  - Exercício: prove essa afirmação

Estabilidade

- Bubble-sort é estável?

Estabilidade

- Bubble-sort é estável?

Sim, pois não há troca quando elementos consecutivos são iguais.

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