Ordenação de dados

Profª. Rose Yuri Shimizu

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

- Robert Sedgewick Algorithms in C, Addison Wesley, 3nd ed.
- Algorithms, 4th Edition - Robert Sedgewick e Kevin Wayne
- 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

Algoritmos de Ordenação - Características

1. Complexidade (espacial, temporal)
   - Quadráticos: simples e suficiente para arquivos pequenos
   - Linearítmicos: mais complexos 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
   - Diminui-se o custo

Algoritmos de Ordenação - Características

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

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

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Posicionar menor(swap): `v[ i ]` $\leftrightarrow$ `v[ m ]`
- Elemento selecionado na sua posição correta/final (i)

```
          l                                   r
          0     1     2     3     4     5
v     [ _3_ | 2 | 4 | 6 | _1_ | 5 ]
          i                             j
                                m
```

Algoritmos de Ordenação Elementares

Selection Sort

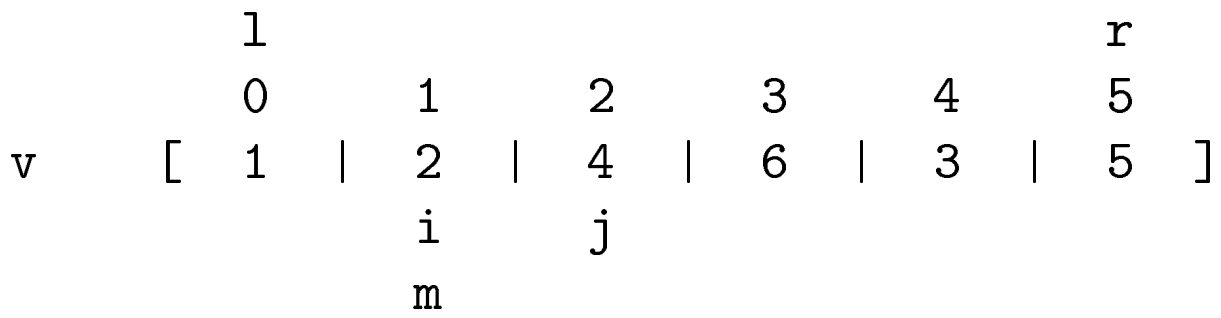
- i : posicionar
- j : selecionar
- m : índice do menor
- Posicionar menor(swap): `v[ i ]` $\leftrightarrow$ `v[ m ]`
- Elemento selecionado na sua posição correta/final (i)

```
          l                                   r
          0     1     2     3     4     5
v     [ _1_ | 2   | 4   | 6   | _3_ | 5   ]
          i                             j
                                m
```

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?



Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

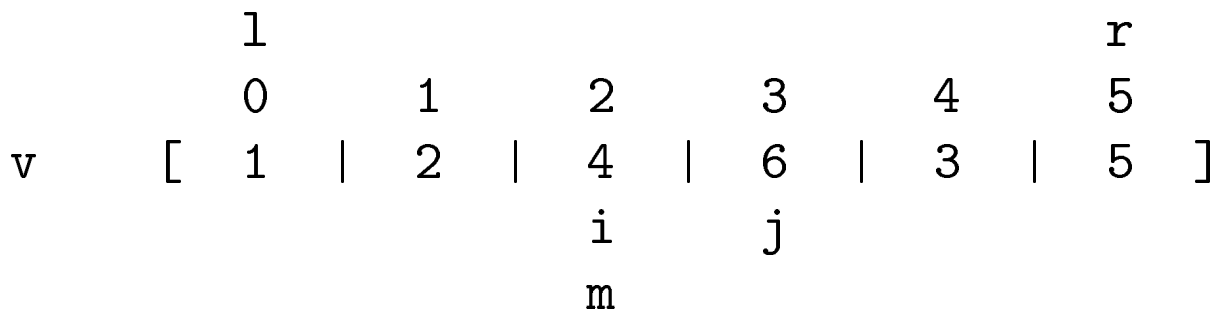
- i : posicionar
- j : selecionar
- m : índice do menor
- Posicionar menor(swap): `v[ j ] == v[ m ]  ?  sem swap`

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

Algoritmos de Ordenação Elementares

Selection Sort

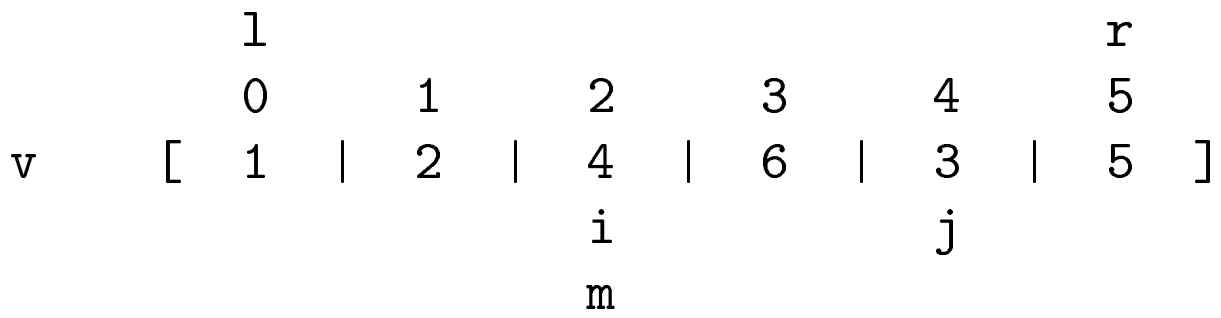
- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?



Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?



Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Posicionar menor(swap): `v[ i ]` $\leftrightarrow$ `v[ m ]`
- Elemento selecionado na sua posição correta/final (i)

```
          l                                   r
          0     1     2     3     4     5
v     [  1  |  2  | _4_ |  6  | _3_ |  5  ]
                    i                       j
                                m
```

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Posicionar menor(swap): `v[ i ]` $\leftrightarrow$ `v[ m ]`
- Elemento selecionado na sua posição correta/final (i)

```
          l                                   r
          0     1     2     3     4     5
v     [  1  |  2  | _3_ |  6  | _4_ |  5  ]
                    i                       j
                                m
```

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Posicionar menor(swap): `v[ i ]` $\leftrightarrow$ `v[ m ]`
- Elemento selecionado na sua posição correta/final (i)

```
          l                                   r
          0      1      2      3      4      5
v     [  1   |   2   |   3   | _6_  | _4_  |   5   ]
                              i                  j
                                     m
```

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Posicionar menor(swap): `v[ i ]` $\leftrightarrow$ `v[ m ]`
- Elemento selecionado na sua posição correta/final (i)

```
          l                                   r
          0      1      2      3      4      5
v     [  1   |  2   |  3   | _4_  | _6_  |  5   ]
                             i                  j
                                    m
```

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Selecionar `v[ j ] < v[ m ]` ?

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

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Posicionar menor(swap): `v[ i ]` $\leftrightarrow$ `v[ m ]`
- Elemento selecionado na sua posição correta/final (i)

```
          l                                   r
          0     1     2     3     4     5
v     [ 1   |  2  |  3  |  4  | _6_ | _5_ ]
                                  i           j
                                        m
```

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Posicionar menor(swap): `v[ i ]` $\leftrightarrow$ `v[ m ]`
- Elemento selecionado na sua posição correta/final (i)

```
          l                                   r
          0     1     2     3     4     5
v     [  1  |  2  |  3  |  4  | _5_ | _6_ ]
                                  i           j
                                        m
```

Algoritmos de Ordenação Elementares

Selection Sort

- i : posicionar
- j : selecionar
- m : índice do menor
- Terminou? Vetor ordenado.

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

```
1 void selection_sort(int v[], int l, int r) {
2   int menor; //variável auxiliar
```

```
1 void selection_sort(int v[], int l, int r) {
2   int menor; //variável auxiliar
3
4   //para cada elemento
```

```
1 void selection_sort(int v[], int l, int r) {
2   int menor; //variável auxiliar
3
4   //para cada elemento
5   for(int i=l; i<r; i++) {
6     //supõe-se que i seja o menor
```

```
1  void selection_sort(int v[], int l, int r) {
2    int menor; //variável auxiliar
3
4    //para cada elemento
5    for(int i=l; i<r; i++) {
6      //supõe-se que i seja o menor
7      menor = i;
8
9      //SELECIONAR
10     //para os próximos elementos
```

```
1  void selection_sort(int v[], int l, int r) {
2    int menor; //variável auxiliar
3
4    //para cada elemento
5    for(int i=l; i<r; i++) {
6      //supõe-se que i seja o menor
7      menor = i;
8
9      //SELECIONAR
10     //para os próximos elementos
11     for(int j=i+1; j<=r; j++)
12         if(v[j] < v[menor]) //se achar um menor
```

```
1  void selection_sort(int v[], int l, int r) {
2    int menor; //variável auxiliar
3
4    //para cada elemento
5    for(int i=l; i<r; i++) {
6      //supõe-se que i seja o menor
7      menor = i;
8
9      //SELECIONAR
10     //para os próximos elementos
11     for(int j=i+1; j<=r; j++)
12         if(v[j] < v[menor]) //se achar um menor
13             menor = j;      //salve a posição do menor
```

```
1  void selection_sort(int v[], int l, int r) {
2    int menor; //variável auxiliar
3
4    //para cada elemento
5    for(int i=l; i<r; i++) {
6      //supõe-se que i seja o menor
7      menor = i;
8
9      //SELECIONAR
10     //para os próximos elementos
11     for(int j=i+1; j<=r; j++)
12         if(v[j] < v[menor]) //se achar um menor
13             menor = j;      //salve a posição do menor
14
15     //POSICIONAR
16     //se a suposição estava incorreta
```

```
1  void selection_sort(int v[], int l, int r) {
2    int menor; //variável auxiliar
3
4    //para cada elemento
5    for(int i=l; i<r; i++) {
6      //supõe-se que i seja o menor
7      menor = i;
8
9      //SELECIONAR
10     //para os próximos elementos
11     for(int j=i+1; j<=r; j++)
12         if(v[j] < v[menor]) //se achar um menor
13             menor = j;      //salve a posição do menor
14
15     //POSICIONAR
16     //se a suposição estava incorreta
17     if(i != menor)
18         exch(v[i], v[menor]) //reposicione o menor
19   }
20 }
```

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
- Adaptatividade? (aproveita ordenação)
- Estabilidade? (mantém ordem relativa)
- *In-place*? (espaço adicional)

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
- Estabilidade? (mantém ordem relativa)
- *In-place*? (espaço adicional)

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
  - Possível identificar ordenação?
- Estabilidade? (mantém ordem relativa)
- *In-place*? (espaço adicional)

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
  - Possível identificar ordenação?
  - Não, pois a cada iteração, cada elemento é comparado somente com o menor
  - Portanto, não é adaptativo
- Estabilidade? (mantém ordem relativa)
- *In-place*? (espaço adicional)

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
  - Possível identificar ordenação?
  - Não, pois a cada iteração, cada elemento é comparado somente com o menor
  - Portanto, não é adaptativo
- Estabilidade? (mantém ordem relativa)
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
- *In-place*? (espaço adicional)

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
  - Possível identificar ordenação?
  - Não, pois a cada iteração, cada elemento é comparado somente com o menor
  - Portanto, não é adaptativo
- Estabilidade? (mantém ordem relativa)
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
- *In-place*? (espaço adicional)

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
  - Possível identificar ordenação?
  - Não, pois a cada iteração, cada elemento é comparado somente com o menor
  - Portanto, não é adaptativo
- Estabilidade? (mantém ordem relativa)
  - 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?
  - Tem trocas com saltos?
    - ⋆ 1 3 4' 4
- *In-place*? (espaço adicional)

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
  - Possível identificar ordenação?
  - Não, pois a cada iteração, cada elemento é comparado somente com o menor
  - Portanto, não é adaptativo
- Estabilidade? (mantém ordem relativa)
  - 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*? (espaço adicional)

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
  - Possível identificar ordenação?
  - Não, pois a cada iteração, cada elemento é comparado somente com o menor
  - Portanto, não é adaptativo
- Estabilidade? (mantém ordem relativa)
  - 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*? (espaço adicional)
  - Utiliza memória extra significativa?

Algoritmos de Ordenação Elementares - Selection Sort

- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e N trocas: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
  - Possível identificar ordenação?
  - Não, pois a cada iteração, cada elemento é comparado somente com o menor
  - Portanto, não é adaptativo
- Estabilidade? (mantém ordem relativa)
  - 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*? (espaço adicional)
  - 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: $O(N^2)$
  - Pior, Melhor, Médio caso: $O(N^2)$
- Adaptatividade? (aproveita ordenação)
  - Possível identificar ordenação?
  - Não, pois a cada iteração, cada elemento é comparado somente com o menor
  - Portanto, não é adaptativo
- Estabilidade? (mantém ordem relativa)
  - 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*? (espaço adicional)
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?
  - Não, portanto, é *in-place*
  - Complexidade espacial auxiliar constante

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 com trocas de elementos: possível com listas encadeadas?

Roteiro

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Comparar adjacentes:
  1. Do início(base), 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

- Topo: 5
- Swaps: 0
- 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

- Topo: 5
- Swaps: 0
- 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

- Topo: 5
- Swaps: 0
- 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

- Topo: 5
- Swaps: 0
- 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

- Topo: 5
- Swaps: 0
- 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

- Topo: 5
- Swaps: 1
- 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

- Topo: 5
- Swaps: 1
- 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

- Topo: 5
- Swaps: 2 $\rightarrow$ reordenações
- Comparar adjacentes `v[ j ] > v[ j+1 ]` ? Flutua (swap)
- Um elemento flutuado para a sua posição correta/final (topo = r)

```
          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

- Topo: 4
- Swaps: 0
- 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

- Topo: 4
- Swaps: 0
- 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

- Topo: 4
- Swaps: 0
- 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

- Topo: 4
- Swaps: 1
- 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

- Topo: 4
- Swaps: 1
- 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

- Topo: 4
- Swaps: 1 $\rightarrow$ reordenações
- Comparar adjacentes `v[ j ] > v[ j+1 ]` ?
- Um elemento flutuado para a sua posição correta/final (topo = r)

```
          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

- Topo: 3
- Swaps: 0
- 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

- Topo: 3
- Swaps: 0
- 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

- Topo: 3
- Swaps: 1
- 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

- Topo: 3
- Swaps: 1
- 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

- Topo: 3
- Swaps: 1 $\rightarrow$ reordenações
- Comparar adjacentes `v[ j ] > v[ j+1 ]` ?
- Um elemento flutuado para a sua posição correta/final (topo = r)

```
          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

- Topo: 2
- Swaps: 0
- 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

- Topo: 2
- Swaps: 1
- 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

- Topo: 2
- Swaps: 1 $\rightarrow$ reordenações
- Comparar adjacentes `v[ j ] > v[ j+1 ]` ?
- Um elemento flutuado para a sua posição correta/final (topo = r)

```
          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

- Topo: 1
- Swaps: 0
- 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

- Topo: 1
- Swaps: 0 $\rightarrow$ sem reordenações
- Comparar adjacentes `v[ j ] > v[ j+1 ]` ?
- Um elemento flutuado para a sua posição correta/final (topo = r)

```
          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
4     for(int j=l; j<r; j++) { //a partir da base
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1 void bubble_sort(int v[], int l, int r){
2
3
4       for(int j=l; j<r; j++) { //a partir da base
5         if(v[j] > v[j+1]) {  //se for uma bolha maior
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1 void bubble_sort(int v[], int l, int r){
2
3
4     for(int j=l; j<r; j++) { //a partir da base
5       if(v[j] > v[j+1]) {  //se for uma bolha maior
6         exch(v[j], v[j+1]) //flutue
7
8       }
9     }
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1  void bubble_sort(int v[], int l, int r){
2
3
4       for(int j=l; j<r; j++) { //a partir da base
5         if(v[j] > v[j+1]) {  //se for uma bolha maior
6           exch(v[j], v[j+1]) //flutue
7
8         }
9       }
10      //topo?!
11
12 }
13 //
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1  void bubble_sort(int v[], int l, int r){
2
3    while(r>l) { //para cada novo topo
4      for(int j=l; j<r; j++) { //a partir da base
5        if(v[j] > v[j+1]) {  //se for uma bolha maior
6          exch(v[j], v[j+1]) //flutue
7
8        }
9      }
10     r--;
11   }
12 }
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1  void bubble_sort(int v[], int l, int r){
2
3    while(r>l) { //para cada novo topo
4      for(int j=l; j<r; j++) { //a partir da base
5        if(v[j] > v[j+1]) {  //se for uma bolha maior
6          exch(v[j], v[j+1]) //flutue
7
8        }
9      }
10     r--;
11   }
12 }
13 //é possível identificar reordenações??
```

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

- Topo: 5
- **Swaps: 0**
- Alcançou o topo sem trocas?

```
          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

- Topo: 5
- **Swaps: 0**
- Alcançou o topo sem trocas?
- Vetor ordenado

```
          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    int swap; //identificar trocas
3    while(r>l) {
4
5      for(int j=l; j<r; j++) {
6        if(v[j] > v[j+1]) {
7          exch(v[j], v[j+1])
8
9        }
10     }
11     r--;
12   }
13 }
14
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1  void bubble_sort(int v[], int l, int r){
2    int swap;
3    while(r>l) {
4      swap = 0;  //sem trocas nesta iteração
5      for(int j=l; j<r; j++) {
6        if(v[j] > v[j+1]) {
7          exch(v[j], v[j+1])
8
9        }
10     }
11     r--;
12   }
13 }
14
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1  void bubble_sort(int v[], int l, int r){
2    int swap;
3    while(r>l) {
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; //reordenação
9        }
10     }
11     r--;
12   }
13 }
14
```

Algoritmos de Ordenação Elementares
Bubble Sort

```
1  void bubble_sort(int v[], int l, int r){
2    int swap = 1; //primeira verificação
3    while(r>l && swap) { //se ocorreu reordenação
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     r--;
12   }
13 }
14
```

```c
void bubble_sort(int v[], int l, int r){
    int swap = 1;
    while(r>l && swap) { //n
        swap = 0;
        for(int j=l; j<r; j++) { //n-x, sendo x={1,2,3...n}

            //comparações
            //(n-1), (n-2), (n-3), .. , 0
            if(v[j] > v[j+1]) {

                //trocas
                //(n-1), (n-2), (n-3), .. , 0
                exch(v[j], v[j+1])
                swap = 1;
            }
        }
        r--;
    }
    //f(n) = n*(n-x) = (n-1)+(n-2)+(n-3)+...+(n-n)
    //f(n) = (n*n)/2 + (n*n)/2
}
```

Algoritmos de Ordenação Elementares - Bubble Sort

- Adaptatividade?
- Complexidade assintótica?
- Adaptatividade x Custo: cada elemento é posicionado até encontrar um maior (decrescente) no sub-conjunto dos sucessores, não sendo necessário a troca com todos os elementos e a identificação da ordenação total diminui as possíveis comparações posteriores

Algoritmos de Ordenação Elementares - Bubble Sort

- Adaptatividade?
  - Possível identificar ordenação?
- Complexidade assintótica?
- Adaptatividade x Custo: cada elemento é posicionado até encontrar um maior (decrescente) no sub-conjunto dos sucessores, não sendo necessário a troca com todos os elementos e a identificação da ordenação total diminui as possíveis comparações posteriores

Algoritmos de Ordenação Elementares - Bubble Sort

- Adaptatividade?
  - Possível identificar ordenação?
  - Sim, pois a cada iteração, os elementos são comparados entre si sendo possível a identificação da ordenação
  - Portanto, é adaptativo
- Complexidade assintótica?

- Adaptatividade x Custo: cada elemento é posicionado até encontrar um maior (decrescente) no sub-conjunto dos sucessores, não sendo necessário a troca com todos os elementos e a identificação da ordenação total diminui as possíveis comparações posteriores

Algoritmos de Ordenação Elementares - Bubble Sort

- Adaptatividade?
  - Possível identificar ordenação?
  - Sim, pois a cada iteração, os elementos são comparados entre si sendo possível a identificação da ordenação
  - Portanto, é adaptativo
- Complexidade assintótica?
  - Cerca de $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas: $O(N^2)$
  - Pior, Médio caso: $O(N^2)$
  - Melhor caso: $O(N)$ (como?)
- Adaptatividade x Custo: cada elemento é posicionado até encontrar um maior (decrescente) no sub-conjunto dos sucessores, não sendo necessário a troca com todos os elementos e a identificação da ordenação total diminui as possíveis comparações posteriores

Algoritmos de Ordenação Elementares - Bubble Sort

- Estabilidade?

- *In-place*?

Algoritmos de Ordenação Elementares - Bubble Sort

- Estabilidade?
  - 2 4 3 4' 1 $\rightarrow$ mantém a ordem relativa?

- *In-place*?

Algoritmos de Ordenação Elementares - Bubble Sort

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

- *In-place*?

Algoritmos de Ordenação Elementares - Bubble Sort

- 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

- 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

- 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

- 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*
  - Complexidade espacial auxiliar constante

Algoritmos de Ordenação Elementares - Bubble Sort

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

Algoritmos de Ordenação Elementares - Bubble Sort

- Selection Sort x Bubble sort?
  - Bubble sort ($2\frac{N^2}{2}$) é pior que o selection ($\frac{N^2}{2} + N$)
  - Sempre?
  - Teste com as entradas "16-aleatorio" e "17-quaseordenado" do conjunto de testes
- Bubble Sort com listas encadeadas??
- Variação

Algoritmos de Ordenação Elementares - Bubble Sort

- Selection Sort x Bubble sort?
  - Bubble sort ($2\frac{N^2}{2}$) é pior que o selection ($\frac{N^2}{2} + N$)
  - Sempre?
  - Teste com as entradas "16-aleatorio" e "17-quaseordenado" do conjunto de testes
- Bubble Sort com listas encadeadas??
  - Percorre a lista sequencialmente com troca de elementos: possível com listas encadeadas?
- Variação

Algoritmos de Ordenação Elementares - Bubble Sort

- Selection Sort x Bubble sort?
  - Bubble sort ($2\frac{N^2}{2}$) é pior que o selection ($\frac{N^2}{2} + N$)
  - Sempre?
  - Teste com as entradas "16-aleatorio" e "17-quaseordenado" do conjunto de testes
- Bubble Sort com listas encadeadas??
  - Percorre a lista sequencialmente com troca de elementos: possível com listas encadeadas?
- Variação
  - Shaker sort: a cada iteração, colocar o maior elemento no topo e o menor elemento na base
  - Complexidade assintótica quadrática
- Vamos testar.

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 até encontrar um menor
3. Cada iteração, resulta em um vetor parcialmente ordenado
   - Início até a posição atual

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- i: posição inicial
- j: percorre antecessores
- Menor que o 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
```

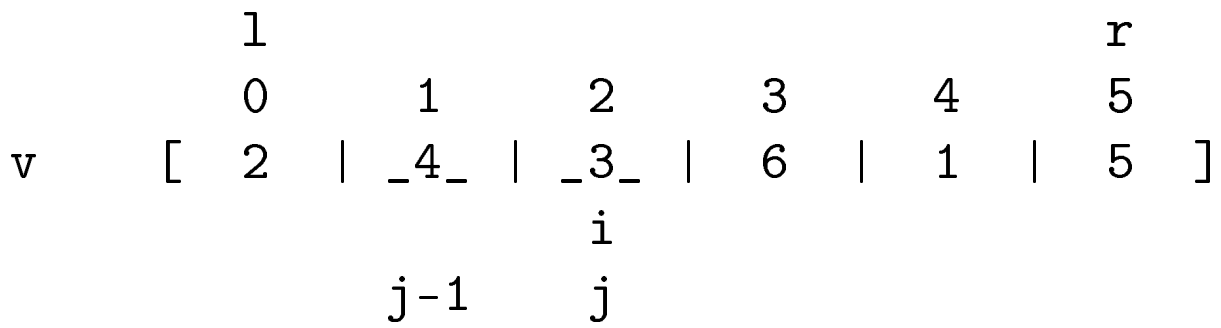
Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ? Insere (swap) e `j--`
- Elemento inserido não necessariamente na posição final

```
          l                                   r
          0       1       2       3       4       5
v     [ _2_ | _4_ |   3   |   6   |   1   |   5   ]
                  i
          j
```

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ?



Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ? Insere (swap) e `j--`
- Elemento inserido não necessariamente na posição final

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- i: posição inicial
- j: percorre antecessores
- Menor que o 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

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ?
- Não insere (sem swap) e `i++`

```
          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

- i: posição inicial
- j: percorre antecessores
- Menor que o 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

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ?
- Não insere (sem swap) e `i++`

```
          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

- i: posição inicial
- j: percorre antecessores
- Menor que o 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

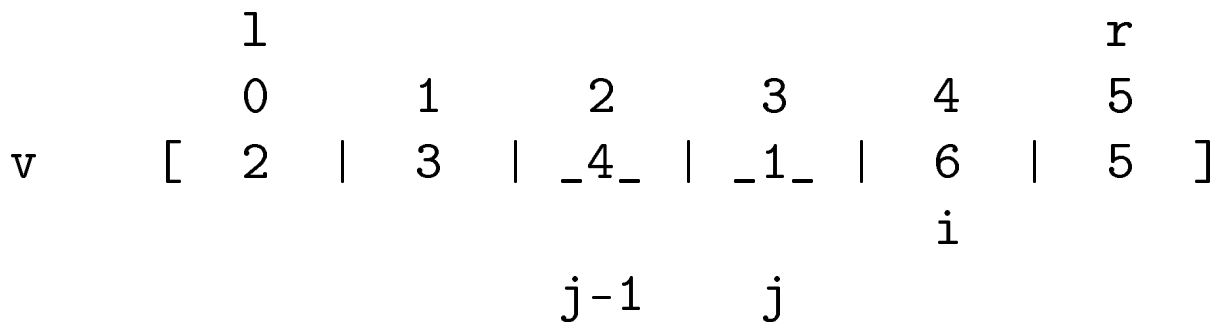
Insertion Sort - inserir na posição

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ? Insere (swap) e `j--`
- Elemento inserido não necessariamente na posição final

```
          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

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ?



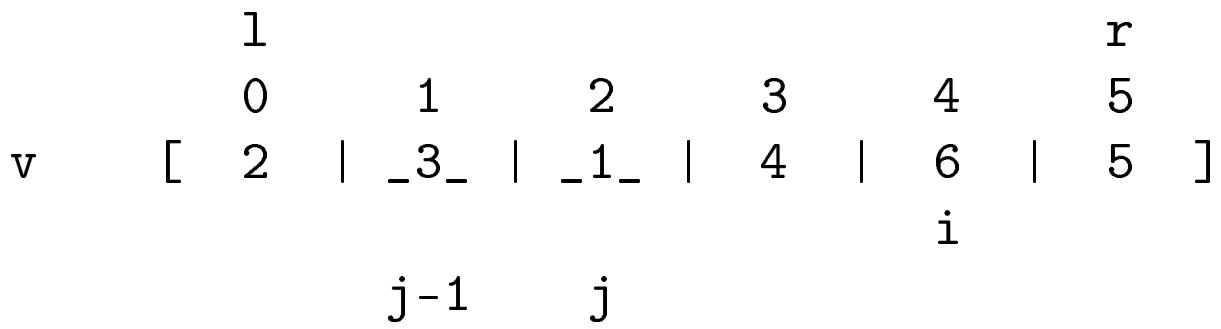
Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ? Insere (swap) e `j--`
- Elemento inserido não necessariamente na posição final

```
          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

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ?



Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ? Insere (swap) e `j--`
- Elemento inserido não necessariamente na posição final

```
          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

- i: posição inicial
- j: percorre antecessores
- Menor que o 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

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ? Insere (swap) e `j--`
- Elemento inserido não necessariamente na posição final

```
          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

- i: posição inicial
- j: percorre antecessores
- Menor que o 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

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ? Insere (swap) e `j--`
- Elemento inserido não necessariamente na posição final

```
          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

- i: posição inicial
- j: percorre antecessores
- Menor que o 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

- i: posição inicial
- j: percorre antecessores
- Menor que o antecessor (`v[ j ] < v[ j-1 ]`) ?
- Não insere (sem swap) e `i++`

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

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
```

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     for(int i=l+1; i<=r; i++)
5     {
6         //procurar antecessores menores que v[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     for(int i=l+1; i<=r; i++)
5     {
6         //procurar antecessores menores que v[j]
7         for(int j=i; j>l && v[j]<v[j-1]; j--)
8         {
9             //inserir na posição
```

Algoritmos de Ordenação Elementares
Insertion Sort

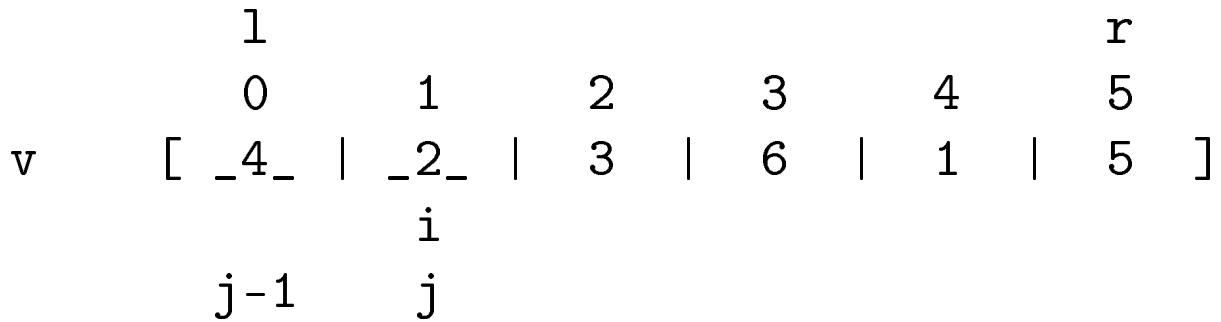
```
1  void insertion_sort(int v[], int l, int r)
2  {
3      //percorrer array a partir do segundo elemento
4      for(int i=l+1; i<=r; i++)
5      {
6          //procurar antecessores menores que v[j]
7          for(int j=i; j>l && v[j]<v[j-1]; j--)
8          {
9              //inserir na posição
10             exch(v[j], v[j-1])
11         }
12     }
13 }
14
```

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

- Puxar antecessores maiores (`v[j-1]`)
- Inserir na posição (`v[j]`)

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `2 < v[j-1]`?



Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `2 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`

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

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `2 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`
2. Inserir `v[j] = 2` e `i++`

```
          l                                   r
          0      1      2      3      4      5
v     [ _2_  | _4_  |  3   |  6   |  1   |  5   ]
                 i
          j
```

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `3 < v[j-1]`?

```
          l                                   r
          0      1      2      3      4      5
v     [   2   | _4_ | _3_ |   6   |   1   |   5   ]
                         i
                  j-1    j
```

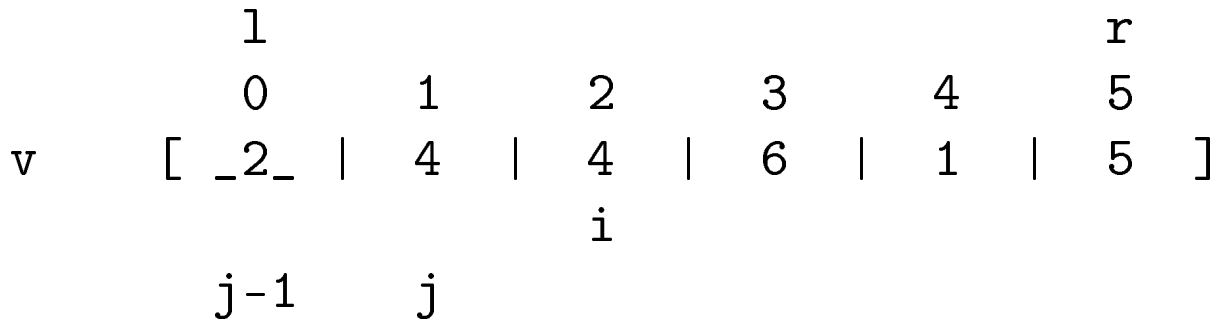
Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `3 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`

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

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `3 < v[j-1]`?
   - ▶



Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `3 < v[j-1]`?
   - Não

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

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `3 < v[j-1]`?
   - Não
2. Inserir `v[j] = 3` e `i++`

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

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `6 < v[j-1]`?

```
          l                                   r
          0      1      2      3      4      5
v     [   2   |  3   | _4_  | _6_  |  1   |  5   ]
                          i
                   j-1    j
```

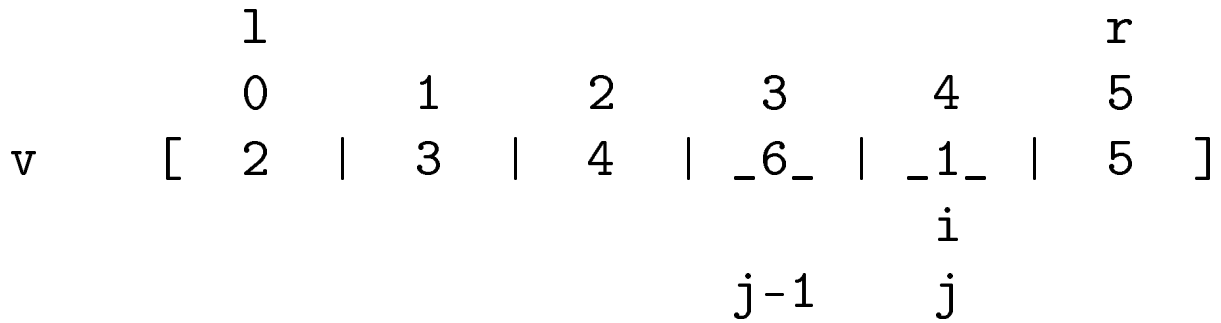
Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `6 < v[j-1]`?
   - Não
2. Inserir `v[j] = 6` e `i++`

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

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `1 < v[j-1]`?



Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `1 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`

```
          l                                       r
          0       1       2       3       4       5
v     [   2   |   3   | _4_   | _6_   | _6_   |   5   ]
                                          i
                        j-1     j
```

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `1 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`

```
          l                                   r
          0      1      2      3      4      5
v     [   2   | _3_ | _4_ | _4_ | _6_ |   5   ]
                                     i
                 j-1    j
```

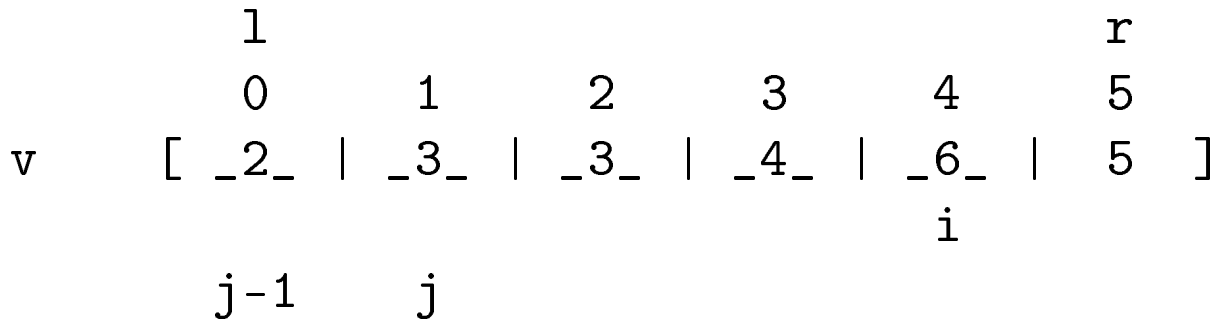
Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `1 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`

```
          l                                   r
          0      1      2      3      4      5
v     [   2   | _3_  | _3_  | _4_  | _6_  |  5   ]
                                      i
                 j-1    j
```

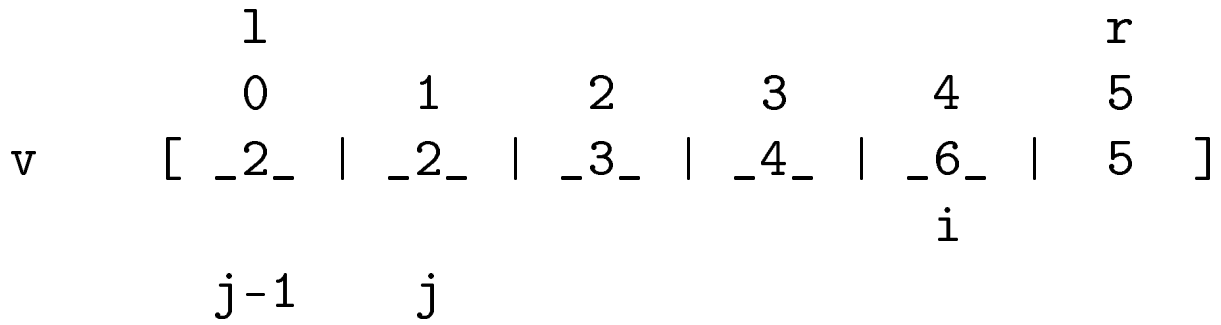
Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `1 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`



Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `1 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`



Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `1 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`
2. Inserir `v[j] = 1` e `i++`

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

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `5 < v[j-1]`?

```
          l                                   r
          0       1       2       3       4       5
v     [   1   |   2   |   3   |   4   | _6_   | _5_ ]
                                              i
                                    j-1       j
```

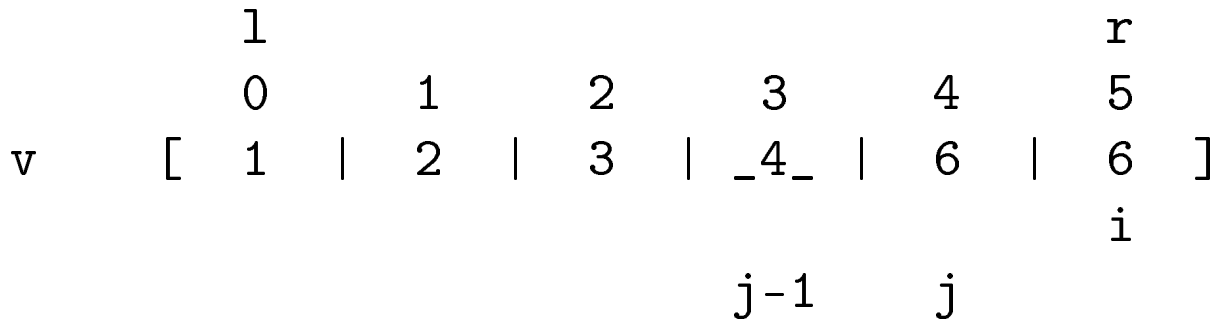
Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `5 < v[j-1]`?
   - `v[j] = v[j-1]` e `j--`

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

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `5 < v[j-1]`?
   - ▶



Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `5 < v[j-1]`?
   - Não

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

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

1. Puxar antecessores maiores que `v[i]`: `5 < v[j-1]`?
   - Não
2. Inserir `v[j] = 5` e `i++`

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

Algoritmos de Ordenação Elementares
Insertion Sort - otimizado

2. Fim: vetor ordenado

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

Insertion Sort - versão otimizada

```
1 //eliminar múltiplos swaps
2 void insertion_sort(int v[], int l, int r)
3 {
4     int elem, i, j;
5
6     //percorrer array a partir do segundo elemento
```

Insertion Sort - versão otimizada

```
1  //eliminar múltiplos swaps
2  void insertion_sort(int v[], int l, int r)
3  {
4      int elem, i, j;
5
6      //percorrer array a partir do segundo elemento
7      for(i=l+1; i<=r; i++)
8      {
9          //elemento que será (re)inserido
10         elem =
```

Insertion Sort - versão otimizada

```
1  //eliminar múltiplos swaps
2  void insertion_sort(int v[], int l, int r)
3  {
4      int elem, i, j;
5
6      //percorrer array a partir do segundo elemento
7      for(i=l+1; i<=r; i++)
8      {
9          //elemento que será (re)inserido
10         elem = v[i];
11
12         //para cada elemento maior
13         for(j=i; j>l && elem < v[j-1]; j--)
```

Insertion Sort - versão otimizada

```
1  //eliminar múltiplos swaps
2  void insertion_sort(int v[], int l, int r)
3  {
4      int elem, i, j;
5
6      //percorrer array a partir do segundo elemento
7      for(i=l+1; i<=r; i++)
8      {
9          //elemento que será (re)inserido
10         elem = v[i];
11
12         //para cada elemento maior
13         for(j=i; j>l && elem < v[j-1]; j--)
14             v[j] = v[j-1]; //"puxar" o maior
```

Insertion Sort - versão otimizada

```
1  //eliminar múltiplos swaps
2  void insertion_sort(int v[], int l, int r)
3  {
4      int elem, i, j;
5
6      //percorrer array a partir do segundo elemento
7      for(i=l+1; i<=r; i++)
8      {
9          //elemento que será (re)inserido
10         elem = v[i];
11
12         //para cada elemento maior
13         for(j=i; j>l && elem < v[j-1]; j--)
14             v[j] = v[j-1]; //"puxar" o maior
15
16         //inserir no elemento a sua posição
```

Insertion Sort - versão otimizada

```
1  //eliminar múltiplos swaps
2  void insertion_sort(int v[], int l, int r)
3  {
4      int elem, i, j;
5
6      //percorrer array a partir do segundo elemento
7      for(i=l+1; i<=r; i++)
8      {
9          //elemento que será (re)inserido
10         elem = v[i];
11
12         //para cada elemento maior
13         for(j=i; j>l && elem < v[j-1]; j--)
14             v[j] = v[j-1]; //"puxar" o maior
15
16         //inserir no elemento a sua posição
17         v[j] = elem;
18     }
19 }
```

Algoritmos de Ordenação Elementares - Insertion Sort

- Complexidade assintótica?
  - Pior caso $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ movimentações
  - Entradas invertidas
  - Não otimizado: desempenho do bubble
  - Otimizado: redução de acesso à memória (ciclo longo de instrução)

```
1  //não otimizado : (n^2)/2 + 3*(n^2)/2
2  for(int i=l+1; i<=r; i++) {
3      //1 2 3 ... (n-1) -> PA ((n-1+1)n)/2
4      //comparações (n^2)/2
5      for(int j=i; j>l && v[j]<v[j-1]; j--)
6      {
7          //trocas (n^2)/2
8          exch(v[j], v[j-1]); //≈ 3 movimentos
9      }
10 }
11
12 //otimizado : (n^2)/2 + (n^2)/2
13 for(i=l+1; i<=r; i++) {
14     elem = v[i];
15     //comparações (n^2)/2
16     for(j=i; j>l && elem < v[j-1]; j--) {
17         //(n^2)/2 movimentações
18         v[j] = v[j-1];
19     }
20     v[j] = elem;
21 }
```

Algoritmos de Ordenação Elementares - Insertion Sort

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

```
1 void insertion_sort(int v[], int l, int r) {
2     if(r<=l) return;
3     insertion_sort(v, n-1); //f(n-1)
4     for(int j=i; j>l && v[j]<v[j-1]; j--) //n-1
5         exch(v[j], v[j-1]);
6 }
```

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

Algoritmos de Ordenação Elementares

Insertion Sort

- Complexidade assintótica?
  - Pior caso $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ movimentos
  - Médio aprox. $\frac{N^2}{4}$ comparações e $\frac{N^2}{4}$ movimentos
  - Função custo das comparações:

$$
\begin{aligned}
f(n) &\approx f(n-1) + \frac{n-1}{n} + \frac{n-2}{n} + \ldots + \frac{1}{n} \\
&\approx f(n-1) + \frac{\frac{(n-1+1)n}{2}}{n} = f(n-1) + \frac{n}{2} \\
&\approx f(n-2) + \frac{n-1}{2} + \frac{n}{2} \\
&\approx f(n-3) + \frac{n-2}{2} + \frac{n-1}{2} + \frac{n}{2} \\
&\approx \ldots \\
&\approx f(n-i) + \frac{n-i+1}{2} + \frac{n-i+2}{2} + \ldots + \frac{n}{2} \\
&\approx f(0) + \frac{1}{2} + \frac{2}{2} + \ldots + \frac{n}{2} \\
&\approx \frac{\frac{(1+n)*n}{2}}{2} = \frac{n^2+n}{4}
\end{aligned}
$$

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Possível identificar uma ordenação?

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, pois como os antecessores estão ordenados, basta a comparação com 1 elemento para a decisão de continuar a percorrer ou não o vetor
  - Portanto, é adaptativo.

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, pois como os antecessores estão ordenados, basta a comparação com 1 elemento para a decisão de continuar a percorrer ou não o vetor
  - Portanto, é adaptativo.
- Complexidade assintótica?
  - Pior caso: $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ movimentos
  - Médio aprox.: $\frac{N^2}{4}$ comparações e $\frac{N^2}{4}$ movimentos
  - Melhor caso: $O(N)$ (quando?)

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, pois como os antecessores estão ordenados, basta a comparação com 1 elemento para a decisão de continuar a percorrer ou não o vetor
  - Portanto, é adaptativo.
- Complexidade assintótica?
  - Pior caso: $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ movimentos
  - Médio aprox.: $\frac{N^2}{4}$ comparações e $\frac{N^2}{4}$ movimentos
  - Melhor caso: $O(N)$ (quando?)
- Adaptatividade x Custo: cada elemento é posicionado até encontrar um menor (decrescente) no sub-conjunto dos antecessores, não sendo necessário a comparação com todos os elementos

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes
    1. 3 4 2 5 2' $\rightarrow$ 3 3 4 5 2' $\rightarrow$ 2 3 4 5 2'

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes
    1. 3 4 2 5 2' $\rightarrow$ 3 3 4 5 2' $\rightarrow$ 2 3 4 5 2'
    2. 2 3 4 5 2' $\rightarrow$ 2 3 3 4 5 $\rightarrow$ 2 2' 3 4 5

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes
    1. 3 4 2 5 2' $\rightarrow$ 3 3 4 5 2' $\rightarrow$ 2 3 4 5 2'
    2. 2 3 4 5 2' $\rightarrow$ 2 3 3 4 5 $\rightarrow$ 2 2' 3 4 5
    3. 2 2' 3 4 5

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes
    1. 3 4 2 5 2' $\rightarrow$ 3 3 4 5 2' $\rightarrow$ 2 3 4 5 2'
    2. 2 3 4 5 2' $\rightarrow$ 2 3 3 4 5 $\rightarrow$ 2 2' 3 4 5
    3. 2 2' 3 4 5
  - Mantém a ordem (não trocar os iguais): estável.

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes
    1. 3 4 2 5 2' $\rightarrow$ 3 3 4 5 2' $\rightarrow$ 2 3 4 5 2'
    2. 2 3 4 5 2' $\rightarrow$ 2 3 3 4 5 $\rightarrow$ 2 2' 3 4 5
    3. 2 2' 3 4 5
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes
    1. 3 4 2 5 2' $\rightarrow$ 3 3 4 5 2' $\rightarrow$ 2 3 4 5 2'
    2. 2 3 4 5 2' $\rightarrow$ 2 3 3 4 5 $\rightarrow$ 2 2' 3 4 5
    3. 2 2' 3 4 5
  - Mantém a ordem (não trocar os iguais): estável.
- *In-place*?
  - Utiliza memória extra significativa?

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes
    1. 3 4 2 5 2' → 3 3 4 5 2' → 2 3 4 5 2'
    2. 2 3 4 5 2' → 2 3 3 4 5 → 2 2' 3 4 5
    3. 2 2' 3 4 5
  - 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?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes
    1. 3 4 2 5 2' $\rightarrow$ 3 3 4 5 2' $\rightarrow$ 2 3 4 5 2'
    2. 2 3 4 5 2' $\rightarrow$ 2 3 3 4 5 $\rightarrow$ 2 2' 3 4 5
    3. 2 2' 3 4 5
  - 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

- Estabilidade?
  - Tem trocas com saltos? Comparação com adjacente
  - E o otimizado? Desloca um por um, sem trocas distantes
    1. 3 4 2 5 2' → 3 3 4 5 2' → 2 3 4 5 2'
    2. 2 3 4 5 2' → 2 3 3 4 5 → 2 2' 3 4 5
    3. 2 2' 3 4 5
  - 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*
  - Complexidade espacial auxiliar constante

Algoritmos de Ordenação Elementares
Insertion Sort x Bubble sort

- Bubble:
  - Comparação: maior que o sucessor
  - O posicionamento de um item não garante a ordenação dos outros elementos
    - ⋆ Garante que os elementos à esquerda sejam menores
    - ⋆ Não necessariamente ordenados a cada passagem
  - Cada passagem: um elemento na posição final e um vetor mais ordenado
  - Adaptativo, estável, *in-place*
- Insertion:
  - Comparação: menor que o antecessor
  - O posicionamento de um item garante a ordenação dos elementos à sua esquerda
  - Cada passagem: não garante o item na sua posição final mas um sub-vetor ordenado
  - Adaptativo, estável, *in-place*

Algoritmos de Ordenação Elementares
Insertion Sort x Selection sort

- Selection:
  - Da posição atual:
    - ⋆ Itens à esquerda $\rightarrow$ ordenados e na posição final
  - Não-adaptativo, não-estável, *in-place*
- Insertion:
  - Da posição atual:
    - ⋆ Itens à esquerda $\rightarrow$ ordenados mas, não garante a posição final
  - Adaptativo, estável, *in-place*

Algoritmos de Ordenação Elementares
Selection x Bubble x Insertion

- Bubble:
  - Comparação item a item, flutuando o item, até encontrar um maior
  - Prosseguindo até o topo
  - $\frac{N^2}{2}$ comparações e $\frac{N^2}{2}$ trocas
- Selection:
  - Dada a posição, selecionamos o elemento
  - $\frac{N^2}{2}$ comparações e N trocas
- Insertion:
  - Dado o elemento, inserimos na sua posição do sub-vetor esquerdo
  - Para ao encontrar um menor
  - $\frac{N^2}{4}$ comparações e $\frac{N^2}{4}$ movimentações

Algoritmos de Ordenação Elementares
Selection x Bubble x Insertion

- Mais especificamente (atribuições):

$$
\begin{aligned}
Bubble &\approx \frac{n^2}{2} + 3\frac{n^2}{2} &&= 2n^2 \\
Selection &\approx \frac{n^2}{2} + \frac{n^2}{2} + 3n &&= n^2 + 3n \\
InsertionS &\approx \frac{n^2}{2} + 3\frac{n^2}{2} &&= 2n^2 \\
InsertionO &\approx \frac{n^2}{2} + \frac{n^2}{2} + 2n &&= n^2 + 2n \\
&\approx \frac{n^2}{4} + \frac{n^2}{4} + 2n &&= \frac{n^2}{2} + 2n
\end{aligned}
$$

- Variações nos tempos: otimizações de linguagem e compilação (O2, O3)
- Teste:
  - insertion simples x otimizado : 15-aleatorio
  - bubble x insertion : 16-quaseordenado
  - selection x insertion : 16-aleatorio e 16-quaseordenado
  - bubble x selection x insertion : 16-reverso

Algoritmos de Ordenação Elementares - Shell Sort

- Extensão do algoritmo de ordenação Insertion Sort
- Ideia:
  - Ordenação parcial a cada passagem
  - Ordenação de elementos distantes: colocando, possivelmente, mais perto da sua posição final
  - Posteriormente, eficientemente, ordenados pelo Insertion Sort
- Diminuir o número de movimentações
- Troca de itens que estão distantes um do outro
- Implementação é muito simples, similar ao algoritmo de inserção

Algoritmos de Ordenação Elementares - Shell Sort

- Troca de itens que estão distantes um do outro
  - Separados a h distância
  - São rearranjados, resultando uma sequência 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): taxa de crescimento cerca de 1/3
    - 3*h+1 = 1, 4, 13, 40, 121, até $\approx$ n/3
  - Sequências 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

Algoritmos de Ordenação Elementares
Shell Sort

```
Sequência dos valores de h
h = 1
h = 3*h+1 → alternar pares e ímpares - aumentar aleatoriedade
h = 1, 4, 13, 40, 121, 364, 1093, ...

Determinando o h inicial
r = 15 → 15/3 ~ 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 (inicial)
```

Algoritmos de Ordenação Elementares
Shell Sort

1. h = 13
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. h = 13
2. menor que o antecessor (`v[j] < v[j-h]`) ? não
3. `++i` e `j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. h = 13
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. h = 13
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. h = 13
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? não: `++i, j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. Fim
2. Atualizar `h = h/3`
3. `h = 13/3 = 4`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? não: `++i, j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? não
3. `++i` e `j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? não
3. `++i` e `j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? não: `++i, j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? não
3. `++i` e `j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? não
3. `++i` e `j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? sim: `j-=h`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? não: `++i, j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? sim: `j-=h`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? não
3. `++i` e `j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? sim: `j-=h`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? sim: `j-=h`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? não: ++i, j

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? sim: `j-=h`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? sim: `j-=h`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? não
3. `++i` e `j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? sim: `j-=h`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)?

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? sim: swap
3. + antecessores (`j-h > l+h`)? sim: `j-=h`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. `h = 13/3 = 4`
2. menor que o antecessor (`v[j] < v[j-h]`) ? não
3. `++i` e `j=i`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. Fim: atualizar `h = h/3`
2. `h = 13/3 = 4/3 = 1`

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

Algoritmos de Ordenação Elementares
Shell Sort

1. Fim: atualizar `h = h/3`
2. `h = 13/3 = 4/3 = 1`
3. Insertion sort

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

Algoritmos de Ordenação Elementares
Shell Sort

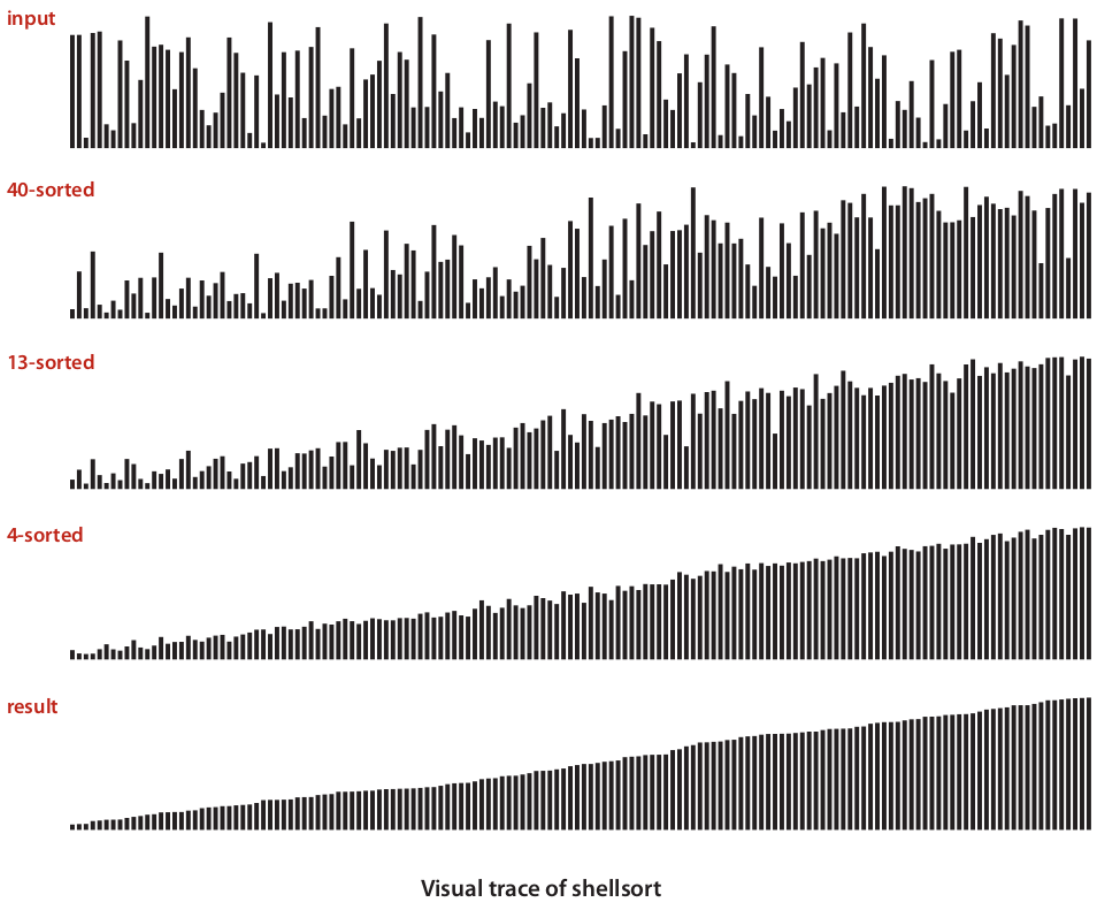


Figura: fonte: Algorithms - 4 edição, Robert Sedgewick e Kevin Wayne

```
1  void shell_sort(int v[], int l, int r)
2  {
3      int h = 1; //h-distância
4
5      //calcular o máximo h
6      while(h < (r-l+1)/3) //(r-l+1)/9
7          h = 3*h+1;
8
9      while(h>=1){
10         for(int i=l+h; i<=r; i++)
11         {
12             for(int j=i; j>=l+h && v[j]<v[j-h]; j-=h)
13             {
14                 exch(v[j], v[j-h])
15             }
16         }
17         h = h/3;
18     }
19 }
```

Shell Sort - otimizado

```
1  void shell_sort(int v[], int l, int r)
2  {
3      int h = 1, elem, i, j;
4      while(h < (r-l+1)/3) h = 3*h+1;
5
6      while(h>=1){
7          for(i=l+h; i<=r; i++)
8          {
9              elem = v[i];
10             for(j=i; j>=l+h && elem<v[j-h]; j-=h)
11             {
12                 v[j] = v[j-h];
13             }
14             v[j] = elem;
15         }
16         h = h/3;
17     }
18 }
```

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
    - ★ 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)
    - ⋆ As comparações são proporcionais a $N^{3/2}$
    - ⋆ Pior caso com pior h: $O(n^2)$
    - ⋆ Melhor caso com pior 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 $N^{3/2}$
    - ⋆ Pior caso com pior h: $O(n^2)$
    - ⋆ Melhor caso com pior h: $O(nlog^2n)$ (Pratt, Vaughan Ronald (1979). Shellsort and Sorting Networks - Outstanding Dissertations in the Computer Sciences)
  - Melhor caso com um bom 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 $N^{3/2}$
    - ⋆ Pior caso com pior h: $O(n^2)$
    - ⋆ Melhor caso com pior h: $O(nlog^2n)$ (Pratt, Vaughan Ronald (1979). Shellsort and Sorting Networks - Outstanding Dissertations in the Computer Sciences)
  - Melhor caso com um bom 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?
  - Possível identificar uma ordenação?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- Estabilidade?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- Estabilidade?
  - 2 3 2' 1 $\rightarrow$ mantém a ordem relativa? h=3
  - Tem trocas com saltos?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- 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?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- 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?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- 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**
  - Sim, portanto, não é estável

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- 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**
  - Sim, portanto, não é estável
- *In-place*?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- 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**
  - Sim, portanto, não é estável
- *In-place*?
  - Utiliza memória extra significativa?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- 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**
  - Sim, portanto, não é estável
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- 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**
  - Sim, portanto, não é estável
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?
  - É in-place

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- 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**
  - Sim, portanto, não é estável
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?
  - É in-place
- Teste:

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Possível identificar uma ordenação?
  - Sim, com interrupção ao identificar
  - Portanto, é adaptativo
- 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**
  - Sim, portanto, não é estável
- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?
  - É in-place
- Teste:
  - insertion (16-aleatorio) x shell (21-aleatorio)