Ordenação de dados

Profª. Rose Yuri Shimizu

Roteiro

Ordenação de dados - importância

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

Recomendações

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

Roteiro

Algoritmos de Ordenação - Características

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

Algoritmos de Ordenação - Características

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

Algoritmos de Ordenação - Características

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

Algoritmos de Ordenação - Características

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

Algoritmos de Ordenação - Características

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

Algoritmos de Ordenação - Elementares x Eficientes

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

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

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

Roteiro

Roteiro

Algoritmos de Ordenação Elementares
Selection Sort - selecionar e posicionar

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

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

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

Selection Sort

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

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

Algoritmos de Ordenação Elementares
Selection Sort

- Terminou? Vetor ordenado.

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

Algoritmos de Ordenação Elementares
Selection Sort

```c
void selection_sort(int v[], int l, int r){
    int menor;
    for(int i=l; i<r; i++){
        menor = i;

        for(int j=i+1; j<=r; j++)
            if(v[j] < v[menor])
                menor = j;

        if(i != menor)
            exch(v[i], v[menor])
    }
}
```

Algoritmos de Ordenação Elementares
Selection Sort

```c
void selection_sort(int v[], int l, int r){
    int menor;
    //n
    for(int i=l; i<r; i++){
        menor = i;

        //(n-1), (n-2), (n-3), .. , 0
        //PA ((n+0)n)/2 = (n^2)/2
        for(int j=i+1; j<=r; j++)
            if(v[j] < v[menor])
                menor = j;

        if(i != menor)
            exch(v[i], v[menor]) //n
    }
    //f(n) = (n^2)/2 + n
}
```

- Complexidade assintótica?

- Adaptatividade?

- Estabilidade?

- *In-place*?

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

- Selection Sort estável??

- Selection Sort com listas encadeadas??

Algoritmos de Ordenação Elementares
Selection Sort

- Selection Sort estável??
  - Não realizar o swap

- Selection Sort com listas encadeadas??

Algoritmos de Ordenação Elementares
Selection Sort

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

Algoritmos de Ordenação Elementares
Selection Sort

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

Roteiro

1. Ordenação de dados
2. Algoritmos de Ordenação
   - Algoritmos de Ordenação Elementares
     - Selection Sort
     - Bubble Sort
     - Insertion Sort
   - Algoritmos de Ordenação Eficientes
     - Merge Sort
     - Quick Sort

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort - flutue o maior

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

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

Algoritmos de Ordenação Elementares
Bubble Sort

```
void bubble_sort(int v[], int l, int r){

    for(; r>l; r--) {
        for(int j=l; j<r; j++) {
            if(v[j] > v[j+1]) {
                exch(v[j], v[j+1])
            }
        }
    }

}
```

Algoritmos de Ordenação Elementares
Bubble Sort

```c
void bubble_sort(int v[], int l, int r){
    //n
    for(; r>l; r--) {

        //(n-1), (n-2), (n-3), .. , 0
        //PA ((n+0)n)/2 = (n^2)/2
        for(int j=l; j<r; j++) {

            if(v[j] > v[j+1]) {

                //(n-1), (n-2), (n-3), .. , 0
                //PA ((n+0)n)/2 = (n^2)/2
                exch(v[j], v[j+1])
            }
        }
    }
    //f(n) = (n^2)/2 + (n^2)/2
}
```

Algoritmos de Ordenação Elementares
Bubble Sort

- Complexidade assintótica?

- Adaptatividade?

- Estabilidade?

- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

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

Algoritmos de Ordenação Elementares
Bubble Sort

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

- Adaptatividade?
  - Ordenação diminui processamento?

- Estabilidade?

- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

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

- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.

- Estabilidade?

- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

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

- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.

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

- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

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

Algoritmos de Ordenação Elementares
Bubble Sort

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

- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.

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

- *In-place*?

Algoritmos de Ordenação Elementares
Bubble Sort

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

Algoritmos de Ordenação Elementares
Bubble Sort

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

Algoritmos de Ordenação Elementares
Bubble Sort

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

- Adaptatividade?
  - Ordenação diminui processamento?
  - Sim, portanto, é adaptativo.

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

- *In-place*?
  - Utiliza memória extra significativa?
  - Copia os conteúdos para outra estrutura de dados?

Algoritmos de Ordenação Elementares
Bubble Sort

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

Algoritmos de Ordenação Elementares
Bubble Sort

```
void bubble_sort(int v[], int l, int r){
    int swap = 1;
    for(; r>l && swap; r--) {
        swap = 0;
        for(int j=l; j<r; j++) {
            if(v[j] > v[j+1]) {
                exch(v[j], v[j+1])
                swap = 1;
            }
        }
    }
}
```

Algoritmos de Ordenação Elementares
Bubble Sort

- Selection Sort x Bubble sort?

- Bubble Sort com listas encadeadas??

- Otimização?

Algoritmos de Ordenação Elementares
Bubble Sort

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

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

Algoritmos de Ordenação Elementares
Bubble Sort

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

Roteiro

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Insertion Sort - inserir na posição

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

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

Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort - inserir na posição

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

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

Algoritmos de Ordenação Elementares
Insertion Sort

```
void insertion_sort(int v[], int l, int r)
{
    PERCORRER ARRAY A PARTIR DO SEGUNDO ELEMENTO






}

```

Algoritmos de Ordenação Elementares
Insertion Sort

```
void insertion_sort(int v[], int l, int r)
{
    for(int i=l+1; i<=r; i++)
    {
        PROCURANDO ANTECESSORES MENORES QUE V[J]



    }
}

```

Algoritmos de Ordenação Elementares
Insertion Sort

```
void insertion_sort(int v[], int l, int r)
{
    for(int i=l+1; i<=r; i++)
    {
        for(int j=i; j>l && v[j]<v[j-1]; j--)
        {
            INSERINDO NA POSICAO
        }
    }
}

```

Algoritmos de Ordenação Elementares
Insertion Sort

```c
void insertion_sort(int v[], int l, int r)
{
    for(int i=l+1; i<=r; i++)
    {
        for(int j=i; j>l && v[j]<v[j-1]; j--)
        {
            exch(v[j], v[j-1]);
        }
    }
}
```

Algoritmos de Ordenação Elementares
Insertion Sort - versão otimizada 1

```
void insertion_sort(int v[], int l, int r)
{
    int elem, i, j;
    for(i=l+1; i<=r; i++)
    {
        elem = v[i];
        for(j=i; j>l && elem < v[j-1]; j--)
            v[j] = v[j-1]; //puxando o maior

        v[j] = elem; //encaixando o elemento
    }
}

```

Algoritmos de Ordenação Elementares
Insertion Sort - versão otimizada 2

```c
void insertion_sort(int v[], int l, int r) {
  int elem, i, j;
  //empurre o menor para a esquerda (sentinela),
  //enquanto puxa os maiores para a direita
  for(i=r; i>l; i--) compexch(v[l],v[i]);

  //a partir do terceiro elemento
  for(i=l+2; i<=r; i++) {
    elem = v[i];
    //antecessores, até o sentinela
    for(j=i; elem < v[j-1]; j--)
       v[j] = v[j-1]; //puxando o maior

    v[j] = elem; //encaixando o elemento
  }
}
```

Algoritmos de Ordenação Elementares

Insertion Sort

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

```c
void insertion_sort(int v[], int l, int r)
{
    for(int i=l+1; i<=r; i++)
    {
        //1 2 3 ... (n-1)
        //PA ((n-1+1)n)/2 = (n^2)/2
        for(int j=i; j>l && v[j]<v[j-1]; j--)
        {
            exch(v[j], v[j-1]);
        }
    }
}

```

Algoritmos de Ordenação Elementares
Insertion Sort

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

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?

Algoritmos de Ordenação Elementares
Insertion Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

- Estabilidade?

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

Algoritmos de Ordenação Elementares
Insertion Sort

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

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

Algoritmos de Ordenação Elementares
Insertion Sort x Bubble sort

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

Algoritmos de Ordenação Elementares
Insertion Sort x Selection sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Shell Sort

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

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

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

Algoritmos de Ordenação Elementares
Shell Sort

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

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

Algoritmos de Ordenação Elementares
Shell Sort

```
h = 13/3 ~ 4

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

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

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

Algoritmos de Ordenação Elementares
Shell Sort

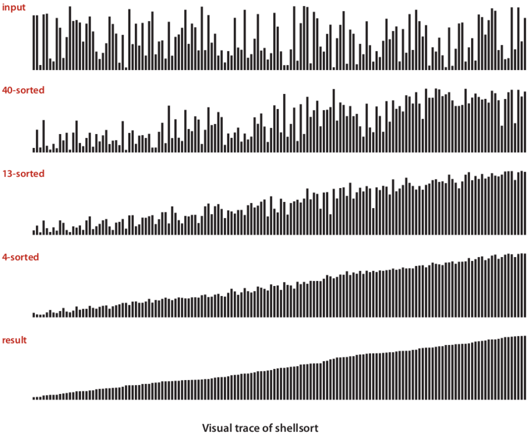


Figura: fonte: Algorithms – 4 edição, Robert Sedgewick e Kevin Wayne

Algoritmos de Ordenação Elementares - Shell Sort

```
void shell_sort(int v[], int l, int r)
{
    int h = 1;
    while(h < (r-l+1)/3) h = 3*h+1;

    while(h>=1){
        for(int i=l+h; i<=r; i++)
        {
            for(int j=i; j>=l+h && v[j]<v[j-h]; j-=h)
            {
                exch(v[j], v[j-h])
            }
        }
        h = h/3;
    }
}
```

Algoritmos de Ordenação Elementares
Shell Sort

- Complexidade assintótica ?

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?

Algoritmos de Ordenação Elementares
Shell Sort

- Adaptatividade?
  - Ordenação diminui comparações/trocas?

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Algoritmos de Ordenação Elementares
Shell Sort

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

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

Algoritmos de Ordenação Elementares
Shell Sort

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

Roteiro

Roteiro

Algoritmos de Ordenação Eficientes - Merge Sort

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[        ]
 k
```

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

```
[2      ]
 k
```

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2	7]
	k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

```
[2   7]
 k
```

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[ ]

k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2]
k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2		]
	k	

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2	7	]
	k	

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2	7	9]
		k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2	7	9]
		k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[ ]

k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

                 [4        ]
                  k
```

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[4	10]
	k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[4 10]
k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[ ]

k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2				]
k				

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2				]
	k			

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2	4			]
	k			

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2	4			]
		k		

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2	4	7		]
			k	

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2	4	7	9	]
			k	

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[2	4	7	9	10]
				k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

```
[         ]
k
```

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[1	3]
k	

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

```
[1   3]
 k
```

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[ ]

k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[1]

k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[1	3	8]
		k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[1	3	8]
		k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[5	6]
k	

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[ ]

k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[1	3			]
		k		

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[1	3	5	6	]
				k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

[1	3	5	6	8]
				k

Algoritmos de Ordenação Eficientes - Merge Sort

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

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



[                                                                                       ]
 k
```

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

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

Algoritmos de Ordenação Eficientes - Merge Sort

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

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

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

Algoritmos de Ordenação Eficientes

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

           6 **5** 3 1 2 4 : 5 6
    3. `merge_sort(v, 2, 2) : direita`
    4. `merge(v, 0, 1, 2)`

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

       5 6 3 1 2 4 : 3 5

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

Algoritmos de Ordenação Eficientes

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

         3 5 6 **1** **2** 4 : 1 2
   3. `merge_sort(v, 5, 5) : direita`
   4. `merge(v, 3, 4, 5)`

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

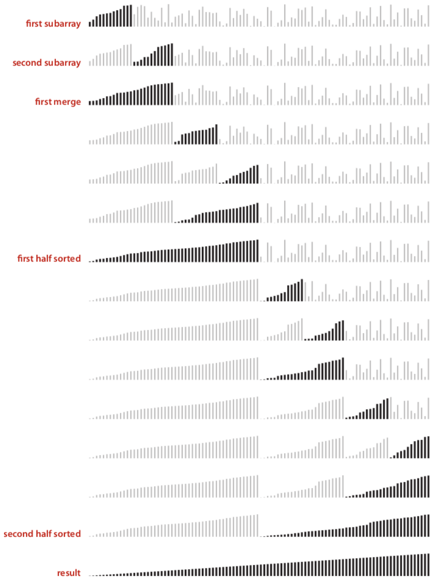
      3 5 6 1 **2** **4** : 1 2

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

Algoritmos de Ordenação Eficientes

5. `merge(v, 0, 2, 5)`
   3 5 6 **1** 2 4 : 1
   3 5 6 1 **2** 4 : 1 2
   3 5 6 1 2 **4** : 1 2 3
   3 **5** 6 1 2 **4** : 1 2 3 4
   3 **5** 6 1 2 4 : 1 2 3 4 5
   3 5 **6** 1 2 4 : 1 2 3 4 5
   3 5 6 1 2 4 : 1 2 3 4 5 6

Algoritmos de Ordenação Eficientes



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

Algoritmos de Ordenação Eficientes

```c
void merge_sort(int *v, int l, int r)
{
    if (l >= r) return;
    int m = (r+l)/2;

    merge_sort(v, l, m);
    merge_sort(v, m+1, r);
    merge(v, l, m, r);
}
```

Algoritmos de Ordenação Eficientes

```c
void merge(int *v,  int l, int m, int r) {
  //l=0  r=9 -> 10 itens = 9+1-0
  //l=2  r=8 -> 7 itens = 8+1-2
  int tam = r+1-l;

  //alocar espaço auxiliar
  int *aux = malloc(sizeof(int)*tam);

  int i=l;   //inicio do sub-vetor esquerdo
  int j=m+1; //inicio do sub-vetor direito
  int k=0;   //inicio do vetor auxiliar

  while(i<=m && j<=r) { //percorrer os sub-vetores
    if(v[i] <= v[j])    //testar sub-vetores
      aux[k++] = v[i++];//ordenar no vetor auxiliar
    else
      aux[k++] = v[j++];//ordenar no vetor auxiliar
  }
```

Algoritmos de Ordenação Eficientes

```
  //ainda tem elementos no sub-vetor esquerdo?
  while(i<=m) aux[k++] = v[i++];

  //ainda tem elementos no sub-vetor direito?
  while(j<=r) aux[k++] = v[j++];


  k=0; //indice do aux
  for(i=l; i<=r; i++) //indice do v
    v[i] = aux[k++];  //copiar o aux[k] para v[i]

  //liberar memória
  free(aux);
}
```

Algoritmos de Ordenação Eficientes

```c
void merge(int *v, int l, int m, int r) {
  //quanto elementos?
  int tam = r+1-l;

  //alocar espaço auxiliar
  int *aux = malloc(tam*sizeof(int));

  int i=l; //inicio do sub-vetor esquerdo
  int j=m+1; //inicio do sub-vetor direito
  int k=0; //inicio do vetor auxiliar

  //ordenar em aux[k]
  while(k<tam) { //condição de parada do aux

    if(i>m) //ordenou todo o primeiro sub-vetor
      aux[k++] = v[j++];//consome o segundo sub-vetor


```

Algoritmos de Ordenação Eficientes

```
    else if (j>r) //ordenou todo o segundo sub-vetor
      aux[k++] = v[i++];//consome o primeiro sub-vetor

    else if (v[i] < v[j]) //testar sub-vetores
      aux[k++] = v[i++];//ordene no aux

    else
      aux[k++] = v[j++];//ordene no aux
  }

  k=0; //indice do aux
  for(i=l; i<=r; i++) //indice do v
    v[i] = aux[k++];//copiar o aux[k] para v[i]

  //liberar memória
  free(aux);
}
```

Algoritmos de Ordenação Eficientes

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

```
void merge(int *v, int l, int m, int r) {
    ...

    while(k<r+1-l) { ... }  //n

}
```

```
void merge_sort(int *v, int l, int r) {
    if (l >= r) return;
    int m = (r+l)/2;

    merge_sort(v, l, m);  //F(n/2)
    merge_sort(v, m+1, r); //F(n/2)
    merge(v, l, m, r); //n
}
```

Algoritmos de Ordenação Eficientes

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

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

Algoritmos de Ordenação Eficientes

- In-place?

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

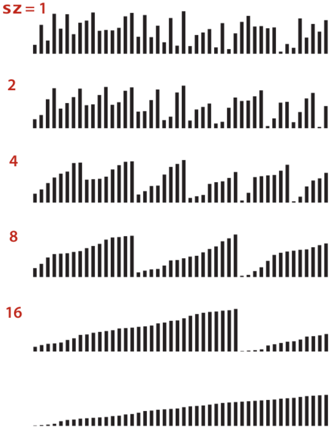
Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes



Visual trace of bottom-up mergesort

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

Algoritmos de Ordenação Eficientes

```c
void mergeBU_sort(int *v, int l, int r)
{
    int tam = (r-l+1);

    for (int sz=1; sz<tam; sz=2*sz)
    {
        for (int lo=l; lo<tam-sz; lo+=2*sz)
        {
            int hi = lo+2*sz-1;
            if(hi>tam-1) hi = tam-1;

            merge(v, lo, lo+sz-1, hi);
        }
    }
}
```

Algoritmos de Ordenação Eficientes

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

Algoritmos de Ordenação Eficientes

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

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

Algoritmos de Ordenação Eficientes

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

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

Algoritmos de Ordenação Eficientes

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

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

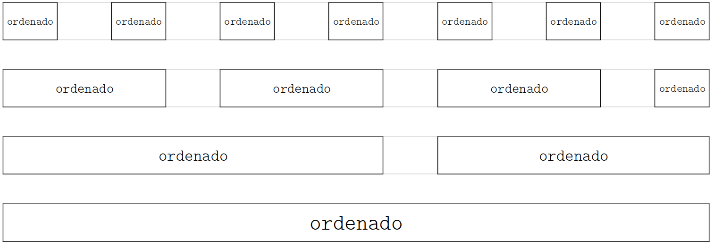
Roteiro

Algoritmos de Ordenação Eficientes - Quick Sort

- Um dos mais utilizados
- Simples
- Eficiente
- Muito pesquisado
  - Bem embasado
  - Bem comprovado

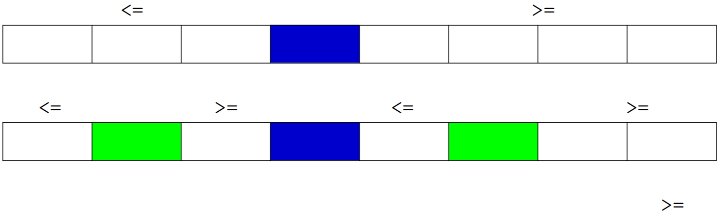
Algoritmos de Ordenação Eficientes - Quick Sort

- Método dividir e conquistar
- Particiona o vetor em sub-vetores
- Ordenando cada sub-vetor independentemente
- Merge x Quick
  - merge:
    - ★ Divide
    - ★ Ordena separadamente
    - ★ Combina reordenando
    - ★ Conquista um vetor mais ordenado



Algoritmos de Ordenação Eficientes - Quick Sort

- Método dividir e conquistar
- Particiona o vetor em sub-vetores
- Ordenando cada sub-vetor independentemente
- Merge x Quick
  - merge:
    - Divide
    - Ordena separadamente
    - Combina reordenando
    - Conquista um vetor mais ordenado
  - quick:
    - Separa os elementos baseados em 1 elemento
    - Conquista um elemento ordenado e dois sub-vetores pseudo-ordenados
    - Divide e repete para os sub-vetores



Algoritmos de Ordenação Eficientes - Quick Sort

- Ideia:
  - Particionar (separar) - processo crucial no quick
  - Escolhar um elemento de referência: pivô
  - Reorganizar os elementos de acordo com o pivô
  - Pivô na posição final
  - Pivô marca a divisão dos sub-vetores
  - Repetir o processo até ordenar todos os elementos
- Condições que devem ser satisfeitas:
  - O elemento a[j] está na sua posição final no vetor, para algum j
  - Nenhum elemento anterior ao a[j] é maior do que o a[j]
  - Nenhum elemento posterior ao a[j] é menor do que o a[j]

<= >=

[2 1 3 4 5]

Algoritmos de Ordenação Eficientes - Quick Sort

- **particionar** : pivô (+ direita) e rearranja
- dividir

[3 1 4 2 4]

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô?

```
[3    1    4    2    4]
 i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô?

```
[3    1    4    2    4]
      i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô?

```
[3    1    4    2    4]
           i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor que o pivô
- v[j] >= pivô?

[3	1	4	2	4]
		i		
				j

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor que o pivô
- v[j] >= pivô?

```
[3    1    4    2    4]
               i
                    j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- v[j] >= pivô? não
- i<j?

```
[3    1    4    2    4]
          i
               j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- i<j? sim
- swap v[i] $\leftrightarrow$ v[j]

```
[3    1    2    4    4]
               i
                    j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô?

```
[3    1    2    4    4]
           i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô?

```
[3    1    2    4    4]
                i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor que o pivô
- v[j] >= pivô?

```
[3    1    2    4    4]
                i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor que o pivô
- v[j] >= pivô?

```
[3    1    2    4    4]
                i
           j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- v[j] >= pivô? não
- i<j?

```
[3    1    2    4    4]
                i
           j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- i<j? não
- swap último maior $\leftrightarrow$ pivô

```
[3    1    2    4    4]
                i
           j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- swap v[i] $\leftrightarrow$ pivô
- i $\rightarrow$ pivô na sua posição final

```
[3    1    2    4    4]
                i=pivô
          j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- particionar : pivô + rearranjar
- **dividir**

[3 1 2 4 4]

Algoritmos de Ordenação Eficientes - Quick Sort

- **particionar** : pivô (+ direita) e rearranja
- dividir

[3 1 2 4 4]

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô?

```
[3    1    2    4    4]
 i
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor que o pivô
- v[j] >= pivô?

```
[3    1    2    4    4]
 i
               j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor que o pivô
- v[j] >= pivô?

```
[3    1    2    4    4]
 i
      j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- v[j] >= pivô? não
- i<j?

```
[3    1    2    4    4]
 i
      j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- i<j? sim
- swap v[i] $\leftrightarrow$ v[j]

```
[1    3    2    4    4]
 i
      j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô?

```
[1   3   2   4   4]
 i
     j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento maior ou igual que o pivô
- v[i] < pivô?

```
[1    3    2    4    4]
      i
      j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor que o pivô
- v[j] >= pivô?

```
[1    3    2    4    4]
      i
      j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- procurando um elemento menor que o pivô
- v[j] >= pivô?

```
[1    3    2    4    4]
      i
 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- v[j] >= pivô? não
- i<j?

```
[1    3    2    4    4]
      i
 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- i<j? não
- swap último maior $\leftrightarrow$ pivô

```
[1    3    2    4    4]
      i
 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- swap v[i] $\leftrightarrow$ pivô
- i $\rightarrow$ pivô na sua posição final

```
[1     2     3     4     4]
       i=pivô
 j
```

Algoritmos de Ordenação Eficientes - Quick Sort

- particionar : pivô + rearranjar
- **dividir**

[1 2 3 4 4]

p-1<=l r<=p+1

Algoritmos de Ordenação Eficientes - Quick Sort

- **particionar** : pivô (+ direita) e rearranja
- dividir

[1 2 3 4 4]

p-1<=l r<=p+1

Algoritmos de Ordenação Eficientes - Quick Sort

- particionar : pivô + rearranjar
- **dividir**

[1 2 3 4 4]

Algoritmos de Ordenação Eficientes - Quick Sort

- **particionar** : pivô (+ direita) e rearranja
- dividir

[1 2 3 4 4]

r<=p+1

Algoritmos de Ordenação Eficientes - Quick Sort

[1 2 3 4 4]

Algoritmos de Ordenação Eficientes - Quick Sort

Algoritmo do Quicksort

1. `quick_sort(v, l, r)`
   1. `p = partition(v, l, r)`
   2. `quick_sort(v, l, p-1)`
   3. `quick_sort(v, p+1, r)`

Vamos executar as chamadas.

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**

Algoritmos de Ordenação Eficientes - Quick Sort

quick_sort(v, 0, 4) : 5 2 4 1 3

1. p = partition(v, 0, 4)
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. quick_sort(v, 0, 1) : 1 2 3 5 4
   1. p = partition(v, 0, 1)
      - 1 **2**
   2. quick_sort(v, 0, 0) : **1**

Algoritmos de Ordenação Eficientes - Quick Sort

quick_sort(v, 0, 4) : 5 2 4 1 3

1. p = partition(v, 0, 4)
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. quick_sort(v, 0, 1) : 1 2 3 5 4
   1. p = partition(v, 0, 1)
      - 1 **2**
   2. quick_sort(v, 0, 0) : **1**
   3. quick_sort(v, 2, 1)

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**
      - **4** 5

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**
      - **4** 5
   2. `quick_sort(v, 3, 2)`

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**
      - **4** 5
   2. `quick_sort(v, 3, 2)`
   3. `quick_sort(v, 4, 4)` : **5**

Algoritmos de Ordenação Eficientes - Quick Sort

`quick_sort(v, 0, 4)` : 5 2 4 1 3

1. `p = partition(v, 0, 4)`
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. `quick_sort(v, 0, 1)` : 1 2 3 5 4
   1. `p = partition(v, 0, 1)`
      - 1 **2**
   2. `quick_sort(v, 0, 0)` : **1**
   3. `quick_sort(v, 2, 1)`
   4. 1 2
4. **1 2 3** 5 4
5. `quick_sort(v, 3, 4)` : 1 2 3 5 4
   1. `p = partition(v, 3, 4)`
      - 5 **4**
      - **4** 5
   2. `quick_sort(v, 3, 2)`
   3. `quick_sort(v, 4, 4)` : **5**
   4. 4 5

Algoritmos de Ordenação Eficientes - Quick Sort

quick_sort(v, 0, 4) : 5 2 4 1 3

1. p = partition(v, 0, 4)
   - 5 2 4 1 **3**
   - 1 2 4 5 **3**
   - 1 2 **3** 5 4
2. 1 2 **3** 5 4
3. quick_sort(v, 0, 1) : 1 2 3 5 4
   1. p = partition(v, 0, 1)
      - 1 **2**
   2. quick_sort(v, 0, 0) : **1**
   3. quick_sort(v, 2, 1)
   4. 1 2
4. **1 2 3** 5 4
5. quick_sort(v, 3, 4) : 1 2 3 5 4
   1. p = partition(v, 3, 4)
      - 5 **4**
      - **4** 5
   2. quick_sort(v, 3, 2)
   3. quick_sort(v, 4, 4) : **5**
   4. 4 5
6. **1 2 3 4 5**

Algoritmos de Ordenação Eficientes - Quick Sort

```
void quick_sort(int *v,  int l, int r)
{
    if(r<=l) return;

    int p = partitionRSEDGEWICK(v, l, r);

    quick_sort(v, l, p-1);
    quick_sort(v, p+1, r);
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

```
int partitionRSEDGEWICK(int *v, int l, int r)
{
    int i = l-1;
    int j = r;
    int pivot = v[r];

    PROCURAR MAIOR E MENOR ENQUANTO...









}
```

Algoritmos de Ordenação Eficientes - Quick Sort

```c
int partitionRSEDGEWICK(int *v, int l, int r)
{
    int i = l-1;
    int j = r;
    int pivot = v[r];

    while(i<j)
    {
        PROCURAR O MAIOR : ENQUANTO v[i] FOR ...


    }



}
```

Algoritmos de Ordenação Eficientes - Quick Sort

```
int partitionRSEDGEWICK(int *v, int l, int r)
{
    int i = l-1;
    int j = r;
    int pivot = v[r];

    while(i<j)
    {
        while(v[++i] < pivot);
        PROCURAR O MENOR : ENQUANTO v[j] FOR ...

    }



}
```

Algoritmos de Ordenação Eficientes - Quick Sort

```
int partitionRSEDGEWICK(int *v, int l, int r)
{
    int i = l-1;
    int j = r;
    int pivot = v[r];

    while(i<j)
    {
        while(v[++i] < pivot);
        while(v[--j] >= pivot && j>l); //pivot<v[--j]
        SE i < j ...
    }



}
```

Algoritmos de Ordenação Eficientes - Quick Sort

```
int partitionRSEDGEWICK(int *v, int l, int r)
{
    int i = l-1;
    int j = r;
    int pivot = v[r];

    while(i<j)
    {
        while(v[++i] < pivot);
        while(v[--j] >= pivot && j>l); //pivot<v[--j]
        if(i<j) exch(v[i], v[j]);
    }
    SE i>=j, TROCAR pivot COM O ÚLTIMO MAIOR


}
```

Algoritmos de Ordenação Eficientes - Quick Sort

```
int partitionRSEDGEWICK(int *v, int l, int r)
{
    int i = l-1;
    int j = r;
    int pivot = v[r];

    while(i<j)
    {
        while(v[++i] < pivot);
        while(v[--j] >= pivot && j>l); //pivot<v[--j]
        if(i<j) exch(v[i], v[j]);
    }
    exch(v[i], v[r]);

    RETORNAR A NOVA POSICAO DO PIVOT
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

```c
int partitionRSEDGEWICK(int *v, int l, int r)
{
    int i = l-1;
    int j = r;
    int pivot = v[r];

    while(i<j)
    {
        while(v[++i] < pivot);
        while(v[--j] >= pivot && j>l); //pivot<v[--j]
        if(i<j) exch(v[i], v[j]);
    }
    exch(v[i], v[r]);

    return i;
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

```c
int partitionLSEDGEWICK(int *v,  int l, int r) {
    int i = l;
    int j = r+1;
    int pivot = v[l];

    while (1) {
        while (v[++i] < pivot) if (i == r) break;
        while (pivot < v[--j]) if (j == l) break;
        if (i >= j) break;

        exch(v[i], v[j]);
    }
    exch(v[l], v[j]); //posicionando o pivot
                      //último menor

    return j; //nova posição do pivot

}
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- define-se o pivô – elemento mais a direita

[3 1 4 2 4]

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- v[i] < pivô?
  - swap v[i] ↔ v[j]: "puxando" o maior elemento para direita
  - j++
- i++

```
[3    1    4    2    4]
 i
 j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- v[i] < pivô?
  - swap v[i] $\leftrightarrow$ v[j]: "puxando" o maior elemento para direita
  - j++
- i++

```
[3    1    4    2    4]
      i
      j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- v[i] < pivô?
  - swap v[i] ↔ v[j]: "puxando" o maior elemento para direita
  - j++
- i++

```
[3    1    4    2    4]
               i
               j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- v[i] < pivô?
  - swap v[i] ↔ v[j]: "puxando" o maior elemento para direita
  - j++
- i++

```
[3    1    4    2    4]
                i
           j
```

- v[i] < pivô?
  - swap v[i] ↔ v[j]: “puxando” o maior elemento para direita
  - j++
- i++

```
[3    1    2    4    4]
                i
           j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- i == r
- swap v[j] $\leftrightarrow$ pivô

```
[3    1    2    4    4]
                     i
                4
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort (Cormem)

- j = pivô na sua posição final

```
[3    1    2    4    4]
                     i
                j
```

Algoritmos de Ordenação Eficientes - Quick Sort

```c
int partitionCORMEM(int *v, int l, int r)
{
    int pivot = v[r];
    int j = l;
    int i = l;
    while(i<r)
    {
        if(less(v[i], pivot)){
            exch(v[i], v[j]);
            j++;
        }
        i++;
    }

    exch(v[r], v[j]);

    return j;
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

Observações sobre as implementações (Sedegewick - pivô direita)

- Cuidado com os limites
  - Os apontadores i e j não podem ultrapassar os limites pois serão usados nos swaps
  - Opções:
    - "anda" depois verifica
      ```
      while(v[++i] < pivot); //&& i<r?
      while(pivot < v[--j] && j>l);
      ```
      Se começa "andando", tem que inicializar antes da posição ser verificada
      ```
      i = l-1;
      j = r;
      ```
    - verifica depois "anda"
      ```
      while(v[i] < pivot) i++;
      while(pivot < v[j] && j>l) j--;
      ```
      Se "anda" depois, tem que inicializar na posição ser verificada
      ```
      i = l;
      j = r-1;
      ```
  - Nunca ultrapassa "r", pois sempre chega no pivot

Algoritmos de Ordenação Eficientes - Quick Sort

Observações sobre as implementações (Sedegewick - pivô direita)

- Manipular itens iguais ao pivô
  - Interrompa a varredura à direita para itens iguais ao pivô
  - Estudos mostraram que esta estratégia tende a balancear as partições quando há várias chaves duplicadas
  - Reorganiza as iguais ao pivot
- Se considerar o maior ou igual e o menor ou igual
  - Muitos elementos seriam ultrapassados
  - Partes muito desbalanceadas: sub-vetores com tamanhos muito diferentes e chaves iguais ao pivô ficariam espalhadas

Algoritmos de Ordenação Eficientes - Quick Sort

```
[4   5   4   4   6   5]
 i<=

[4   5   4   4   6   5]
                 i<=

[4   5   4   4   6   5]
                     j>=

[4   5   4   4   6   5]
             j>= i<=

[4   5   4   4   5   6]
             j>= i<=
```

Algoritmos de Ordenação Eficientes - Quick Sort

```
[4    5    4    4    6    5]
 i<                       j>=

[4    5    4    4    6    5]
      i<        j>=

[4    4    4    5    6    5]
      i<        j>=

[4    4    4    5    6    5]
           j>=  i<

[4    4    4    5    6    5]
           j>=  i<
```

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$
- Estabilidade?

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$
- Estabilidade?
  - Mantém a ordem relativa?

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$
- Estabilidade?
  - Mantém a ordem relativa?
  - Tem trocas com saltos? Sim

Algoritmos de Ordenação Eficientes - Quick Sort

- In-place?
  - Somente recursão: proporcional a $\log n$
- Estabilidade?
  - Mantém a ordem relativa?
  - Tem trocas com saltos? Sim
  - Não estável

Algoritmos de Ordenação Eficientes - Quick Sort

- Adaptatividade?
  - Ordenação ajuda a melhorar o desempenho?

Algoritmos de Ordenação Eficientes - Quick Sort

- Adaptatividade?
  - Ordenação ajuda a melhorar o desempenho?
  - Não. Pode cair nos piores casos.

```
[2    1    3    4    5]
 i<

[2    1    3    4    5]
                     i<

[2    1    3    4    5]
                   i<,j>=

[2    1    3    4    5]
                j>=  i<

[2    1    3    4    5]
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Adaptatividade?
  - Ordenação ajuda a melhorar o desempenho?
  - Não. Pode cair nos piores casos.

```
[2    1    3    4    5]
 i<

[2    1    3    4    5]
                     i<

[2    1    3    4    5]
                   i<,j>=

[2    1    3    4    5]
                j>=  i<

[2    1    3    4    5]
```

```
[2    1    3    4    5]
 i<

[2    1    3    4    5]
                i<

[2    1    3    4    5]
                 i<,j>=

[2    1    3    4    5]
           j>=  i<

[2    1    3    4    5]
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Adaptatividade?
  - Ordenação ajuda a melhorar o desempenho?
  - Não. Pode cair nos piores casos.
  - A cada particiona, o pivô, já na sua posição correta, divide o sub-vetor em apenas "menos 1 elemento"

```
[2   1   3   4   5]
 i<

[2   1   3   4   5]
             i<

[2   1   3   4   5]
           i<,j>=

[2   1   3   4   5]
         j>= i<

[2   1   3   4   5]
```

```
[2   1   3   4   5]
 i<

[2   1   3   4   5]
         i<

[2   1   3   4   5]
          i<,j>=

[2   1   3   4   5]
     j>= i<

[2   1   3   4   5]
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Complexidade assintótica
  - Funciona bem com entradas aleatórias
  - Melhor e médio: $O(n \log n)$

```
int partition(int *v, int l, int r) {
  int i=l-1, j=r, pivot = v[r];

  while(i<j) {
    while(v[++i] < pivot);          //l até pivot
    while(pivot < v[--j] && j>l); //r até pivot
    if(i<j) exch(v[i], v[j]);     //até r-l trocas
  }

  exch(v[i], v[r]); //1

  //f(n) ~ 2n + 1
  return i;
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

- Complexidade assintótica
  - Funciona bem com entradas aleatórias
  - Melhor e médio: $O(n \log n)$

```
void quick_sort(int *v,  int l, int r) {
    if(r<=l) return;

    int p = partition(v, l, r); //O(n)

    quick_sort(v, l, p-1); //~f(n/2): melhor caso
    quick_sort(v, p+1, r); //~f(n/2): melhor caso
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

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

Algoritmos de Ordenação Eficientes - Quick Sort

- Complexidade assintótica
  - Funciona bem com entradas aleatórias
  - Melhor e médio: $O(n \log n)$
  - Pior caso: $n^2/2$ comparações
    - Muito itens repetidos, (quase) ordenados, reverso caem nos piores casos

```
void quick_sort(int *v,  int l, int r)
{
    if(r<=l) return;

    int p = partitionRSEDGEWICK(v, l, r); //n

    quick_sort(v, l, p-1); //~f(n-1)
    quick_sort(v, p+1, r); //~f(0)
}
```

Algoritmos de Ordenação Eficientes - Quick Sort

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

Algoritmos de Ordenação Eficientes - Quick Sort

Melhorias

- Mediana de três
  - Pivô: usar a mediana de uma pequena amostra de itens
  - Melhora o partiocionamento
  - Pivô mais à direita
    - Menor para left
    - Mediana para right

```
int meio = (l+r)/2;
if(v[meio] < v[l]) swap(v[meio], v[l]);
if(v[r] < v[l]) swap(v[l], v[r]);
if(v[meio] < v[r]) swap(v[r], v[meio]);
```

[5 x x 4 x x x 6]

[4 x x 5 x x x 6]

[4 x x 6 x x x 5]

Algoritmos de Ordenação Eficientes - Quick Sort

Mediana de três (pivô v[r])

```c
int meio = (l+r)/2;
if(v[meio] < v[l]) swap(v[meio], v[l]);
if(v[r] < v[l]) swap(v[l], v[r]);
if(v[meio] < v[r]) swap(v[r], v[meio]);
```

Utilizando as macros (Sedegewick)

```c
#define key(A) A
#define less(A, B) (key(A) < key(B))
#define exch(A, B) { Item t=A; A=B; B=t; }
#define compexch(A, B) if(less(B, A)) exch(A, B)

compexch(v[l], v[(l+r)/2]);
compexch(v[l], v[r]);
compexch(v[r], v[(l+r)/2]);
```

Implementem a mediana de três com pivô v[l]!

Algoritmos de Ordenação Eficientes - Quick Sort

Otimizando a mediana de três (pivô v[r])

- Menor item já está à esquerda
- Objetivo: colocar o maior item em r
  - Garantir um item maior que o pivô mais à direita
- Fazer o particionamento de (l+1, r-1)

```
exch(v[(l+r)/2], v[r-1]);
compexch(v[l], v[r-1]);
compexch(v[l], v[r]);
compexch(v[r-1], v[r]);

int p = partitionRSEDGEWICK(v, l+1, r-1);
```

[6	x	x	5	x	x	x	4]
[6	x	x	x	x	x	5	4]
[5	x	x	x	x	x	6	4]
[4	x	x	x	x	x	6	5]
[4	x	x	x	x	x	5	6]

Algoritmos de Ordenação Eficientes - Quick Sort

Melhorias

- Utilizar o Insertion Sort
  - Insertion Sort é mais rápido para pequenos vetores
  - Alternar para o Insertion para pequenos vetores
  - Algo entre 5 e 15 chaves

```
if(r-l<=15){
    insertion_sort(v, l, r);
}
...
```

Algoritmos de Ordenação Eficientes - Quick Sort

Melhorias

- Particionar o vetor em três partes (three-way)
  - v[l..i]: elementos menores que o pivô
  - v[i+1..j-1]: elementos iguais ao pivô
  - v[j..r]: elementos maiores que o pivô
- `quick_sort(v, l, i);`
- `quick_sort(v, j, r);`