Ordenação de dados

Profª. Rose Yuri Shimizu

Roteiro

Ordenação de dados - importância

- Ordenação é organização
- Organização otimiza as buscas
  - Lógica de sequencialidade: previsibilidade
- Ordenação de itens (arquivos, estruturas)
  - A chave é a parte do item utilizada como parâmetro/controle de ordenação

Recomendações

- RobertSedgewickAlgorithmsinC,AddisonWesley,3nded.
- Algorithms,4thEdition-RobertSedgewickeKevinWayne
- https://brunoribas.com.br/apostila-eda/ordenacao-elementar.html
- https://www.youtube.com/@ProfBrunoRibas
- https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html
- https://github.com/bcribas/benchmark-ordenacao

Roteiro

Algoritmos de Ordenação - Características

1. Complexidade (espacial, temporal)
   - Quadráticos: simples e suficiente para arquivos pequenos
   - Linearítmicos: mais complexos (*overhead*) e eficientes para arquivos grandes

Algoritmos de Ordenação - Características

1. Complexidade (espacial, temporal)
2. Estabilidade
   - Mantém a posição relativa dos elementos
   - Não há saltos
   - 2 4 1 6 7 *1*
   - *1* 1 2 4 6 7 : não-estável
   - 1 *1* 2 4 6 7 : estável

Algoritmos de Ordenação - Características

1. Complexidade (espacial, temporal)
2. Estabilidade
   - Mantém a posição relativa dos elementos
3. Adaptatividade
   - Aproveita a ordenação existente

Algoritmos de Ordenação - Características

1. Complexidade (espacial, temporal)
2. Estabilidade
   - Mantém a posição relativa dos elementos
3. Adaptatividade
   - Aproveita a ordenação existente
4. Memória extra
   - In-place:
     - ★ Utiliza a própria estrutura
     - ★ Utiliza memória extra: pilha de execução, variáveis auxiliares
   - Não in-place:
     - ★ Utiliza mais uma estrutura
     - ★ Cópias

Algoritmos de Ordenação - Características

1. Complexidade (espacial, temporal)
2. Estabilidade
   - Mantém a posição relativa dos elementos
3. Adaptatividade
   - Aproveita a ordenação existente
4. Memória extra
   - In-place:
     - ★ Utiliza a própria estrutura
     - ★ Utiliza memória extra: pilha de execução, variáveis auxiliares
5. Localização
   - Interna: todos os dados cabem na memória principal
   - Externa: arquivo grande; é ordenado em pedaços (chunks) que caibam na memória principal

Algoritmos de Ordenação - Elementares x Eficientes

- Elementares: custos maiores, mais simples
- Eficientes: custos menores, mais complexos (estratégias)
- Analise as constantes da função custo e o tamanho da entrada

$$f1(n) = n^2$$
$$f2(n) = x * n + y$$

- Array x Listas encadeadas
  - Métodos elementares: lidam bem com qualquer implementação
  - Métodos mais eficientes:
    - Array: mais fácil manipulação pelo acesso direto
    - Estruturas encadeadas: árvores ordenadas

Roteiro

Roteiro

Algoritmos de Ordenação Elementares
Selection Sort - selecionar e posicionar

1. Selecionar o menor item
2. Posicionar: troque com o primeiro item
3. Selecionar o segundo menor item
4. Posicionar: troque com o segundo item
5. Repita para os n elementos do vetor

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 3 | 2 | 4 | 6 | 1 | 5 ]      índice do menor = i = 0?
          i     j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                               r
          0    1    2    3    4    5
v     [ 3 | 2 | 4 | 6 | 1 | 5 ]      índice do menor = j = 1
          i    j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 3 | 2 | 4 | 6 | 1 | 5 ]       índice do menor = 1?
          i           j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 3 | 2 | 4 | 6 | 1 | 5 ]       índice do menor = 1?
          i           j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 3 | 2 | 4 | 6 | 1 | 5 ]       índice do menor = 1?
          i                 j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 3 | 2 | 4 | 6 | 1 | 5 ]       índice do menor = j = 4
          i                 j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 3 | 2 | 4 | 6 | 1 | 5 ]       índice do menor = 4?
          i                                   j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                       r
          0   1   2   3   4   5
v     [ 3 | 2 | 4 | 6 | 1 | 5 ]      índice do menor = 4
          i                   j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Posicionar menor: troca(swap) v[i] $\leftrightarrow$ v[menor]

```
          l                                   r
          0     1     2     3     4     5
v     [ _3_ | 2 | 4 | 6 | _1_ | 5 ]       índice do menor = 4
          i                             j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Posicionar menor: troca(swap) v[i] $\leftrightarrow$ v[menor]

```
          l                                   r
          0     1     2     3     4     5
v     [ _1_ | 2 | 4 | 6 | _3_ | 5 ]       índice do menor = 4
          i                             j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 4 | 6 | 3 | 5 ]      índice do menor = i = 1?
                i     j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 4 | 6 | 3 | 5 ]      índice do menor = 1?
                i           j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 4 | 6 | 3 | 5 ]       índice do menor = 1?
              i                 j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 4 | 6 | 3 | 5 ]       índice do menor = 1?
                i                       j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                           r
          0    1    2    3    4    5
v     [ 1 | 2 | 4 | 6 | 3 | 5 ]      índice do menor = 1?
               i                   j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Posicionar menor: v[i] == v[menor] ? sem swap

```
          l                       r
          0    1    2    3    4    5
v     [ 1  |  2  |  4  |  6  |  3  |  5 ]      índice do menor = 1?
               i                   j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 4 | 6 | 3 | 5 ]      índice do menor = i = 2?
                    i     j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 4 | 6 | 3 | 5 ]       índice do menor = 2?
                    i           j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 4 | 6 | 3 | 5 ]       índice do menor = j = 4
                      i           j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v      [ 1 | 2 | 4 | 6 | 3 | 5 ]       índice do menor = 4?
                    i           j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                           r
          0    1    2    3    4    5
v     [ 1 | 2 | 4 | 6 | 3 | 5 ]      índice do menor = 4
                   i                   j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Posicionar menor: troca(swap) v[i] $\leftrightarrow$ v[menor]

```
          l                                   r
          0    1     2     3     4     5
v     [ 1 | 2 | _4_ | 6 | _3_ | 5 ]       índice do menor = 4
                    i                   j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Posicionar menor: troca(swap) v[i] $\leftrightarrow$ v[menor]

```
          l                                   r
          0    1     2     3     4     5
v     [ 1 | 2 | _3_ | 6 | _4_ | 5 ]       índice do menor = 4
                    i                 j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 3 | 6 | 4 | 5 ]       índice do menor = 3?
                        i     j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v      [ 1 | 2 | 3 | 6 | 4 | 5 ]       índice do menor = j = 4
                        i     j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 3 | 6 | 4 | 5 ]       índice do menor = 4?
                        i           j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                r
          0    1    2    3    4    5
v     [ 1  |  2  |  3  |  6  |  4  |  5 ]      índice do menor = 4?
                         i              j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Posicionar menor: troca(swap) v[i] $\leftrightarrow$ v[menor]

```
          l                                   r
          0    1    2     3      4     5
v     [ 1 | 2 | 3 | _6_ | _4_ | 5 ]       índice do menor = 4
                       i               j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Posicionar menor: troca(swap) v[i] $\leftrightarrow$ v[menor]

```
          l                                   r
          0    1    2     3      4     5
v     [ 1 | 2 | 3 | _4_ | _6_ | 5 ]       índice do menor = 4
                       i               j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 3 | 4 | 6 | 5 ]       índice do menor = i = 4?
                                  i     j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                                   r
          0     1     2     3     4     5
v     [ 1 | 2 | 3 | 4 | 6 | 5 ]       índice do menor = j = 5
                                 i     j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Selecionar v[j] < v[menor] ?

```
          l                             r
          0    1    2    3    4    5
v     [ 1 | 2 | 3 | 4 | 6 | 5 ]      índice do menor = 5?
                              i         j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Posicionar menor: troca(swap) v[i] $\leftrightarrow$ v[menor]

```
          l                                   r
          0    1    2    3    4    5
v     [ 1  |  2  |  3  |  4  | _6_ | _5_ ]        índice do menor = 5
                              i          j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Posicionar menor: troca(swap) v[i] $\leftrightarrow$ v[menor]

```
          l                                   r
          0    1    2    3    4    5
v     [ 1 | 2 | 3 | 4 | _5_ | _6_ ]      índice do menor = 5
                              i     j
```

Algoritmos de Ordenação Elementares
Selection Sort

- Terminou? Vetor ordenado.

```
          l                             r
          0    1    2    3    4    5
v     [ 1 | 2 | 3 | 4 | 5 | 6 ]      índice do menor = i = 5
                                 i    j
```

Algoritmos de Ordenação Elementares
Selection Sort

```
1  void selection_sort(int v[], int l, int r){
2      int menor;
3      for(int i=l; i<r; i++){
4          menor = i;
5
6          for(int j=i+1; j<=r; j++)
7              if(v[j] < v[menor])
8                  menor = j;
9
10         if(i != menor)
11             exch(v[i], v[menor])
12     }
13 }
14
```

Algoritmos de Ordenação Elementares
Selection Sort

```
1  void selection_sort(int v[], int l, int r){
2      int menor;
3      //n
4      for(int i=l; i<r; i++){
5          menor = i;
6
7          //(n-1), (n-2), (n-3), .. , 0
8          //PA ((n+0)n)/2 = (n^2)/2
9          for(int j=i+1; j<=r; j++)
10             if(v[j] < v[menor])
11                 menor = j;
12
13         if(i != menor)
14             exch(v[i], v[menor]) //n
15     }
16     //f(n) = (n^2)/2 + n
17 }
18
```

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
- Adaptatividade?
- Estabilidade?
- *In-place*?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
- Estabilidade?
- *In-place*?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
  - Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!
- Estabilidade?
- *In-place*?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
  - Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!
  - Não, portanto, não é adaptativo.
- Estabilidade?
- *In-place*?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
  - Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!
  - Não, portanto, não é adaptativo.
- Estabilidade?
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
- *In-place*?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
  - Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!
  - Não, portanto, não é adaptativo.
- Estabilidade?
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
- *In-place*?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
  - Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!
  - Não, portanto, não é adaptativo.
- Estabilidade?
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ★ 1 3 4' 4
- *In-place*?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
  - Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!
  - Não, portanto, não é adaptativo.
- Estabilidade?
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ★ 1 3 4' 4
  - Não mantém a ordem: não estável.
- *In-place*?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
  - Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!
  - Não, portanto, não é adaptativo.
- Estabilidade?
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ⋆ 1 3 4' 4
  - Não mantém a ordem: não estável.
- *In-place*?
  - Utiliza memória extra significativa?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
  - Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!
  - Não, portanto, não é adaptativo.
- Estabilidade?
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ⋆ 1 3 4' 4
  - Não mantém a ordem: não estável.
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?

Algoritmos de Ordenação Elementares
Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
- Adaptatividade?
  - Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!
  - Não, portanto, não é adaptativo.
- Estabilidade?
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ⋆ 1 3 4' 4
  - Não mantém a ordem: não estável.
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?
  - Não, portanto, é *in-place*.

Algoritmos de Ordenação Elementares
Selection Sort

- Selection Sort estável??

- Selection Sort com listas encadeadas??

Algoritmos de Ordenação Elementares
Selection Sort

- Selection Sort estável??
  - Não realizar o swap

- Selection Sort com listas encadeadas??

Algoritmos de Ordenação Elementares
Selection Sort

- Selection Sort estável??
  - Não realizar o swap
  - Ideia: "abrir" um espaço na posição, "empurrando" os itens para frente
  - Boa solução?
- Selection Sort com listas encadeadas??

Algoritmos de Ordenação Elementares
Selection Sort

- Selection Sort estável??
  - Não realizar o swap
  - Ideia: "abrir" um espaço na posição, "empurrando" os itens para frente
  - Boa solução?
- Selection Sort com listas encadeadas??
  - Percorre a lista sequencialmente

Roteiro

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

1. Do início, flutuar o item
2. Ao achar uma "bolha" maior, esta passa a flutuar
3. No fim, o maior (ou menor) está no topo: topo–;
4. Volte para o item 1

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                           r
          0    1    2    3    4    5
v     [ 3 | 2 | 4 | 6 | 1 | 5 ]
          j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ? Flutua (swap)

```
          l                           r
          0    1    2    3    4    5
v     [ 2 | 3 | 4 | 6 | 1 | 5 ]
          j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                                 r
          0     1     2     3     4     5
v     [ 2 | 3 | 4 | 6 | 1 | 5 ]
              j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                                 r
          0     1     2     3     4     5
v      [ 2 | 3 | 4 | 6 | 1 | 5 ]
                    j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                                 r
          0     1     2     3     4     5
v      [ 2 | 3 | 4 | 6 | 1 | 5 ]
                        j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ? Flutua (swap)

```
          l                                 r
          0     1     2     3     4     5
v     [ 2 | 3 | 4 | 1 | 6 | 5 ]
                        j     j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                                 r
          0     1     2     3     4     5
v      [ 2 | 3 | 4 | 1 | 6 | 5 ]
                              j    j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ? Flutua (swap)
- A cada flutuação, um elemento é posicionado corretamente (topo)

```
          l                                r
          0    1    2    3    4    5
v     [ 2 | 3 | 4 | 1 | 5 | _6_ ]
                        j    j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                       r
          0    1    2    3    4    5
v     [ 2 | 3 | 4 | 1 | 5 | 6 ]
          j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                       r
          0    1    2    3    4    5
v     [ 2 | 3 | 4 | 1 | 5 | 6 ]
               j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                       r
          0    1    2    3    4    5
v     [ 2 | 3 | 4 | 1 | 5 | 6 ]
                   j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ? Flutua (swap)

```
          l                       r
          0    1    2    3    4    5
v     [ 2 | 3 | 1 | 4 | 5 | 6 ]
                   j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                       r
          0    1    2    3    4    5
v     [ 2 | 3 | 1 | 4 | 5 | 6 ]
                      j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                       r
          0    1    2    3    4    5
v     [ 2 | 3 | 1 | 4 | _5_ | 6 ]
                      j    j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                   r
          0     1     2     3     4     5
v      [ 2 | 3 | 1 | 4 | 5 | 6 ]
          j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                   r
          0    1    2    3    4    5
v     [ 2 | 3 | 1 | 4 | 5 | 6 ]
               j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ? Flutua (swap)

```
          l                   r
          0    1    2    3    4    5
v     [ 2 | 1 | 3 | 4 | 5 | 6 ]
               j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                   r
          0    1    2    3    4    5
v     [ 2 | 1 | 3 | 4 | 5 | 6 ]
                   j   j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l                   r
          0    1    2     3     4     5
v     [ 2 | 1 | 3 | _4_ | 5 | 6 ]
                   j    j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l           r
          0   1   2   3   4   5
v     [ 2 | 1 | 3 | 4 | 5 | 6 ]
          j  j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ? Flutua (swap)

```
          l           r
          0   1   2   3   4   5
v     [ 1 | 2 | 3 | 4 | 5 | 6 ]
          j  j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l           r
          0    1      2      3    4    5
v      [ 1  |  2  |  _3_  |  4  |  5  |  6 ]
               j     j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l   r
          0   1    2   3   4   5
v     [ 1 | 2 | 3 | 4 | 5 | 6 ]
          j  j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l     r
          0     1     2     3     4     5
v     [ 1 | _2_ | 3 | 4 | 5 | 6 ]
          j    j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes v[j] > v[j+1] ?

```
          l r
           0     1    2    3    4    5
v      [ _1_ | 2 | 3 | 4 | 5 | 6 ]
           j    j+1
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1  void bubble_sort(int v[], int l, int r){
2
3      for(; r>l; r--) {
4          for(int j=l; j<r; j++) {
5              if(v[j] > v[j+1]) {
6                  exch(v[j], v[j+1])
7              }
8          }
9      }
10
11 }
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1  void bubble_sort(int v[], int l, int r){
2
3      //n
4      for(; r>l; r--) {
5
6          //(n-1), (n-2), (n-3), .. , 0
7          //PA ((n+0)n)/2 = (n^2)/2
8          for(int j=l; j<r; j++) {
9
10             if(v[j] > v[j+1]) {
11
12                 //(n-1), (n-2), (n-3), .. , 0
13                 //PA ((n+0)n)/2 = (n^2)/2
14                 exch(v[j], v[j+1])
15             }
16         }
17     }
18     //f(n) = (n^2)/2 + (n^2)/2
19 }
```

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
- Adaptatividade?
- Estabilidade?
- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
- Estabilidade?
- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
  - Ordenação diminui processamento?
- Estabilidade?
- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.
- Estabilidade?
- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.
- Estabilidade?
  - 2 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.
- Estabilidade?
  - 2 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.
- Estabilidade?
  - 2 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ⋆ 2 3 4 1 4'
- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.
- Estabilidade?
  - 2 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ⋆ 2 3 4 1 4'
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.
- Estabilidade?
  - 2 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ⋆ 2 3 4 1 4'
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?
  - Utiliza memória extra significativa?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.
- Estabilidade?
  - 2 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ⋆ 2 3 4 1 4'
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.
- Estabilidade?
  - 2 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ⋆ 2 3 4 1 4'
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?
  - Não, portanto, é *in-place*.

Algoritmos de Ordenação Elementares
Bubble Sort

```
1  void bubble_sort(int v[], int l, int r){
2      int swap = 1;
3      for(; r>l && swap; r--) {
4          swap = 0;
5          for(int j=l; j<r; j++) {
6              if(v[j] > v[j+1]) {
7                  exch(v[j], v[j+1])
8                  swap = 1;
9              }
10         }
11     }
12 }
```

Algoritmos de Ordenação Elementares
Bubble Sort

- Selection Sort x Bubble sort?
- Bubble Sort com listas encadeadas??
- Otimização?

Algoritmos de Ordenação Elementares
Bubble Sort

- Selection Sort x Bubble sort?
  - Bubble sort é pior que o selection
  - Sempre?
  - Teste com as entradas “16-aleatorio” e “17-quaseordenado” do conjunto de testes
- Bubble Sort com listas encadeadas??
- Otimização?

Algoritmos de Ordenação Elementares
Bubble Sort

- Selection Sort x Bubble sort?
  - Bubble sort é pior que o selection
  - Sempre?
  - Teste com as entradas “16-aleatorio” e “17-quaseordenado” do conjunto de testes
- Bubble Sort com listas encadeadas??
  - Percorre a lista sequencialmente
- Otimização?

Algoritmos de Ordenação Elementares
Bubble Sort

- Selection Sort x Bubble sort?
  - Bubble sort é pior que o selection
  - Sempre?
  - Teste com as entradas "16-aleatorio" e "17-quaseordenado" do conjunto de testes
- Bubble Sort com listas encadeadas??
  - Percorre a lista sequencialmente
- Otimização?
  - Shaker sort: consiste em realizar uma iteração para colocar o menor elemento em cima e na volta colocar o maior elemento no fundo

Roteiro

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

1. Inserir cada elemento na posição correta em relação aos seus antecessores
2. Comparação item a item com seus antecessores

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                       r
          0    1    2    3    4    5
v     [ 4  |  2  |  3  |  6  |  1  |  5 ]
               i
         j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Insere (swap)

```
          l                           r
          0    1    2    3    4    5
v     [ 2  |  4  |  3  |  6  |  1  |  5 ]
               i
         j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                                 r
          0    1    2    3    4    5
v     [ 2 | 4 | 3 | 6 | 1 | 5 ]
                   i
             j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Insere (swap)

```
          l                                 r
          0    1    2    3    4    5
v     [ 2 | 3 | 4 | 6 | 1 | 5 ]
                   i
              j-1  j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                           r
          0    1    2    3    4    5
v     [ 2  |  3  |  4  |  6  |  1  |  5 ]
                    i
         j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Não insere (sem swap)

```
          l                                 r
          0    1    2    3    4    5
v     [ 2  |  3  |  4  |  6  |  1  |  5 ]
                    i
         j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                                 r
          0     1     2     3     4     5
v     [ 2 | 3 | 4 | 6 | 1 | 5 ]
                        i
                  j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Não insere (sem swap)

```
          l                                 r
          0     1     2     3     4     5
v     [ 2 | 3 | 4 | 6 | 1 | 5 ]
                        i
                  j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                                 r
          0     1     2     3     4     5
v     [ 2 | 3 | 4 | 6 | 1 | 5 ]
                          i
                    j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Insere (swap)

```
          l                                 r
          0     1     2     3     4     5
v     [ 2 | 3 | 4 | 1 | 6 | 5 ]
                              i
                        j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                           r
          0    1    2    3    4    5
v     [ 2 | 3 | 4 | 1 | 6 | 5 ]
                              i
                   j-1  j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Insere (swap)

```
          l                       r
          0    1    2    3    4    5
v     [ 2 | 3 | 1 | 4 | 6 | 5 ]
                          i
               j-1  j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                       r
          0    1    2    3    4    5
v     [ 2 | 3 | 1 | 4 | 6 | 5 ]
                             i
             j-1  j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Insere (swap)

```
          l                                 r
          0    1    2    3    4    5
v     [ 2 | 1 | 3 | 4 | 6 | 5 ]
                              i
             j-1  j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                       r
          0   1   2   3   4   5
v     [ 2 | 1 | 3 | 4 | 6 | 5 ]
                        i
         j-1  j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Insere (swap)

```
          l                             r
          0    1    2    3    4    5
v     [ 1  |  2  |  3  |  4  |  6  |  5 ]
                                 i

         j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                                 r
          0    1    2    3    4    5
v     [ 1  |  2  |  3  |  4  |  6  |  5 ]
                                    i
                              j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Insere (swap)

```
          l                                 r
          0    1    2    3    4    5
v     [ 1  |  2  |  3  |  4  |  5  |  6 ]
                                    i
                              j-1   j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ?

```
          l                             r
          0    1    2    3    4    5
v     [ 1 | 2 | 3 | 4 | 5 | 6 ]
                                   i
                        j-1  j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- Comparar com antecessor v[j] < v[j-1] ? Não insere (sem swap)

```
          l                                r
          0    1    2    3    4    5
v     [ 1 | 2 | 3 | 4 | 5 | 6 ]
                                    i
                         j-1  j
```

Algoritmos de Ordenação Elementares
Insertion Sort

```
1  void insertion_sort(int v[], int l, int r)
2  {
3      PERCORRER ARRAY A PARTIR DO SEGUNDO ELEMENTO
4
5
6
7
8
9
10 }
11
```

Algoritmos de Ordenação Elementares
Insertion Sort

```
1  void insertion_sort(int v[], int l, int r)
2  {
3      for(int i=l+1; i<=r; i++)
4      {
5          PROCURANDO ANTECESSORES MENORES QUE V[J]
6
7
8
9      }
10 }
11
```

Algoritmos de Ordenação Elementares
Insertion Sort

```
1  void insertion_sort(int v[], int l, int r)
2  {
3      for(int i=l+1; i<=r; i++)
4      {
5          for(int j=i; j>l && v[j]<v[j-1]; j--)
6          {
7              INSERINDO NA POSICAO
8          }
9      }
10 }
11
```

Algoritmos de Ordenação Elementares
Insertion Sort

```
1  void insertion_sort(int v[], int l, int r)
2  {
3      for(int i=l+1; i<=r; i++)
4      {
5          for(int j=i; j>l && v[j]<v[j-1]; j--)
6          {
7              exch(v[j], v[j-1]);
8          }
9      }
10 }
11
```

Algoritmos de Ordenação Elementares
Insertion Sort

- Complexidade assintótica?
  - Pior caso $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas
    - ★ Não é indicado para grandes entradas totalmente desordenadas ou invertida
    - ★ Desempenho do Bubble Sort
    - ★ Envolve trocas com somente com os adjacentes

```
1  void insertion_sort(int v[], int l, int r)
2  {
3      for(int i=l+1; i<=r; i++)
4      {
5          //1 2 3 ... (n-1)
6          //PA ((n-1+1)n)/2 = (n^2)/2
7          for(int j=i; j>l && v[j]<v[j-1]; j--)
8          {
9              exch(v[j], v[j-1]);
10         }
11     }
12 }
13
```

Algoritmos de Ordenação Elementares
Insertion Sort

- Complexidade assintótica: $O(n^2)$
  - Pior caso $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas

$$
\begin{aligned}
f(n) &\approx f(n-1)+n-1 \\
&\approx f(n-2)+(n-1)-1+n-1 \\
&\approx f(n-2)+(n-1)+n-2 \\
&\approx f(n-3)+(n-2)-1+(n-1)+n-2 \\
&\approx f(n-3)+(n-2)+(n-1)+n-3 \\
&\approx f(n-i)+(n-i+1)+(n-i+2)+\ldots+(n-2)+(n-1)+n-i \\
&\approx \ldots \\
&\approx f(0)+1+2+\ldots+(n-2)+(n-1)+n-n \\
&\approx \frac{(1+(n-1))*n}{2} \\
&\approx \frac{n^2}{2}
\end{aligned}
$$

Algoritmos de Ordenação Elementares
Insertion Sort

- Complexidade assintótica?
  - Pior caso $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas
  - Médio aprox. $\frac{N^2}{4}$ comparações e $\frac{N^2}{4}$ trocas
  - Melhor caso: $O(N)$ (quando?)

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
    1. **1 2** 3 4 6 5

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
    1. **1 2** 3 4 6 5
    2. 1 **2 3** 4 6 5

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
    1. **1 2** 3 4 6 5
    2. 1 **2 3** 4 6 5
    3. 1 2 **3 4** 6 5

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
    1. **1 2** 3 4 6 5
    2. 1 **2 3** 4 6 5
    3. 1 2 **3 4** 6 5
    4. 1 2 3 **4 6** 5

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
    1. **1 2** 3 4 6 5
    2. 1 **2 3** 4 6 5
    3. 1 2 **3 4** 6 5
    4. 1 2 3 **4 6** 5
    5. 1 2 3 4 **6 5**

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
    1. **1 2** 3 4 6 5
    2. 1 **2 3** 4 6 5
    3. 1 2 **3 4** 6 5
    4. 1 2 3 **4 6** 5
    5. 1 2 3 4 **6 5**
    6. 1 2 3 4 **5 6**

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
    1. **1 2** 3 4 6 5
    2. 1 **2 3** 4 6 5
    3. 1 2 **3 4** 6 5
    4. 1 2 3 **4 6** 5
    5. 1 2 3 4 **6 5**
    6. 1 2 3 4 **5 6**
  - Sim, portanto, é adaptativo.

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1
    4. 2 2' **3 1**

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1
    4. 2 2' **3 1**
    5. 2 **2' 1** 3

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1
    4. 2 2' **3 1**
    5. 2 **2' 1** 3
    6. **2 1** 2' 3

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1
    4. 2 2' **3 1**
    5. 2 **2' 1** 3
    6. **2 1** 2' 3
    7. 1 2 2' 3

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1
    4. 2 2' **3 1**
    5. 2 **2' 1** 3
    6. **2 1** 2' 3
    7. 1 2 2' 3
  - Mantém a ordem (não trocar os iguais): estável.

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1
    4. 2 2' **3 1**
    5. 2 **2' 1** 3
    6. **2 1** 2' 3
    7. 1 2 2' 3
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1
    4. 2 2' **3 1**
    5. 2 **2' 1** 3
    6. **2 1** 2' 3
    7. 1 2 2' 3
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?
  - Utiliza memória extra significativa?

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1
    4. 2 2' **3 1**
    5. 2 **2' 1** 3
    6. **2 1** 2' 3
    7. 1 2 2' 3
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - 3 2 2' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    1. **2 3** 2' 1
    2. 2 **2' 3** 1
    3. **2 2'** 3 1
    4. 2 2' **3 1**
    5. 2 **2' 1** 3
    6. **2 1** 2' 3
    7. 1 2 2' 3
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?
  - Não, portanto, é *in-place*.

Algoritmos de Ordenação Elementares
Insertion Sort x Bubble sort

- Bubble:
  - o posicionamento de um item não garante a ordenação dos outros elementos
    - garante que os elementos à esquerda sejam menores e à direita maiores
    - não necessariamente ordenados a cada passagem
- Insertion:
  - o posicionamento de um item garante a ordenação dos elementos a sua esquerda

Algoritmos de Ordenação Elementares
Insertion Sort x Selection sort

- Selection:
  - Relativo a uma posição atual:
    - itens à esquerda $\rightarrow$ ordenados e na posição final
- Insertion:
  - Relativo a uma posição atual:
    - itens à esquerda $\rightarrow$ ordenados mas,
    - podem não estar posição final
    - podem ter que ser movidos para abrir espaço para itens menores
  - Tempo de execução depende da ordenação inicial
    - É adaptativo
    - Quanto mais ordenado, mais rápido
    - O tempo tende a linear quanto mais ordenado
    - Selection, continua quadrático

Algoritmos de Ordenação Elementares
Shell Sort

- Extensão do algoritmo de ordenação Insertion Sort
- Ideia:
  - Ordenação parcial a cada passagem
  - Posteriormente, eficientemente, ordenados pelo Insertion Sort
- Diminui o número de movimentações
- Troca de itens que estão distantes um do outro
  - Separados a h distância
  - São rearranjados, resultando uma sequencia ordenada para a distância h (h-ordenada)
  - Quando h=1, corresponde ao Insertion Sort
  - A dificuldade é determinar o valor de h
    - Donald Knuth (cientista da computação): recomenda algo em torno de 1/3 da entrada
    - sequencias múltiplas de 2 não performam bem
    - 1 2 4 8 16 32 64 128 256...
    - itens em posições pares não confrontam itens em posições ímpares até o fim do processo e, vice e versa
- Implementação é muito simples, similar ao algoritmo de inserção

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 1
h = 3*h+1 → alternar pares e ímpares
h = 1, 4, 13, 40, 121, 364, 1093, ...

r = 16 → 16/3 ~ 5            terça parte do total
h = 1 < 5? (3*1+1) : 1
h = 4 < 5? (3*4+1) : 4
h = 13 < 5? (3*13+1) : 13    máximo h: imediatamente maior
                                       que a terça parte
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  7 |  4 |  9 | 12 |  6 | 11 | 15 |  5 | 16 | 13 |  3 | 10 |  2 |  8 |  1 | 14 ]
    j-h                                                                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  7 |  4 |  9 | 12 |  6 | 11 | 15 |  5 | 16 | 13 |  3 | 10 |  2 |  8 |  1 | 14 ]
         j-h                                                                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  7 |  1 |  9 | 12 |  6 | 11 | 15 |  5 | 16 | 13 |  3 | 10 |  2 |  8 |  4 | 14 ]
         j-h                                                                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13
swap → antecessores? não, continua...
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  7 |  1 |  9 | 10 |  6 | 11 | 15 |  5 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
              j-h                                                              j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  7 |  1 |  9 | 10 |  6 | 11 | 15 |  5 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
    j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 | 10 |  7 | 11 | 15 |  5 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
    j-h                   j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? não, continua
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 | 10 |  7 | 11 | 15 |  5 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
         j-h                   j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 | 10 |  7 | 11 | 15 |  5 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
               j-h                   j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 | 10 |  7 | 11 | 15 |  5 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
                     j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 |  5 |  7 | 11 | 15 | 10 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
                      j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? não, continua
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 |  5 |  7 | 11 | 15 | 10 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
                          j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 |  5 |  7 | 11 | 15 | 10 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
                              j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 |  5 |  7 | 11 | 15 | 10 | 16 | 13 |  3 | 12 |  2 |  8 |  4 | 14 ]
                                     j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 |  5 |  7 | 11 |  3 | 10 | 16 | 13 | 15 | 12 |  2 |  8 |  4 | 14 ]
                                    j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? sim, procura
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  9 |  5 |  7 | 11 |  3 | 10 | 16 | 13 | 15 | 12 |  2 |  8 |  4 | 14 ]
              j-h                       j                  i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? não, continua
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  3 |  5 |  7 | 11 |  9 | 10 | 16 | 13 | 15 | 12 |  2 |  8 |  4 | 14 ]
              j-h                   j                   i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  3 |  5 |  7 | 11 |  9 | 10 | 16 | 13 | 15 | 12 |  2 |  8 |  4 | 14 ]
                                       j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  3 |  5 |  7 | 11 |  9 | 10 | 16 | 13 | 15 | 12 |  2 |  8 |  4 | 14 ]
                                                 j-h                 j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  3 |  5 |  7 | 11 |  9 | 10 |  2 | 13 | 15 | 12 | 16 |  8 |  4 | 14 ]
                                          j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? sim, procura
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  3 |  5 |  7 | 11 |  9 | 10 |  2 | 13 | 15 | 12 | 16 |  8 |  4 | 14 ]
                         j-h                   j                   i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  3 |  5 |  2 | 11 |  9 | 10 |  7 | 13 | 15 | 12 | 16 |  8 |  4 | 14 ]
                          j-h                   j                   i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? sim, procura
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  3 |  5 |  2 | 11 |  9 | 10 |  7 | 13 | 15 | 12 | 16 |  8 |  4 | 14 ]
                         j-h                   j                   i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  6 |  1 |  3 |  5 |  2 | 11 |  9 | 10 |  7 | 13 | 15 | 12 | 16 |  8 |  4 | 14 ]
    j-h                   j                                       i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 | 11 |  9 | 10 |  7 | 13 | 15 | 12 | 16 |  8 |  4 | 14 ]
    j-h                   j                                       i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? não, continua
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 | 11 |  9 | 10 |  7 | 13 | 15 | 12 | 16 |  8 |  4 | 14 ]
                                                 j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 | 11 |  9 | 10 |  7 |  8 | 15 | 12 | 16 | 13 |  4 | 14 ]
                                                 j-h                 j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? sim, procura
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 | 11 |  9 | 10 |  7 |  8 | 15 | 12 | 16 | 13 |  4 | 14 ]
                             j-h                  j                       i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  9 | 10 |  7 | 11 | 15 | 12 | 16 | 13 |  4 | 14 ]
                              j-h                  j                   i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? sim, procura
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  9 | 10 |  7 | 11 | 15 | 12 | 16 | 13 |  4 | 14 ]
                              j-h                  j                   i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  9 | 10 |  7 | 11 | 15 | 12 | 16 | 13 |  4 | 14 ]
          j-h                   j                                      i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  9 | 10 |  7 | 11 | 15 | 12 | 16 | 13 |  4 | 14 ]
                                                      j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  9 | 10 |  7 | 11 |  4 | 12 | 16 | 13 | 15 | 14 ]
                                                      j-h                  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? sim, procura
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  9 | 10 |  7 | 11 |  4 | 12 | 16 | 13 | 15 | 14 ]
                                    j-h                  j                   i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores?
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  4 | 10 |  7 | 11 |  9 | 12 | 16 | 13 | 15 | 14 ]
                                    j-h                  j                   i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4
swap → antecessores? sim, procura
      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  4 | 10 |  7 | 11 |  9 | 12 | 16 | 13 | 15 | 14 ]
               j-h                   j                                        i
```

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  4 | 10 |  7 | 11 |  9 | 12 | 16 | 13 | 15 | 14 ]
               j-h                   j                                      i
```

Algoritmos de Ordenação Elementares
Shell Sort

h = 13/3 ~ 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v [	2	1	3	5	6	8	4	10	7	11	9	12	16	13	15	14]
												j-h				j=i

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4/3 ~ 1   : Insertion sort

      0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
v  [  2 |  1 |  3 |  5 |  6 |  8 |  4 | 10 |  7 | 11 |  9 | 12 | 16 | 13 | 15 | 14 ]
    j-h  j=i
```

Algoritmos de Ordenação Elementares
Shell Sort

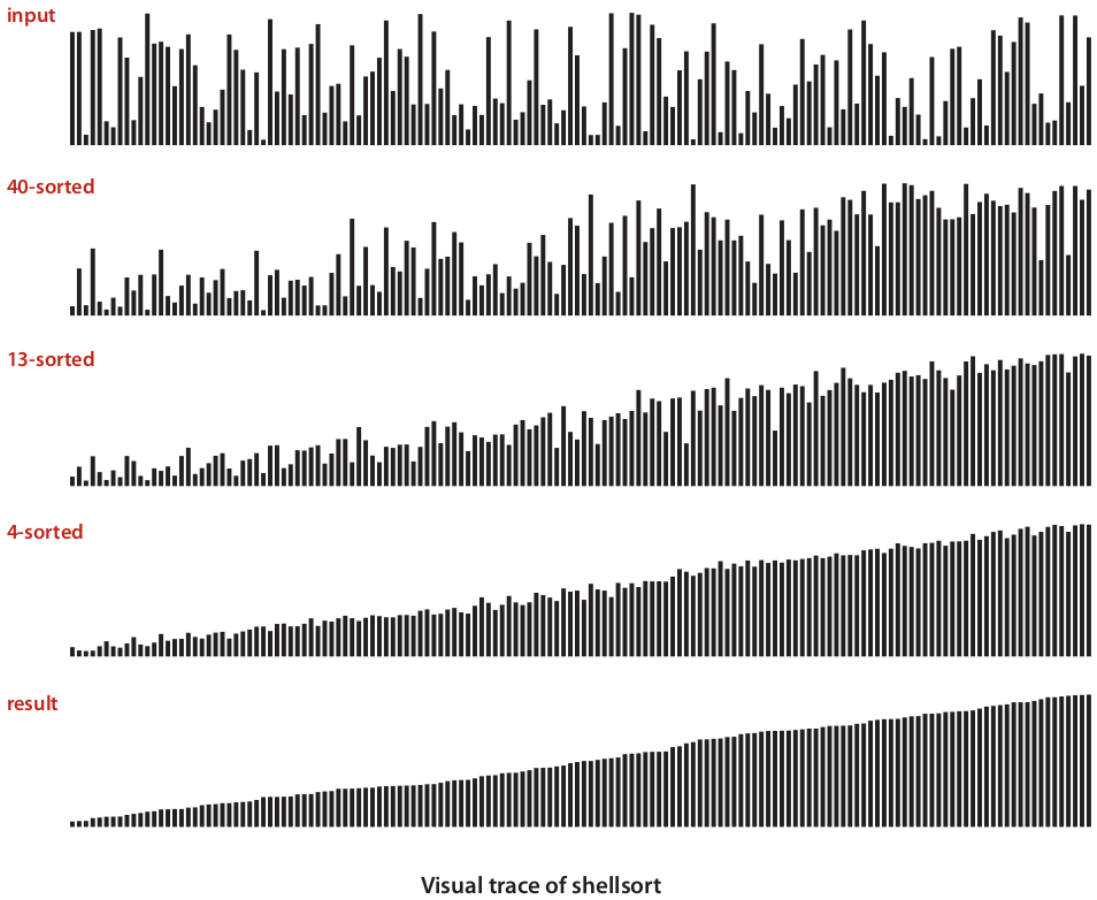


Figura: fonte: Algorithms – 4 edição, Robert Sedgewick e Kevin Wayne

Algoritmos de Ordenação Elementares
Insertion Sort

```
1  void shell_sort(int v[], int l, int r)
2  {
3      int h = 1;
4      while(h < (r-l+1)/3) h = 3*h+1;
5
6      while(h>=1){
7          for(int i=l+h; i<=r; i++)
8          {
9              for(int j=i; j>=l+h && v[j]<v[j-h]; j-=h)
10             {
11                 exch(v[j], v[j-h])
12             }
13         }
14         h = h/3;
15     }
16 }
17
```

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas
    - ⋆ Conta com a possibilidade de acertar (ou aproximar) a posição correta

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas
    - ⋆ Conta com a possibilidade de acertar (ou aproximar) a posição correta
  - Empiricamente, observou-se sua eficiência em diversos casos

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas
    - ⋆ Conta com a possibilidade de acertar (ou aproximar) a posição correta
  - Empiricamente, observou-se sua eficiência em diversos casos
  - No pior caso, shellsort não é necessariamente quadrático (Sedgewick)

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas
    - ⋆ Conta com a possibilidade de acertar (ou aproximar) a posição correta
  - Empiricamente, observou-se sua eficiência em diversos casos
  - No pior caso, shellsort não é necessariamente quadrático (Sedgewick)
    - ⋆ As comparações são proporcionais a $N^{\frac{3}{2}}$

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas
    - ⋆ Conta com a possibilidade de acertar (ou aproximar) a posição correta
  - Empiricamente, observou-se sua eficiência em diversos casos
  - No pior caso, shellsort não é necessariamente quadrático (Sedgewick)
    - ⋆ As comparações são proporcionais a $N^{\frac{3}{2}}$
    - ⋆ Pior caso com pior sequencia de intervalos h: $O(n^2)$

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas
    - ⋆ Conta com a possibilidade de acertar (ou aproximar) a posição correta
  - Empiricamente, observou-se sua eficiência em diversos casos
  - No pior caso, shellsort não é necessariamente quadrático (Sedgewick)
    - ⋆ As comparações são proporcionais a $N^{\frac{3}{2}}$
    - ⋆ Pior caso com pior sequencia de intervalos h: $O(n^2)$
    - ⋆ Melhor caso com pior sequencia de intervalos h: $O(nlog^2 n)$ (Pratt, Vaughan Ronald (1979). Shellsort and Sorting Networks - Outstanding Dissertations in the Computer Sciences)

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas
    - ⋆ Conta com a possibilidade de acertar (ou aproximar) a posição correta
  - Empiricamente, observou-se sua eficiência em diversos casos
  - No pior caso, shellsort não é necessariamente quadrático (Sedgewick)
    - ⋆ As comparações são proporcionais a $N^{\frac{3}{2}}$
    - ⋆ Pior caso com pior sequencia de intervalos h: $O(n^2)$
    - ⋆ Melhor caso com pior sequencia de intervalos h: $O(nlog^2 n)$ (Pratt, Vaughan Ronald (1979). Shellsort and Sorting Networks - Outstanding Dissertations in the Computer Sciences)
  - Melhor caso com uma boa sequencia de intervalos h: $O(nlogn)$

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas
    - ⋆ Conta com a possibilidade de acertar (ou aproximar) a posição correta
  - Empiricamente, observou-se sua eficiência em diversos casos
  - No pior caso, shellsort não é necessariamente quadrático (Sedgewick)
    - ⋆ As comparações são proporcionais a $N^{\frac{3}{2}}$
    - ⋆ Pior caso com pior sequencia de intervalos h: $O(n^2)$
    - ⋆ Melhor caso com pior sequencia de intervalos h: $O(nlog^2 n)$ (Pratt, Vaughan Ronald (1979). Shellsort and Sorting Networks - Outstanding Dissertations in the Computer Sciences)
  - Melhor caso com uma boa sequencia de intervalos h: $O(nlogn)$
  - Caso médio:

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?
  - Tempo de execução: muito sensível à ordem inicial dos elementos
  - Cada passagem de k em k, temos um vetor mais ordenado
    - ⋆ Como é adaptativo, menos comparações serão efetudas
    - ⋆ Conta com a possibilidade de acertar (ou aproximar) a posição correta
  - Empiricamente, observou-se sua eficiência em diversos casos
  - No pior caso, shellsort não é necessariamente quadrático (Sedgewick)
    - ⋆ As comparações são proporcionais a $N^{\frac{3}{2}}$
    - ⋆ Pior caso com pior sequencia de intervalos h: $O(n^2)$
    - ⋆ Melhor caso com pior sequencia de intervalos h: $O(nlog^2 n)$ (Pratt, Vaughan Ronald (1979). Shellsort and Sorting Networks - Outstanding Dissertations in the Computer Sciences)
  - Melhor caso com uma boa sequencia de intervalos h: $O(nlogn)$
  - Caso médio:
    - ⋆ Segundo Sedgewick (2011) nenhum resultado matemático estava disponível sobre o número médio de comparações para shellsort para entrada ordenada aleatoriamente

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
- Estabilidade?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3
  - Tem trocas com saltos?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3
  - Tem trocas com saltos?
    1. **2** 3 2' **1**

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3
  - Tem trocas com saltos?
    1. **2** 3 2' **1**
    2. **1** 3 2' **2**

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3
  - Tem trocas com saltos?
    1. **2** 3 2' **1**
    2. **1** 3 2' **2**
- *In-place*?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3
  - Tem trocas com saltos?
    1. **2** 3 2' **1**
    2. **1** 3 2' **2**
- *In-place*?
  - Utiliza memória extra significativa?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3
  - Tem trocas com saltos?
    1. **2** 3 2' **1**
    2. **1** 3 2' **2**
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3
  - Tem trocas com saltos?
    1. **2** 3 2' **1**
    2. **1** 3 2' **2**
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?
- Vamos testar.

Roteiro

Roteiro

Algoritmos de Ordenação Eficientes
Merge Sort

- Método dividir e conquistar
  - Dividir em pequenas partes
  - Ordenar essas partes
  - Combinar essas partes ordenadas
  - Até formar uma única sequência ordenada
- `https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge-sort-example-300px.gif`

Algoritmos de Ordenação Eficientes
Merge Sort

- Abordagem Top-Down: a partir da lista inteira, dividir em sub-listas
- Recursivamente:
  - A cada chamada, divide a entrada em sub-vetores para serem ordenados
    - merge_sort(int *v, int l, int r)
  - Quando chegar em um tamanho unitário, está ordenado em 1
  - Volta fazendo o *merge* do ordenado
    - merge(int *v, int l, int meio, int r)
    - Utiliza um vetor auxiliar

Algoritmos de Ordenação Eficientes
Merge Sort

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

```
 l                        m                          r
 0     1     2     3     4     5     6     7     8     9
[7     2     9     10    4     3     1     8     6     5]
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

l				m					r
0	1	2	3	4	5	6	7	8	9
[7	2	9	10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

```
 l                m                r
 0     1     2     3     4     5     6     7     8     9
[7     2     9    10     4]   [3     1     8     6     5]
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

```
 l                m                r
 0     1     2     3     4     5     6     7     8     9
[7     2     9] [10     4] [3     1     8     6     5]
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

```
 l       m       r
 0       1       2       3       4       5       6       7       8       9
[7       2       9]    [10      4]     [3       1       8       6       5]
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

l		m		r					
0		1		2	3	4	5	6	7	8	9
[7		2]		[9]	[10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

```
l=m     r
 0      1      2      3      4      5      6      7      8      9
[7      2]    [9]    [10     4]    [3      1      8      6      5]
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

```
l=m    r
 0     1     2     3     4     5     6     7     8     9
[7]   [2]   [9]   [10    4]   [3    1     8     6     5]
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

```
l=r   l=r
 0     1     2     3     4     5     6     7     8     9
[7]   [2]   [9]   [10    4]   [3    1     8     6     5]
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
l=m     r
 0      1      2      3      4      5      6      7      8      9
[7]    [2]    [9]   [10     4]    [3     1      8      6      5]
 i      j

[          ]
 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
l=m     r
 0      1      2      3      4      5      6      7      8      9
[7]    [2]    [9]   [10     4]    [3     1      8      6      5]
 i      j

[2       ]
 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
l=m     r
 0      1      2      3      4      5      6      7      8      9
[7]    [2]    [9]   [10     4]    [3     1      8      6      5]
 i        j>r
```

```
[2      7]
        k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
l=m     r
 0      1      2      3      4      5      6      7      8      9
[2      7]    [9]    [10     4]    [3      1      8      6      5]
```

```
[2      7]
 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l       m       r
 0       1       2       3       4       5       6       7       8       9
[2       7]     [9]     [10      4]     [3       1       8       6       5]
 i               j


[                 ]
 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l       m       r
 0       1       2       3       4       5       6       7       8       9
[2       7]     [9]     [10      4]     [3       1       8       6       5]
 i               j


[2                       ]
 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l       m       r
 0       1       2       3       4       5       6       7       8       9
[2       7]     [9]    [10       4]    [3       1       8       6       5]
         i       j

[2                       ]
         k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l       m       r
 0       1       2       3       4       5       6       7       8       9
[2       7]     [9]     [10      4]     [3       1       8       6       5]
         i       j

[2       7        ]
         k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l       m       r
 0       1       2       3       4       5       6       7       8       9
[2       7]     [9]    [10       4]    [3       1       8       6       5]
           i>m   j
```

```
[2       7       9]
                 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l       m       r
 0       1       2       3       4       5       6       7       8       9
[2       7       9]    [10       4]    [3       1       8       6       5]
```

```
[2       7       9]
                 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

			l=m	r				
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[10	4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

			l=m	r				
0	1	2	3	4	5	6	7	8	9
[2	7	9]	[10]	[4]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                    l=r   l=r
 0    1    2       3     4     5    6    7    8    9
[2    7    9]    [10]  [4]   [3    1    8    6    5]
                   i     j

                 [          ]
                  k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                    l=r   l=r
 0    1    2        3     4     5    6    7    8    9
[2    7    9]    [10]   [4]   [3    1    8    6    5]
                    i     j

                [4          ]
                 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                    l=r   l=r
 0     1     2      3     4     5     6     7     8     9
[2     7     9]   [10]   [4]   [3     1     8     6     5]
                    i          j>r
```

```
[4      10]
        k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                        l=r   l=r
 0     1     2          3     4     5     6     7     8     9
[2     7     9]        [4    10]   [3     1     8     6     5]
```

```
                       [4     10]
                        k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l              m              r
 0     1        2      3       4      5      6      7      8      9
[2     7        9]    [4      10]    [3     1      8      6      5]
 i                     j

[                                   ]
 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l              m              r
 0     1        2      3       4      5      6      7      8      9
[2     7        9]    [4      10]    [3     1      8      6      5]
 i                     j

[2                                   ]
 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l              m              r
 0     1        2      3       4      5      6      7      8      9
[2     7        9]    [4      10]    [3     1      8      6      5]
       i              j
```

```
[2                                   ]
       k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l          m          r
 0    1     2     3    4     5     6     7     8     9
[2    7     9]   [4    10]   [3    1     8     6     5]
      i          j
```

```
[2    4                 ]
      k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l              m              r
 0    1    2    3    4    5    6    7    8    9
[2    7    9]  [4   10]  [3    1    8    6    5]
     i              j

[2    4                   ]
               k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l              m              r
 0    1    2    3    4    5    6    7    8    9
[2    7    9]  [4   10]  [3    1    8    6    5]
     i              j

[2    4    7               ]
          k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l              m              r
 0     1        2      3       4      5      6      7      8      9
[2     7        9]    [4      10]    [3     1      8      6      5]
                i              j


[2     4        7                     ]
                      k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l              m              r
 0    1    2    3    4    5    6    7    8    9
[2    7    9]  [4   10]  [3    1    8    6    5]
           i         j

[2    4    7    9     ]
           k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l              m              r
 0     1        2     3        4     5     6     7     8     9
[2     7        9]   [4       10]   [3     1     8     6     5]
                  i>m            j


[2     4        7     9       10]
                                 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l              m              r
 0    1    2    3    4    5    6    7    8    9
[2    4    7    9   10] [3    1    8    6    5]
```

```
[2    4    7    9   10]
               k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3	1	8	6	5]

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

					l	m	r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3	1	8]	[6	5]

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

					l=m	r	l=r		
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3	1]	[8]	[6	5]

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

					l=r	l=r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3]	[1]	[8]	[6	5]

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

					l=m	r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3]	[1]	[8]	[6	5]
					i	j			
					[	]			
					k				

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

					l=m	r			
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[3]	[1]	[8]	[6	5]
					i	j			
					[1	3]			
					k				

Algoritmos de Ordenação Eficientes
Merge Sort

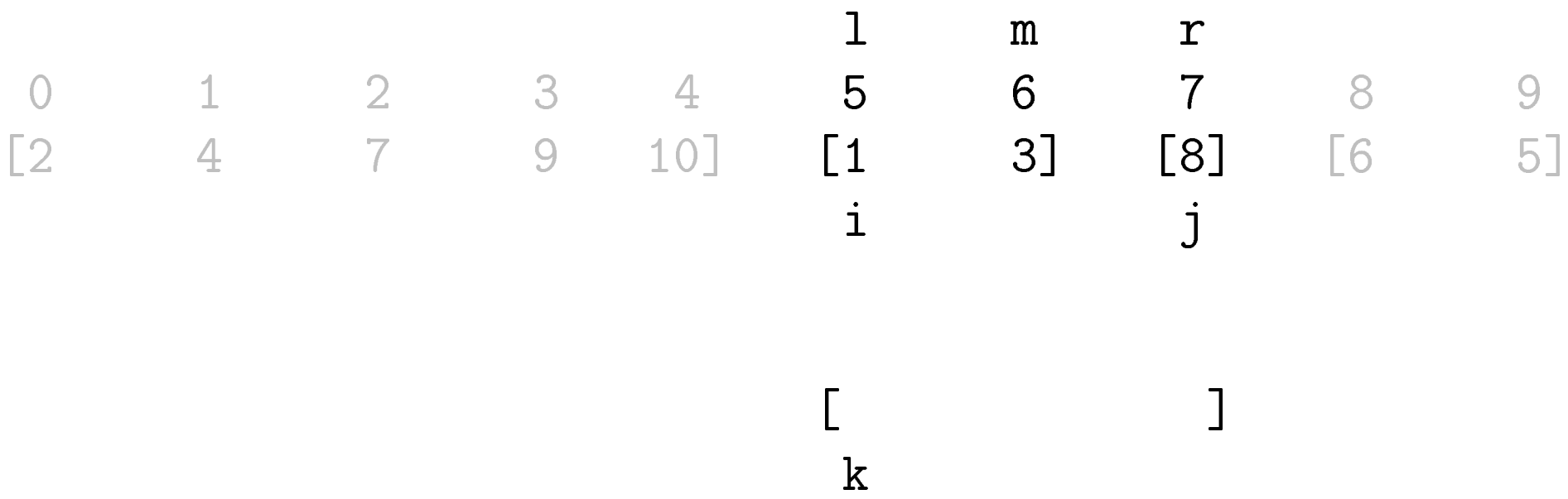
1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                                  l=m     r
 0     1     2     3     4      5      6      7     8     9
[2     4     7     9     10]   [1     3]   [8]   [6     5]


                                  [1     3]
                                   k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?



Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                                l       m       r
 0      1      2      3      4   5       6       7       8      9
[2      4      7      9      10] [1      3]      [8]     [6     5]
                                   i       j

                                [1                      ]
                                   k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                                  l       m       r
 0     1     2     3     4        5       6       7       8       9
[2     4     7     9     10]     [1      3]     [8]     [6      5]
                                     i>m   j

                                 [1      3       8]
                                               k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                                  l       m       r
 0     1     2     3     4        5       6       7       8     9
[2     4     7     9     10]     [1      3       8]     [6     5]


                                 [1      3       8]
                                                 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

```
                                                    l=m     r
 0     1     2     3     4     5     6     7        8      9
[2     4     7     9     10]  [1     3     8]     [6     5]
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. **Dividir** : $m = l + (r - l)/2 = (l + r)/2$
2. Ordenar e juntar

								l=m	r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[6]	[5]

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                                                    l=m     r
 0     1     2     3     4     5     6     7        8       9
[2     4     7     9     10]   [1     3     8]     [6]     [5]
                                                    i       j


                                                    [5      6]
                                                     k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
                                                    l=m     r
 0     1     2     3     4     5     6     7        8       9
[2     4     7     9     10]   [1     3     8]     [5      6]
```

```
                                                    [5      6]
                                                     k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
					i			j	

[				]
k				

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
						i		j	
					[1				]
						k			

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
							i	j	
					[1	3			]
							k		

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
							i		j

[1	3	5		]
			k	

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
							i		j>r
					[1	3	5	6	]
									k

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	8]	[5	6]
							i		j>r

[1	3	5	6	8]
				k

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

					l		m		r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	5	6	8]

					[1	3	5	6	8]
									k

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l                          m                                   r
 0     1     2     3     4     5     6     7     8     9
[2     4     7     9     10]  [1     3     5     6     8]
 i                          j

[                                                              ]
 k
```

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

l				m					r
0	1	2	3	4	5	6	7	8	9
[2	4	7	9	10]	[1	3	5	6	8]
				i					j>r

[1	2	3	4	5	6	7	8	9	10]
									k

Algoritmos de Ordenação Eficientes
Merge Sort

1. Dividir : $m = l + (r - l)/2 = (l + r)/2$
2. **Ordenar e juntar** : v[i] < v[j] ?

```
 l                                                    r
 0     1     2     3     4     5     6     7     8     9
[1     2     3     4     5     6     7     8     9     10]
```

```
[1     2     3     4     5     6     7     8     9     10]
                                                    k
```

Algoritmos de Ordenação Eficientes

1. merge_sort(v, 0, 5)
2. meio = (5+0)/2 = 2
3. merge_sort(v, 0, meio=2) : esquerda
   1. m = (2+0)/2 = 1
   2. merge_sort(v, 0, 1) : esquerda
      1. m = (1+0)/2 = 0
      2. merge_sort(v, 0, 0) : esquerda
      3. merge_sort(v, 1, 1) : direita
      4. merge(v, 0, 0, 1)
         6 5 3 1 2 4 : 5 6
   3. merge_sort(v, 2, 2) : direita
   4. merge(v, 0, 1, 2)
      5 6 **3** 1 2 4 : 3
      5 6 3 1 2 4 : 3 5
      5 **6** 3 1 2 4 : 3 5 6

Algoritmos de Ordenação Eficientes

4. merge_sort(v, meio+1=3, 5) : direita
   1. m = (5+3)/2 = 4
   2. merge_sort(v, 3, 4) : esquerda
      1. m = (4+3)/2 = 3
      2. merge_sort(v, 3, 3) : esquerda
      3. merge_sort(v, 4, 4) : direita
      4. merge(v, 3, 3, 4)
         3 5 6 **1** **2** 4 : 1 2
   3. merge_sort(v, 5, 5) : direita
   4. merge(v, 3, 4, 5)
      3 5 6 **1** 2 **4** : 1
      3 5 6 1 **2** **4** : 1 2
      3 5 6 1 2 **4** : 1 2 4

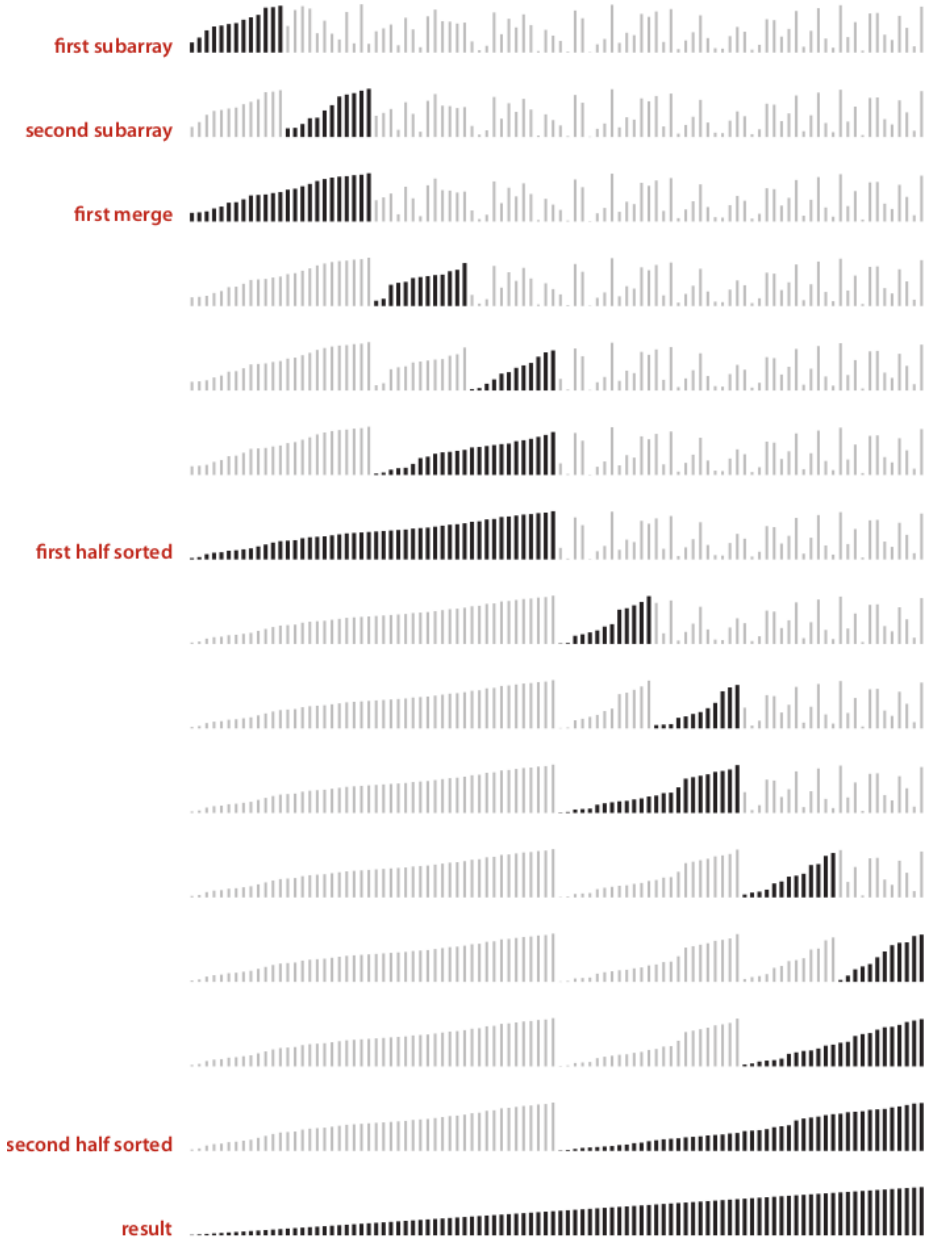
Algoritmos de Ordenação Eficientes

5. merge(v, 0, 2, 5)

3 5 6 **1** 2 4 : 1
3 5 6 1 **2** 4 : 1 2
3 5 6 1 2 **4** : 1 2 3
3 **5** 6 1 2 **4** : 1 2 3 4
3 **5** 6 1 2 4 : 1 2 3 4 5
3 5 **6** 1 2 4 : 1 2 3 4 5
3 5 6 1 2 4 : 1 2 3 4 5 6

Algoritmos de Ordenação Eficientes

Algoritmos de Ordenação Eficientes



Visual trace of top-down mergesort with cutoff for small subarrays

Algoritmos de Ordenação Eficientes

```
1  void merge_sort(int *v, int l, int r)
2  {
3      if (l >= r) return;
4      int m = (r+l)/2;
5
6      merge_sort(v, l, m);
7      merge_sort(v, m+1, r);
8      merge(v, l, m, r);
9  }
10
```

Algoritmos de Ordenação Eficientes

```
1  void merge(int *v, int l, int m, int r) {
2      //l=0  r=9 -> 10 itens = 9+1-0
3      //l=2  r=8 -> 7 itens = 8+1-2
4      int tam = r+1-l;
5
6      //alocar espaço auxiliar
7      int *aux = malloc(sizeof(int)*tam);
8
9      //auxiliares
10     int i=l; //inicio do sub-vetor esquerdo
11     int j=m+1; //inicio do sub-vetor direito
12     int k=0; //inicio do vetor auxiliar
13
14     while(i<=m && j<=r) //percorrer os sub-vetores
15     {
16         if(v[i] <= v[j]) //testar sub-vetores
17             aux[k++] = v[i++];//ordenar no vetor auxiliar
18         else
19             aux[k++] = v[j++];//ordenar no vetor auxiliar
20     }
21
22     //ainda tem elementos no sub-vetor esquerdo?
23     while(i<=m) aux[k++] = v[i++];
24
25     //ainda tem elementos no sub-vetor direito?
26     while(j<=r) aux[k++] = v[j++];
27
```

Algoritmos de Ordenação Eficientes

```
28
29      k=0; //indice do aux
30      for(i=l; i<=r; i++) //indice do v
31          v[i] = aux[k++];//copiar o aux[k] para v[i]
32
33      //liberar memória
34      free(aux);
35  }
```

Algoritmos de Ordenação Eficientes

```
1  void merge(int *v, int l, int m, int r) {
2      //quanto elementos?
3      int tam = r+1-l;
4
5      //alocar espaço auxiliar
6      int *aux = malloc(tam*sizeof(int));
7
8      //auxiliares
9      int i=l; //inicio do sub-vetor esquerdo
10     int j=m+1; //inicio do sub-vetor direito
11     int k=0; //inicio do vetor auxiliar
12
13     //ordenar em aux[k]
14     while(k<tam) //condição de parada do aux
15     {
16         if(i>m) //ordenou todo o primeiro sub-vetor
17             aux[k++] = v[j++];//consome o segundo sub-vetor -> ordene no
       aux
18
19         else if (j>r) //ordenou todo o segundo sub-vetor
20             aux[k++] = v[i++];//consome o primeiro sub-vetor -> ordene no
        aux
21
22         else if (v[i] < v[j]) //testar sub-vetores
23             aux[k++] = v[i++];//ordene no aux
24         else
25             aux[k++] = v[j++];//ordene no aux
```

Algoritmos de Ordenação Eficientes

```
26      }
27
28      k=0; //indice do aux
29      for(i=l; i<=r; i++) //indice do v
30          v[i] = aux[k++];//copiar o aux[k] para v[i]
31
32      //liberar memória
33      free(aux);
34
35  }
```

Algoritmos de Ordenação Eficientes

- Complexidade assintótica
  - Pior caso: O(nlogn)
  - Caso médio: O(nlogn)
  - Melhor caso: O(nlogn)

```
1  void merge(int *v, int l, int m, int r) {
2      ...
3
4      while(k<r+1-l) { ... }  //n
5
6  }
```

```
1  //F(n)
2  void merge_sort(int *v, int l, int r) {
3      if (l >= r) return;
4      int m = (r+l)/2;
5
6      merge_sort(v, l, m);  //F(n/2)
7      merge_sort(v, m+1, r); //F(n/2)
8      merge(v, l, m, r); //n
9  }
10
```

Algoritmos de Ordenação Eficientes

- Complexidade assintótica
  - Pior caso, médio, melhor: $O(n \log n)$

$$
\begin{aligned}
f(n) &= 2 * f(\frac{n}{2}) + n \\
&= 2 * (2 * f(\frac{n}{4}) + \frac{n}{2}) + n \\
&= 2^2 * f(\frac{n}{2^2}) + 2 * \frac{n}{2} + n \\
&= 2^2 * f(\frac{n}{2^2}) + 2 * n \\
&= 2^2 * (2 * f(\frac{n}{2^3}) + \frac{n}{2^2}) + 2 * n \\
&= 2^3 * f(\frac{n}{2^3}) + 2^2 * \frac{n}{2^2} + 2 * n \\
&= 2^3 * f(\frac{n}{2^3}) + 3 * n \\
&= 2^i * f(\frac{n}{2^i}) + i * n : 2^i = n : \log_2 2^i = \log_2 n : i = \log_2 n \\
&= n * f(1) + n * \log n
\end{aligned}
$$

Algoritmos de Ordenação Eficientes

- In-place?

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N
- Adaptatividade?

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N
- Adaptatividade?
  - Ordenação: não diminui as divisões, nem as comparações no merge

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N
- Adaptatividade?
  - Ordenação: não diminui as divisões, nem as comparações no merge
- Estabilidade?

Algoritmos de Ordenação Eficientes

- In-place?
  - Memória extra: proporcional a N
- Adaptatividade?
  - Ordenação: não diminui as divisões, nem as comparações no merge
- Estabilidade?
  - Mantém a ordem relativa

Algoritmos de Ordenação Eficientes

- Otimizações
  - Nos sub-vetores pequenos, alterne para o Insertion Sort

Algoritmos de Ordenação Eficientes

- Otimizações
  - Nos sub-vetores pequenos, alterne para o Insertion Sort
    - Cerca de 15 itens mais ou menos

Algoritmos de Ordenação Eficientes

- Otimizações
  - Nos sub-vetores pequenos, alterne para o Insertion Sort
    - Cerca de 15 itens mais ou menos
    - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento

Algoritmos de Ordenação Eficientes

- Otimizações
  - Nos sub-vetores pequenos, alterne para o Insertion Sort
    - Cerca de 15 itens mais ou menos
    - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento
  - Teste se o vetor já está em ordem

Algoritmos de Ordenação Eficientes

- Otimizações
  - Nos sub-vetores pequenos, alterne para o Insertion Sort
    - Cerca de 15 itens mais ou menos
    - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento
  - Teste se o vetor já está em ordem
    - Podemos reduzir o tempo de execução para linear para arrays que já estão em ordem adicionando um teste para pular a chamada para merge() se $v[meio]$ for menor ou igual a $v[meio + 1]$

Algoritmos de Ordenação Eficientes

- Otimizações
  - Nos sub-vetores pequenos, alterne para o Insertion Sort
    - Cerca de 15 itens mais ou menos
    - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento
  - Teste se o vetor já está em ordem
    - Podemos reduzir o tempo de execução para linear para arrays que já estão em ordem adicionando um teste para pular a chamada para merge() se $v[meio]$ for menor ou igual a $v[meio + 1]$
    - Não diminui as chamadas recursivas, mas o tempo de execução para qualquer subarray ordenado é linear

Algoritmos de Ordenação Eficientes

- Otimizações
  - Nos sub-vetores pequenos, alterne para o Insertion Sort
    - ⋆ Cerca de 15 itens mais ou menos
    - ⋆ Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento
  - Teste se o vetor já está em ordem
    - ⋆ Podemos reduzir o tempo de execução para linear para arrays que já estão em ordem adicionando um teste para pular a chamada para merge() se $v[meio]$ for menor ou igual a $v[meio + 1]$
    - ⋆ Não diminui as chamadas recursivas, mas o tempo de execução para qualquer subarray ordenado é linear
  - Não utilize um vetor auxiliar local na função merge (Sedegewick)

Algoritmos de Ordenação Eficientes

- Otimizações
  - Nos sub-vetores pequenos, alterne para o Insertion Sort
    - Cerca de 15 itens mais ou menos
    - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento
  - Teste se o vetor já está em ordem
    - Podemos reduzir o tempo de execução para linear para arrays que já estão em ordem adicionando um teste para pular a chamada para merge() se $v[meio]$ for menor ou igual a $v[meio + 1]$
    - Não diminui as chamadas recursivas, mas o tempo de execução para qualquer subarray ordenado é linear
  - Não utilize um vetor auxiliar local na função merge (Sedegewick)
    - Declare o auxiliar no merge_sort e passe como argumento para o merge

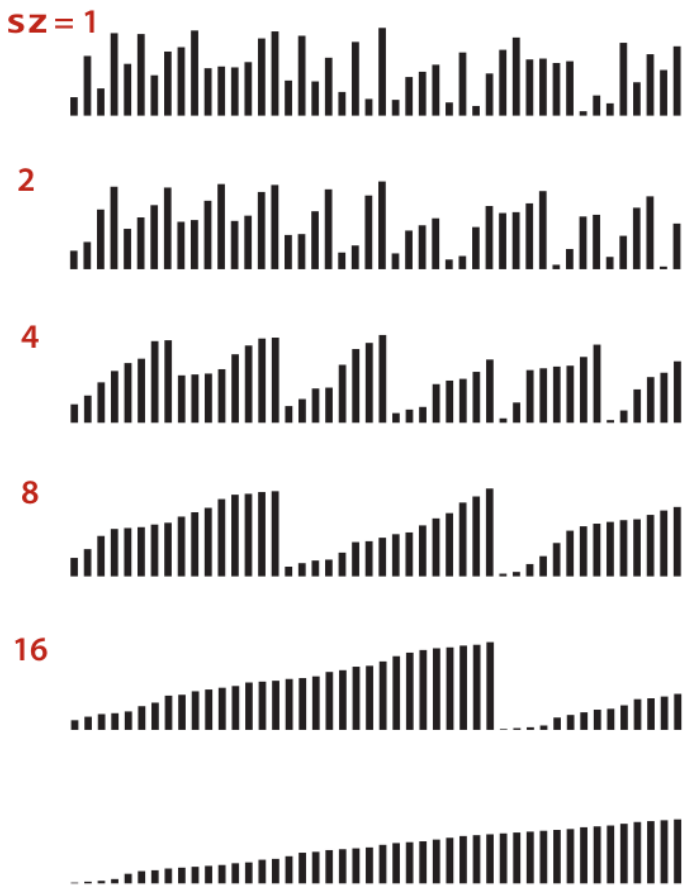
Algoritmos de Ordenação Eficientes

- Otimizações
  - Nos sub-vetores pequenos, alterne para o Insertion Sort
    - Cerca de 15 itens mais ou menos
    - Melhora o tempo de execução de uma implementação típica de mergesort em 10 a 15 por cento
  - Teste se o vetor já está em ordem
    - Podemos reduzir o tempo de execução para linear para arrays que já estão em ordem adicionando um teste para pular a chamada para merge() se $v[meio]$ for menor ou igual a $v[meio + 1]$
    - Não diminui as chamadas recursivas, mas o tempo de execução para qualquer subarray ordenado é linear
  - Não utilize um vetor auxiliar local na função merge (Sedegewick)
    - Declare o auxiliar no merge_sort e passe como argumento para o merge
    - Diminuir o overhead(sobrecarga) dessa criação a cada merge

Algoritmos de Ordenação Eficientes

- Abordagem Bottom-Up
  - merge 1 por 1
    - ★ sub-vetores de tamanho 1
    - ★ resultando em um sub-vetor de tamanho 2
  - merge 2 por 2:
    - ★ sub-vetores de tamanho 2
    - ★ resultando em um sub-vetor de tamanho 4
  - e assim por diante
- Consiste em uma sequencia de passos pelo vetor inteiro, fazendo "sz por sz" uniões
- Começando por 1 por 1 e dobrando em cada passo
- Complexidade: mesma da abordagem top-down

Algoritmos de Ordenação Eficientes



Visual trace of bottom-up mergesort

sz = 1

```
merge(a,  0,  0,  1)
merge(a,  2,  2,  3)
merge(a,  4,  4,  5)
merge(a,  6,  6,  7)
merge(a,  8,  8,  9)
merge(a, 10, 10, 11)
merge(a, 12, 12, 13)
merge(a, 14, 14, 15)
```

sz = 2

```
merge(a,  0,  1,  3)
merge(a,  4,  5,  7)
merge(a,  8,  9, 11)
merge(a, 12, 13, 15)
```

sz = 4

```
merge(a,  0,  3,  7)
merge(a,  8, 11, 15)
```

sz = 8

```
merge(a,  0,  7, 15)
```

Algoritmos de Ordenação Eficientes

```
1  void mergeBU_sort(int *v, int l, int r)
2  {
3      int tam = (r-l+1);
4
5      for (int sz=1; sz<tam; sz=2*sz)
6      {
7          for (int lo=l; lo<tam-sz; lo+=2*sz)
8          {
9              int hi = lo+2*sz-1;
10             if(hi>tam-1) hi = tam-1;
11
12             merge(v, lo, lo+sz-1, hi);
13         }
14     }
15 }
16
```

Algoritmos de Ordenação Eficientes

- MergeSort é mais rápido do que ShellSort?
  - O tempo é similar, diferindo em pequenos fatores constantes
  - Porém, ainda não comprovou-se que o Shell Sort é $O(n \log n)$ para dados aleatórios
  - Portanto, o crescimento assintótico do Shell Sort nos casos médios podem ser altos
- Merge Sort em listas encadeadas?
  - Observem os códigos do Sedgewick e tentem implementar suas versões
  - Merge
  - Abordagem Top-Down
  - Abordagem Bottom-Up

Algoritmos de Ordenação Eficientes

- MergeSort é mais rápido do que ShellSort?
  - O tempo é similar, diferindo em pequenos fatores constantes
  - Porém, ainda não comprovou-se que o Shell Sort é $O(n \log n)$ para dados aleatórios
  - Portanto, o crescimento assintótico do Shell Sort nos casos médios podem ser altos
- Merge Sort em listas encadeadas?
  - Observem os códigos do Sedgewick e tentem implementar suas versões
  - Merge

```
link merge(link a, link b)
  { struct node head; link c = &head;
    while ((a != NULL) && (b != NULL))
      if (less(a->item, b->item))
        { c->next = a; c = a; a = a->next; }
      else
        { c->next = b; c = b; b = b->next; }
    c->next = (a == NULL) ? b : a;
    return head.next;
  }
```

  - Abordagem Top-Down

Algoritmos de Ordenação Eficientes

- MergeSort é mais rápido do que ShellSort?
  - O tempo é similar, diferindo em pequenos fatores constantes
  - Porém, ainda não comprovou-se que o Shell Sort é $O(n \log n)$ para dados aleatórios
  - Portanto, o crescimento assintótico do Shell Sort nos casos médios podem ser altos
- Merge Sort em listas encadeadas?
  - Observem os códigos do Sedgewick e tentem implementar suas versões
  - Merge
  - Abordagem Top-Down

```
link merge(link a, link b);
link mergesort(link c)
  { link a, b;
    if (c == NULL || c->next == NULL) return c;
    a = c; b = c->next;
    while ((b != NULL) && (b->next != NULL))
      { c = c->next; b = b->next->next; }
    b = c->next; c->next = NULL;
    return merge(mergesort(a), mergesort(b));
  }
```

  - Abordagem Bottom-Up

Algoritmos de Ordenação Eficientes

- MergeSort é mais rápido do que ShellSort?
  - O tempo é similar, diferindo em pequenos fatores constantes
  - Porém, ainda não comprovou-se que o Shell Sort é $O(n \log n)$ para dados aleatórios
  - Portanto, o crescimento assintótico do Shell Sort nos casos médios podem ser altos
- Merge Sort em listas encadeadas?
  - Observem os códigos do Sedgewick e tentem implementar suas versões
  - Merge
  - Abordagem Top-Down
  - Abordagem Bottom-Up

```
link mergesort(link t)
  { link u;
    for (Qinit(); t != NULL; t = u)
      { u = t->next; t->next = NULL; Qput(t); }
    t = Qget();
    while (!Qempty())
      { Qput(t); t = merge(Qget(), Qget()); }
    return t;
  }
```