Universidade Federal de Lavras
GCC224 - Introdução aos Algoritmos

Ordenação - Métodos simples

Insertion Sort, Selection Sort e Shell Sort

Prof. Joaquim Quinteiro Uchôa
Profa. Juliana Galvani Greghi
Profa. Marluce Rodrigues Pereira
Profa. Paula Christina Cardoso
Prof. Renato Ramos da Silva

- Contextualização
- Insertion Sort
- Selection Sort
- Shell Sort
- Análise dos métodos simples

Contextualização

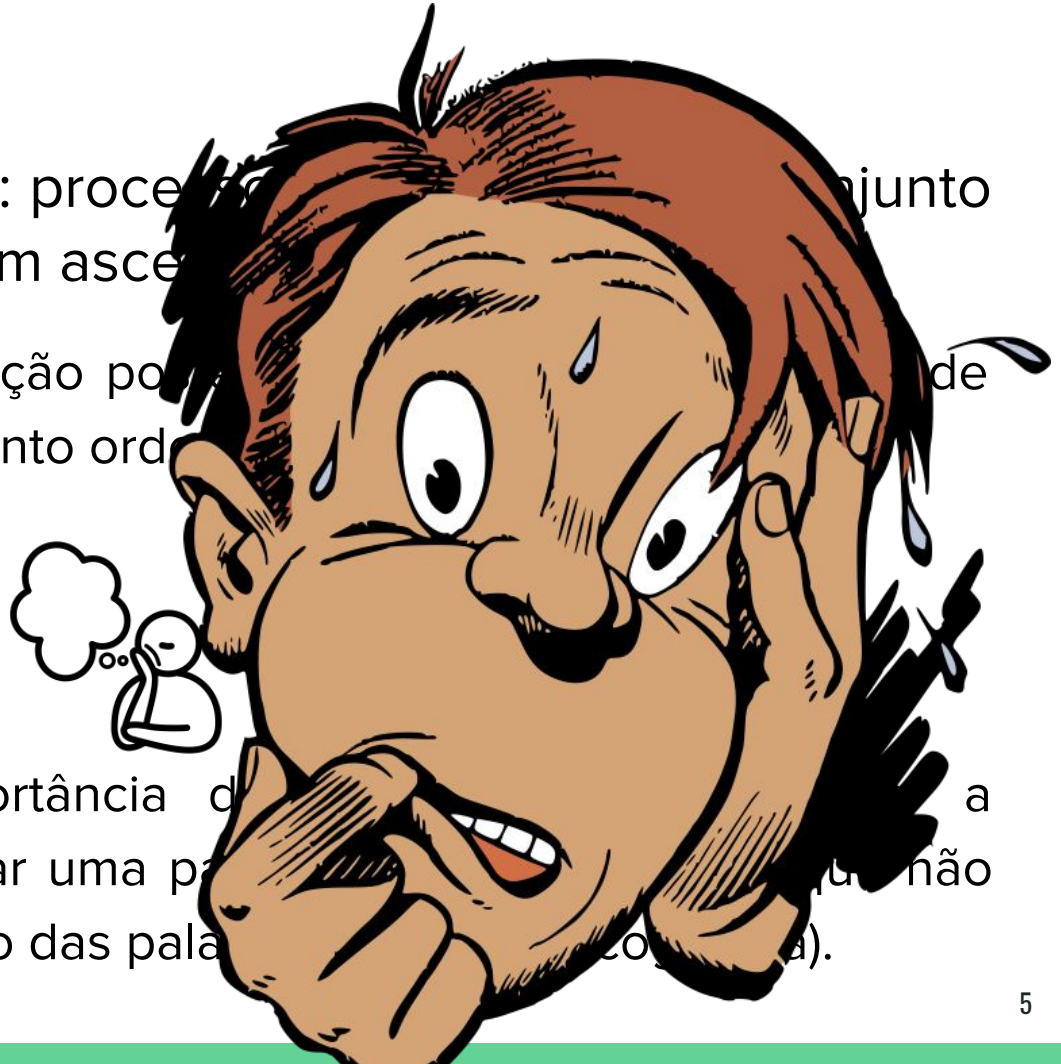
Contextualização

- **Definição de ordenação**: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
  - Visa facilitar a recuperação posterior, por meio de algoritmos de busca, de itens do conjunto ordenado.

  - Para entender a importância desta tarefa, tente imaginar a dificuldade em encontrar uma palavra em um dicionário que não possua um ordenamento das palavras (ordem lexicográfica).

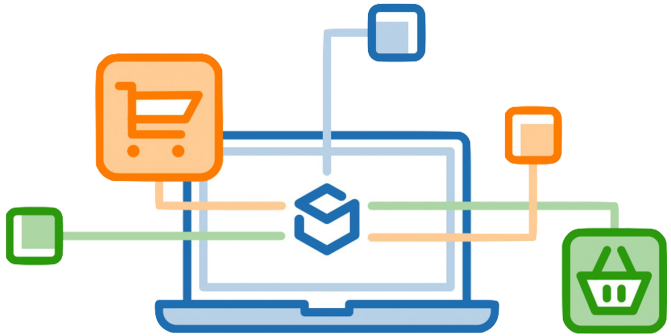
Contextualização

- **Definição de ordenação**: proce[illegible]njunto de objetos em uma ordem asce[illegible]
  - Visa facilitar a recuperação po[illegible]de busca, de itens do conjunto ord[illegible]
  - Para entender a importância d[illegible]a dificuldade em encontrar uma pa[illegible]não possua um ordenamento das pala[illegible]a).



Contextualização

- Outro exemplo que pode ilustrar a importância da ordenação:
  - Imagine que você foi em uma loja e perguntou o preço de um determinado produto. Você espera que o atendente digite o nome do produto em um programa e então apareçam as informações dele.



Contextualização

- Outro exemplo que pode ilustrar a importância da ordenação:

  - Para o programa encontrar as informações do produto, ele precisa percorrer um conjunto de dados (como um vetor, por exemplo) comparando o nome digitado com os dados armazenados no vetor.

Contextualização

- Outro exemplo que pode ilustrar a importância da ordenação:
  - Se houverem muitos produtos cadastrados e eles não estiverem ordenados, a busca irá demorar muito tempo. Se estiverem ordenados, um algoritmo de busca mais eficiente (como a busca binária, por exemplo) poderá encontrar o produto rapidamente.

Contextualização

- **Definição formal de ordenação:**

  - Sejam os itens $\mathbf{a_1}$, $\mathbf{a_2}$, ..., $\mathbf{a_n}$. Ordenar consiste em permutar tais itens em uma ordem $\mathbf{a_{k1}}$, $\mathbf{a_{k2}}$, ..., $\mathbf{a_{kn}}$ tal que dada uma função de ordenação **f** tem-se sempre que $\mathbf{f(a_{k1}) \leq f(a_{k2}) \leq ... \leq f(a_{kn})}$.

Contextualização

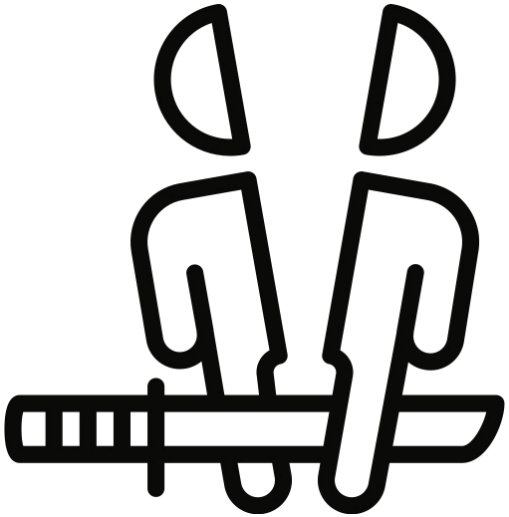
- **Ordenação: valores para comparação**
  - Qualquer tipo de variável para o qual exista uma relação de ordenação bem definida pode ser utilizada para se realizar a comparação de elementos durante o processo de ordenação.
    - O mais comum são: inteiros e sequências de caracteres.
  - Os valores utilizados na comparação do processo de ordenação são chamados de chaves.

Contextualização

- Na prática, existem dezenas de algoritmos diferentes para resolver o problema de ordenação. Os algoritmos existentes podem ser divididos em diversas categorias.
    - De acordo com o uso da memória pelos algoritmos de ordenação, tem-se:
        - **Ordenação interna:** quando todos os elementos a serem ordenados cabem na memória principal do computador e qualquer chave pode ser acessada imediatamente.
        - **Ordenação externa:** quando todos os elementos a serem ordenados não cabem na memória principal do computador e as chaves são acessadas sequencialmente ou em grandes blocos.

Contextualização

- Dentre os algoritmos de ordenação interna, uma segunda categorização, baseada na complexidade das soluções, ainda pode ser definida, dividindo-os em:
  - Algoritmos simples.
  - Algoritmos eficientes.



Contextualização

- Algoritmos simples: métodos com complexidade quadrática, ou seja, que requerem $O(n^2)$ comparações.
  - Fáceis de implementar e de entender.
  - Adequados para conjunto pequenos de elementos.
- Algoritmos eficientes: requerem uma quantidade menor de comparações, usualmente na ordem de $O(n*logn)$.
  - Mais complexos para se implementar e entender.
  - Adequados para conjuntos grandes de elementos.

Contextualização

Neste curso de Introdução aos Algoritmos serão estudados os seguintes algoritmos:

- Algoritmos simples: ***Selection Sort*** (ordenação por seleção), ***Insertion Sort*** (ordenação por inserção) e ***Shell Sort*** (melhoria da ordenação por inserção).
- Algoritmos eficientes: ***Quick Sort*** (ordenação por particionamento) ***e Merge Sort*** (ordenação por intercalação).

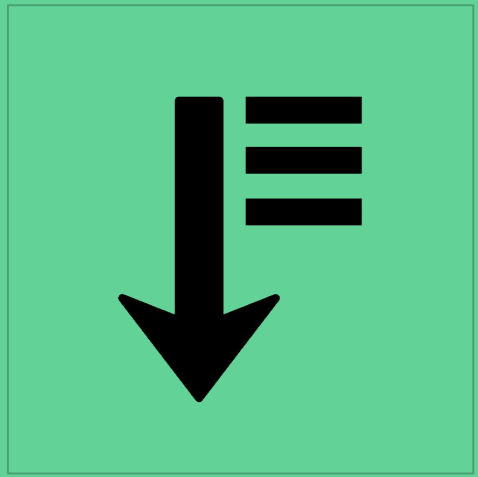
Não abordaremos aqui um método relativamente popular, mas bastante ineficiente, e não necessariamente mais simples que Selection ou Insertion, o Bubble Sort.

Contextualização

Não abordaremos aqui um método relativamente popular, mas bastante ineficiente, e não necessariamente mais simples que Selection Sort ou Insertion Sort, o **Bubble Sort** (método das bolhas).

Esse método, inclusive, deve ser evitado nas atividades da disciplina.

Selection Sort



Selection Sort

- ***Selection Sort***, ou método da seleção, consiste na ideia de selecionar o menor elemento do arranjo e trazê-lo para a primeira posição do vetor (assumindo uma ordenação ascendente), depois o segundo menor valor para a segunda posição, e assim é feito sucessivamente com os elementos restantes.
    - Do lado esquerdo do vetor, a partir de uma posição de referência, vai-se montando um conjunto ordenado e, a cada passo, o menor elemento do lado direito é trazido para o final do conjunto ordenado do lado esquerdo.

Selection Sort - Algoritmo

- O ***Selection Sort*** pode ser resumido por meio das seguintes ações, assumindo que **V** seja um arranjo não ordenado com **N** elementos:
  - Selecionar o elemento em **V** com o menor valor de chave;
  - Trocar de posição o menor elemento encontrado com o elemento que ocupa **V[0]**;
  - Repetir as duas operações acima com os demais **N-1** elementos restantes em **V**, fazendo a troca para a posição **1** do vetor; depois com os demais **N-2** elementos restantes em **V**, fazendo a troca para a posição **2** do vetor; e assim sucessivamente.

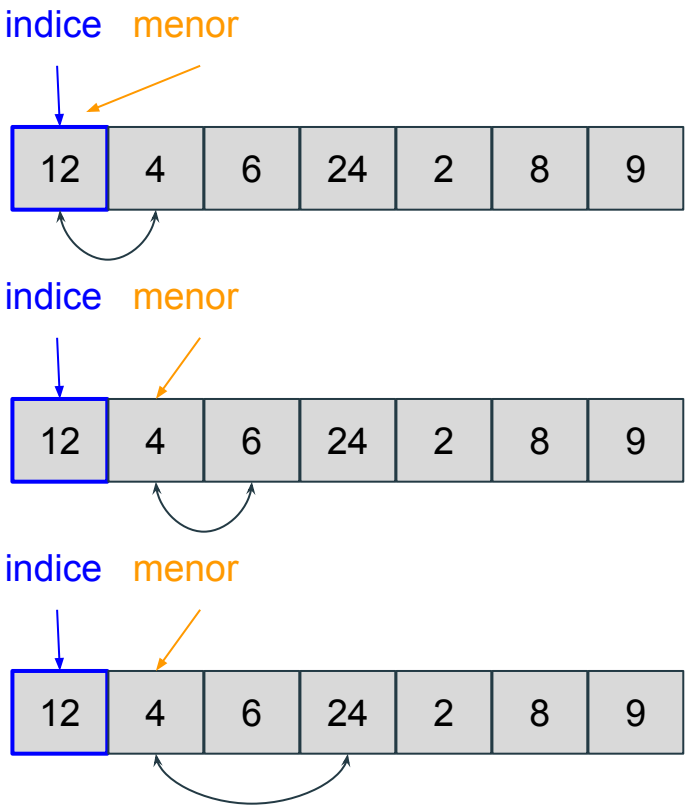
Selection Sort - Exemplo

- Vamos ver um exemplo prático de como o ***Selection Sort*** funciona. Para isso, considere que estamos interessados em ordenar o seguinte arranjo em ordem ascendente:

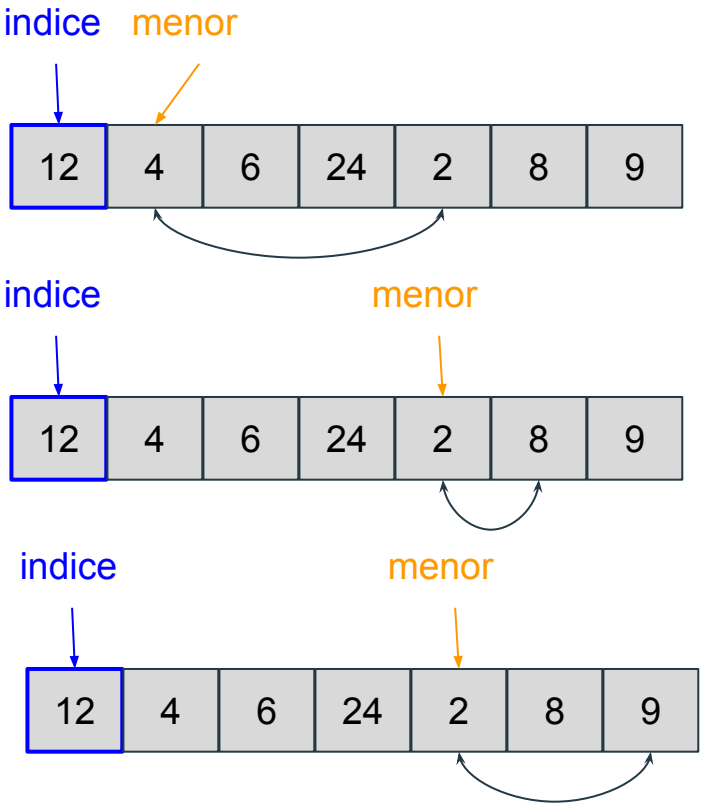
12	4	6	24	2	8	9

  - Nos próximos slides, será utilizada a variável **índice** para demarcar a posição de referência do vetor, a partir da qual os elementos do arranjo **não** estarão ordenados. A variável **menor** será utilizada para demarcar a posição do menor elemento a direita do **índice** (lembre-se que este é um processo iterativo e a posição do menor elemento muda enquanto investigamos o arranjo).

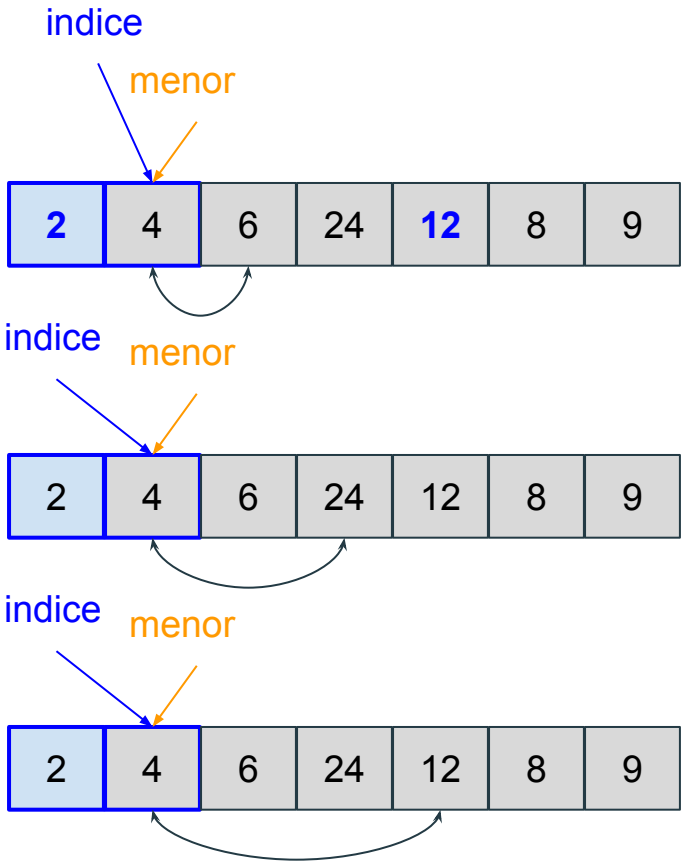
Exemplo Selection Sort



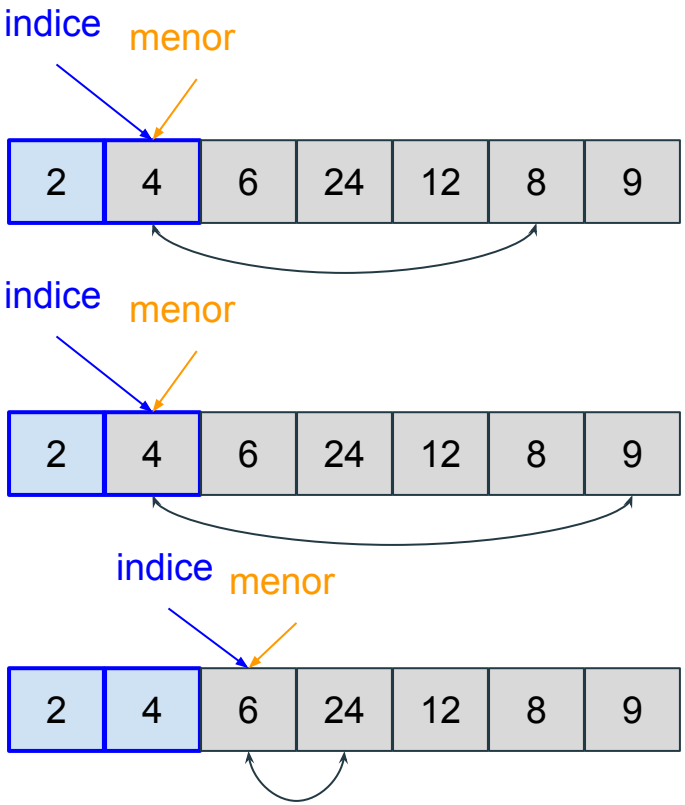
Exemplo Selection Sort



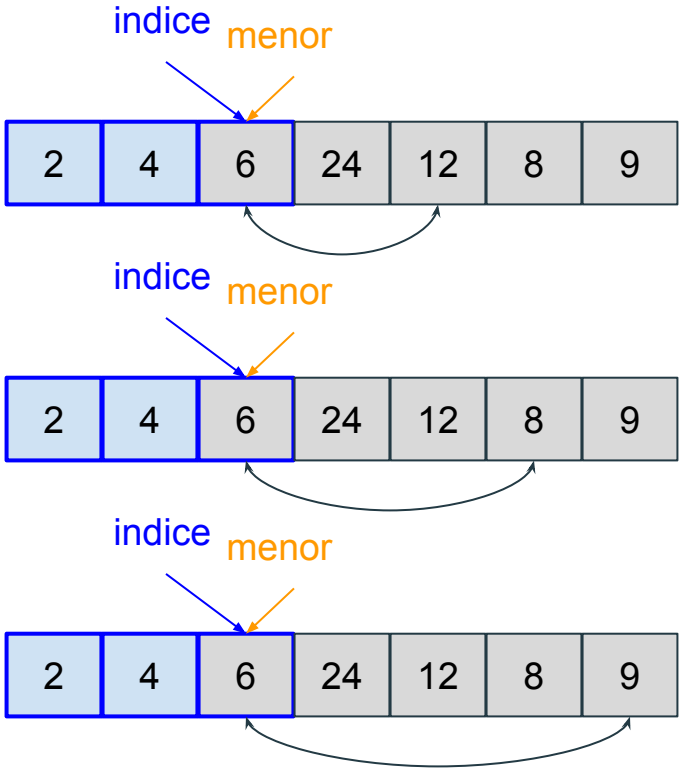
Exemplo Selection Sort



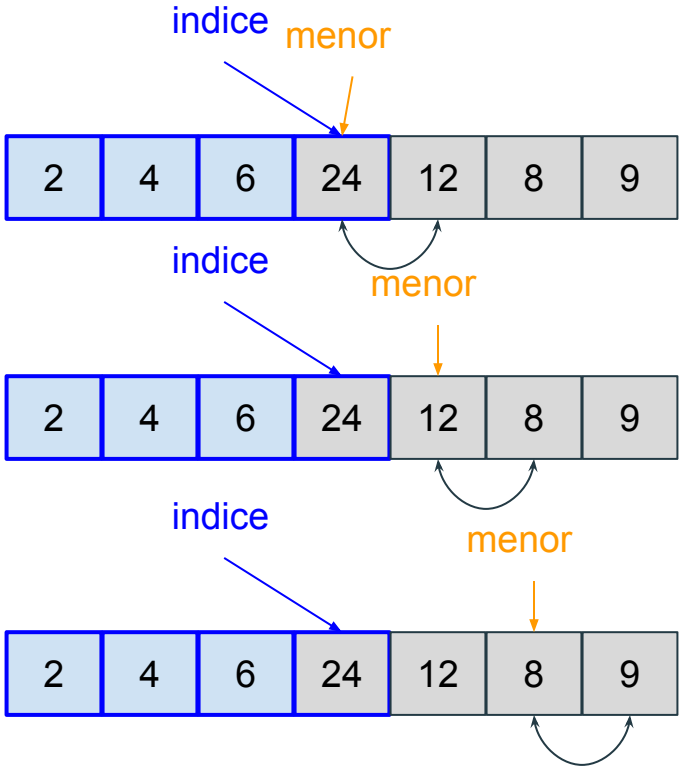
Exemplo Selection Sort



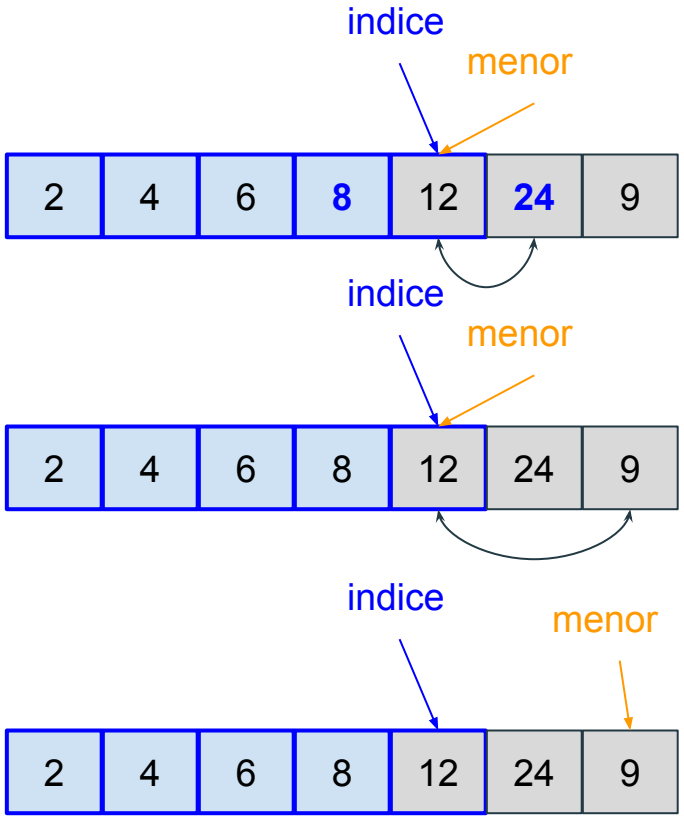
Exemplo Selection Sort



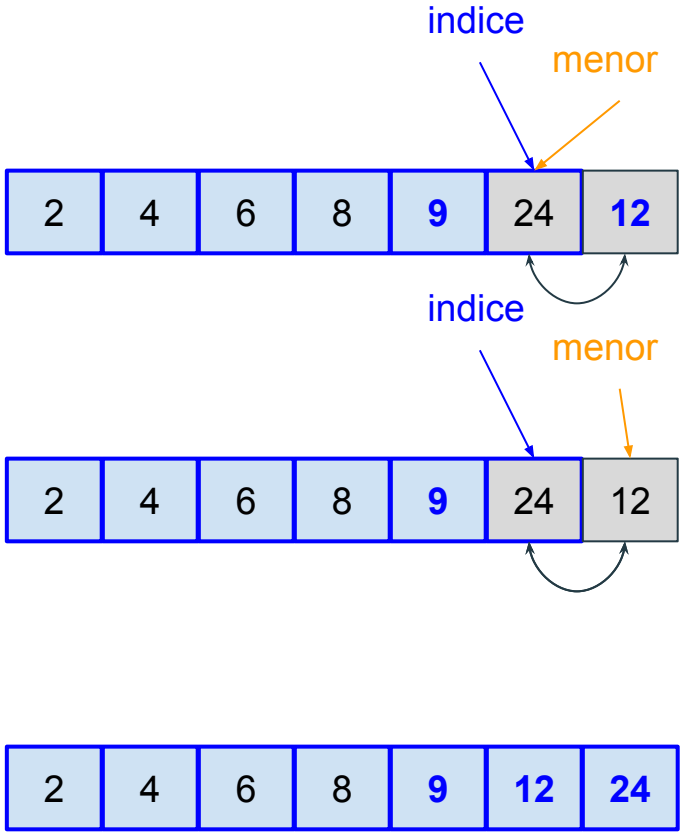
Exemplo Selection Sort



Exemplo Selection Sort



Exemplo Selection Sort



Selection Sort

- **Subprograma do Selection Sort**
  - A seguir, no próximo slide, será apresentado um exemplo de procedimento que realiza o algoritmo ***Selection Sort*** em um vetor de números inteiros.
  - **Atenção**: o exemplo de implementação do próximo slide é apenas isso, um exemplo. É possível implementar o conceito do método da seleção de diferentes formas e utilizando vetores de diferentes tipos de dados.
    - **Recomendação**: não tente decorar a implementação do exemplo. Tente entender como o método da seleção funciona!

Selection Sort - Algoritmo (Relembrando)

- O ***Selection Sort*** pode ser resumido por meio das seguintes ações, assumindo que **V** seja um arranjo não ordenado com **N** elementos:
  - Selecionar o elemento em **V** com o menor valor de chave;
  - Trocar de posição o menor elemento encontrado com o elemento que ocupa **V[0]**;
  - Repetir as duas operações acima com os demais **N-1** elementos restantes em **V**, fazendo a troca para a posição **1** do vetor; depois com os demais **N-2** elementos restantes em **V**, fazendo a troca para a posição **2** do vetor; e assim sucessivamente.

Selection Sort

```
void selection_sort(int vetor[], int tam){
    int menor, aux_troca;
    for (int indice = 0; indice < tam-1; indice++) {
        menor = indice;
        for (int j = indice + 1; j < tam; j++) {
            if (vetor[j] < vetor[menor]){
                menor = j;
            }
        }
        aux_troca = vetor[indice];
        vetor[indice] = vetor[menor];
        vetor[menor] = aux_troca;
    }
}
```

Variáveis:

indice: demarca a posição do arranjo a partir da qual todos os elementos a sua direita ainda não foram devidamente ordenados (loop externo).

menor: demarca a posição do menor elemento do arranjo que está à direita da posição de referência **índice** (loop interno).

aux_troca: variável auxiliar para trocar de posição os elementos do vetor.

j: variável auxiliar que demarca as posições à direita da posição de referência **índice** (loop interno).

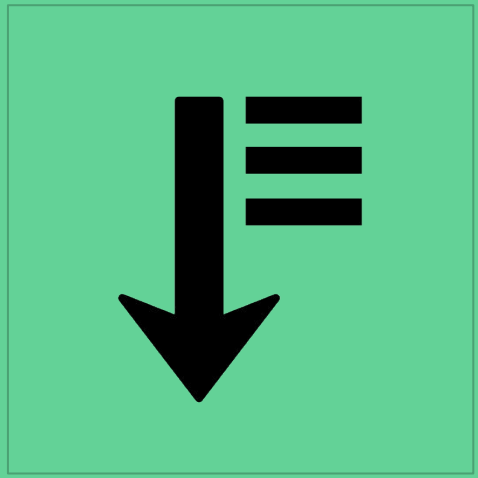
Selection Sort

```
void selection_sort(int vetor[], int tam){
    int menor, aux_troca;
    for (int indice = 0; indice < tam-1; indice++) {
        menor = indice;
        for (int j = indice + 1; j < tam; j++) {
            if (vetor[j] < vetor[menor]){
                menor = j;
            }
        }
        aux_troca = vetor[indice];
        vetor[indice] = vetor[menor];
        vetor[menor] = aux_troca;
    }
}
```

Teste este subprograma. Para isso, não se esqueça que você deve construir a função principal **main()**, realizar a leitura de um vetor de números inteiros, chamar o procedimento **selection_sort()** e exibir o vetor após a chamada do subprograma.

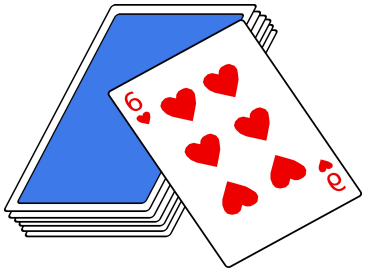
Após este primeiro teste, modifique o subprograma de modo que ele possa ordenar o vetor em ordem decrescente.

Insertion Sort



Insertion Sort

- ***Insertion Sort***, ou Ordenação por Inserção, ordena um vetor (arranjo), inserindo um elemento por vez em sua posição correta.



  - Assemelha-se à forma como algumas pessoas organizam um baralho num jogo de cartas, ao receber uma nova carta. Você recebe uma nova carta e deve colocá-la na posição correta da sua mão de cartas, de forma que as cartas sigam uma determinada ordenação.

Insertion Sort

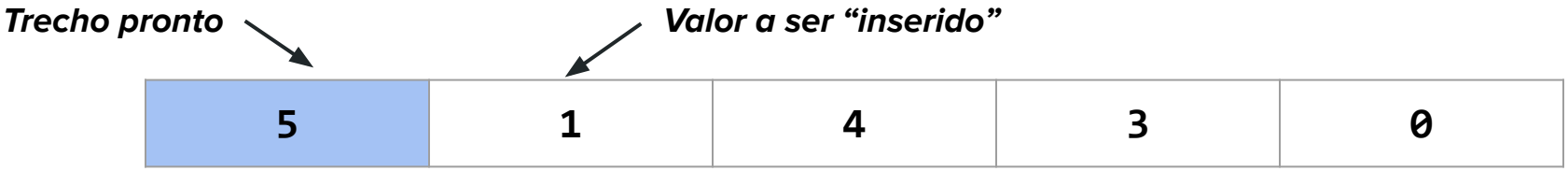
- O ***Insertion Sort*** é baseado em três ações principais, a saber:
  - Definir um elemento do conjunto de dados como pivô;
  - Comparar o elemento pivô com outro elemento do conjunto de dados;
  - Mover o maior dos dois elementos para a direita no arranjo em que se encontram (ordenação em ordem ascendente).

Insertion Sort - Algoritmo

- Dadas estas três ações principais, o algoritmo ***Insertion Sort*** pode ser expresso por meio do aninhamento de duas estruturas de repetição:
  - Estrutura de repetição externa: pegar um dos valores do conjunto não ordenado como sendo o pivô para achar sua posição correta, fazendo isso da segunda posição do arranjo até a última.
  - Estrutura de repetição interna: comparar o pivô com os elementos localizados à sua esquerda no arranjo. Enquanto o subconjunto dos elementos a sua esquerda não acabar e eles forem maiores do que o pivô, move-se os maiores valores para a direita.

Enxergando de outra forma...

Uma forma alternativa de entender o Insertion Sort é pensar que o vetor pronto está no trecho inicial do vetor inteiro.



Inicialmente apenas um elemento está ordenado (a primeira posição). Então pegamos a segunda posição e “inserimos” nesse vetor, já de forma ordenada.

Esse processo é, então, repetido para os demais elementos.

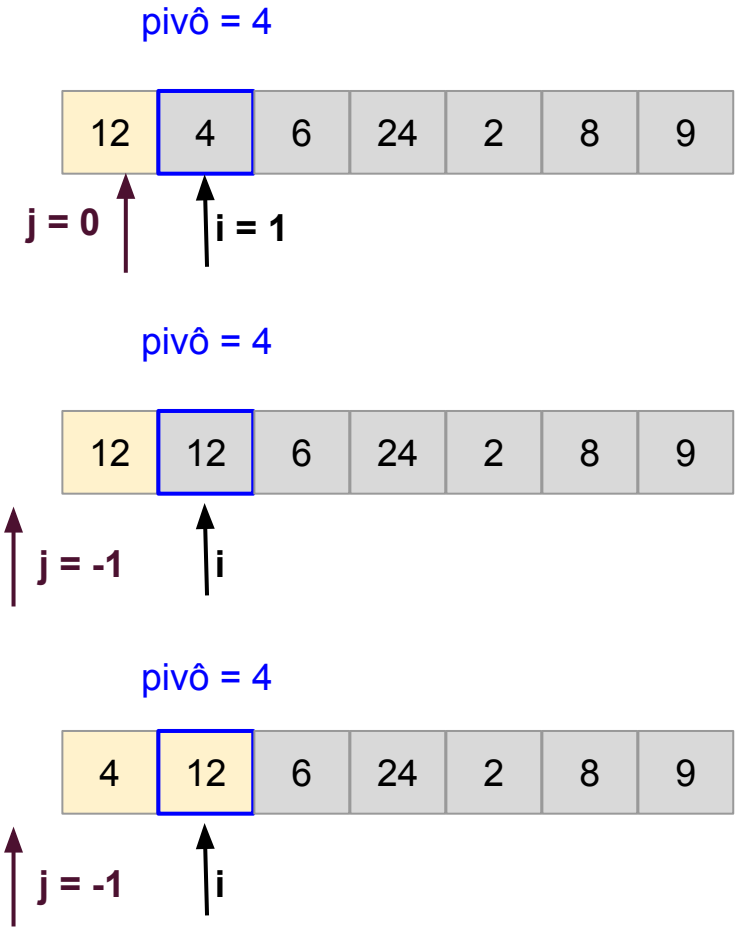
Insertion Sort - Exemplo

- Vamos ver um exemplo prático de como o ***Insertion Sort*** funciona. Para isso, considere que estamos interessados em ordenar o seguinte arranjo em ordem ascendente:

12	4	6	24	2	8	9

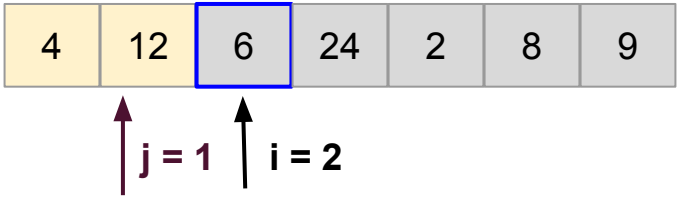
  - Nos próximos slides, será utilizada a variável **i** para demarcar a posição do pivô escolhido a cada etapa do processo de ordenação. A variável **j** será utilizada para demarcar a posição do elemento a esquerda do pivô que está sendo comparado com ele a cada iteração.

Exemplo Insertion Sort

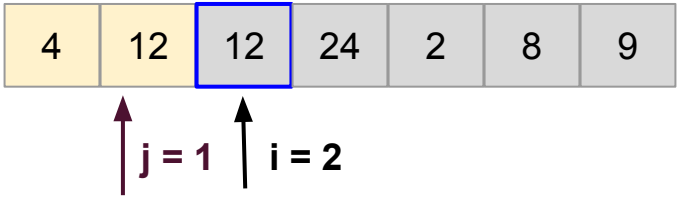


Exemplo Insertion Sort

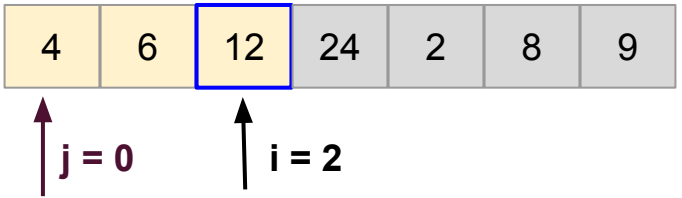
pivô = 6



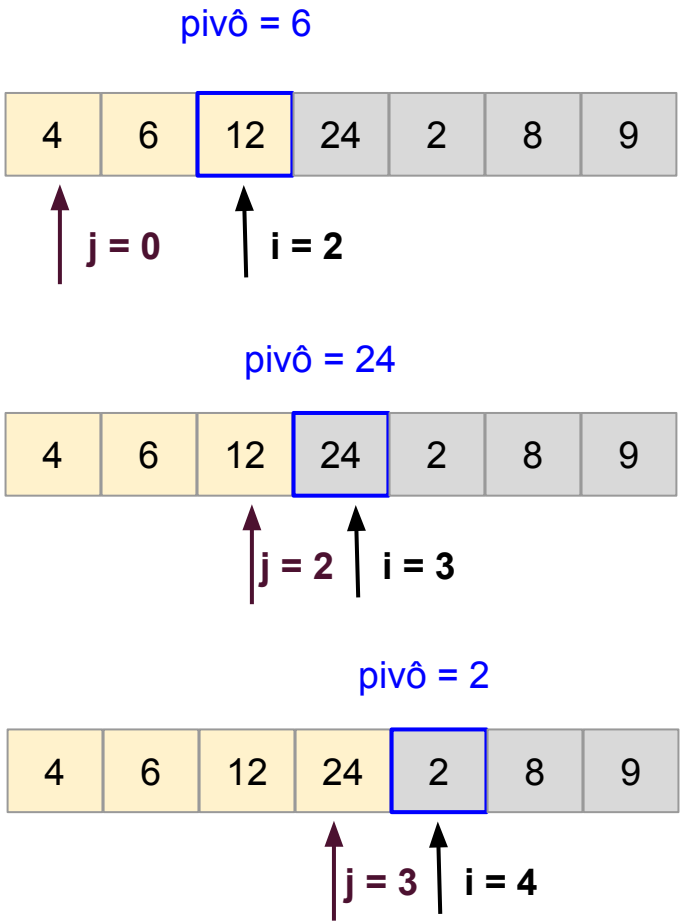
pivô = 6



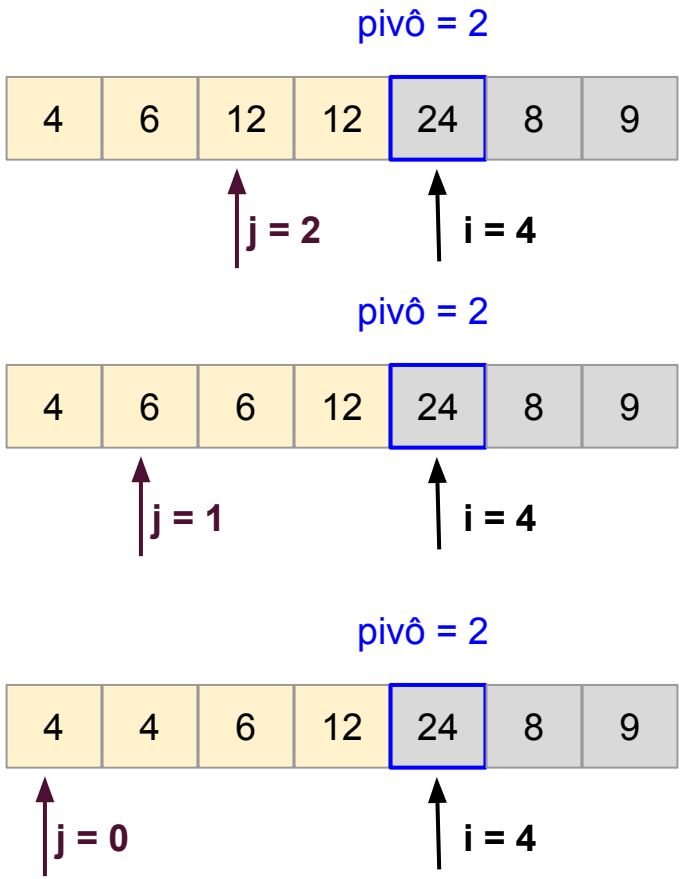
pivô = 6



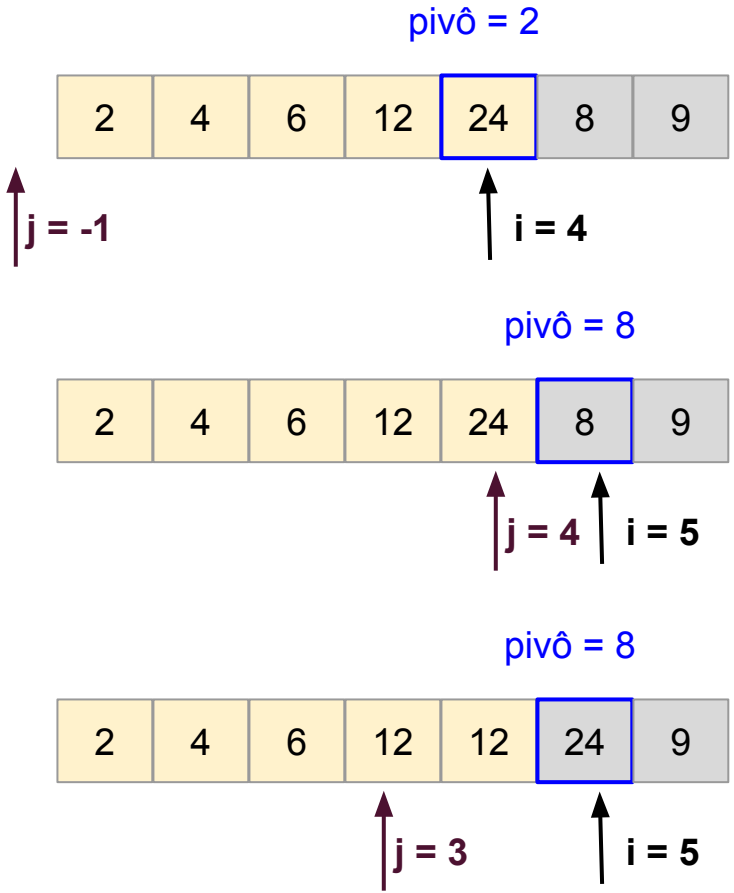
Exemplo Insertion Sort



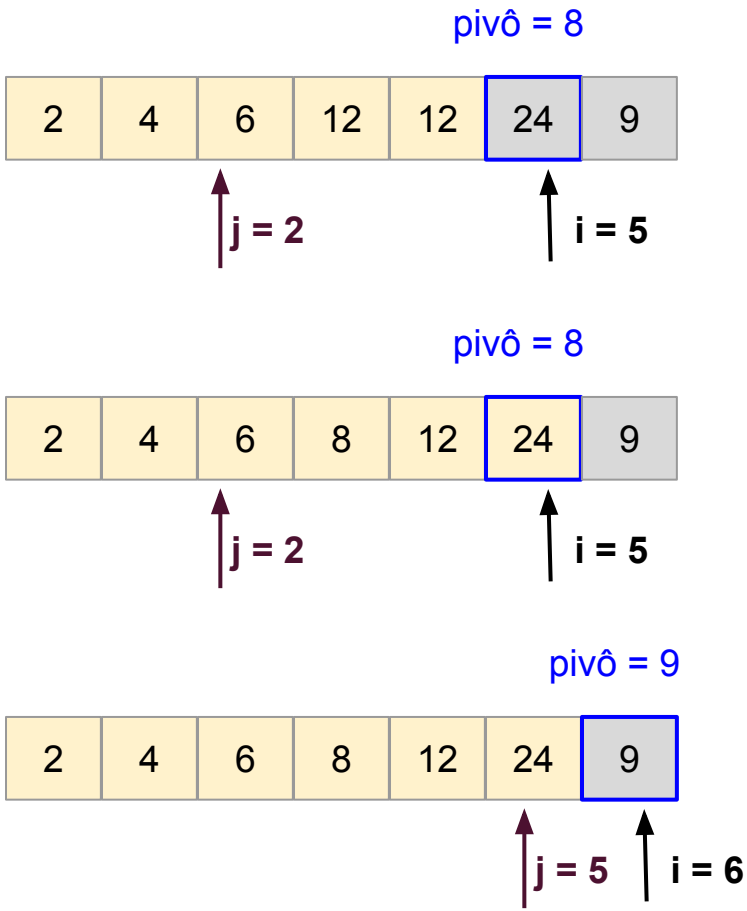
Exemplo Insertion Sort



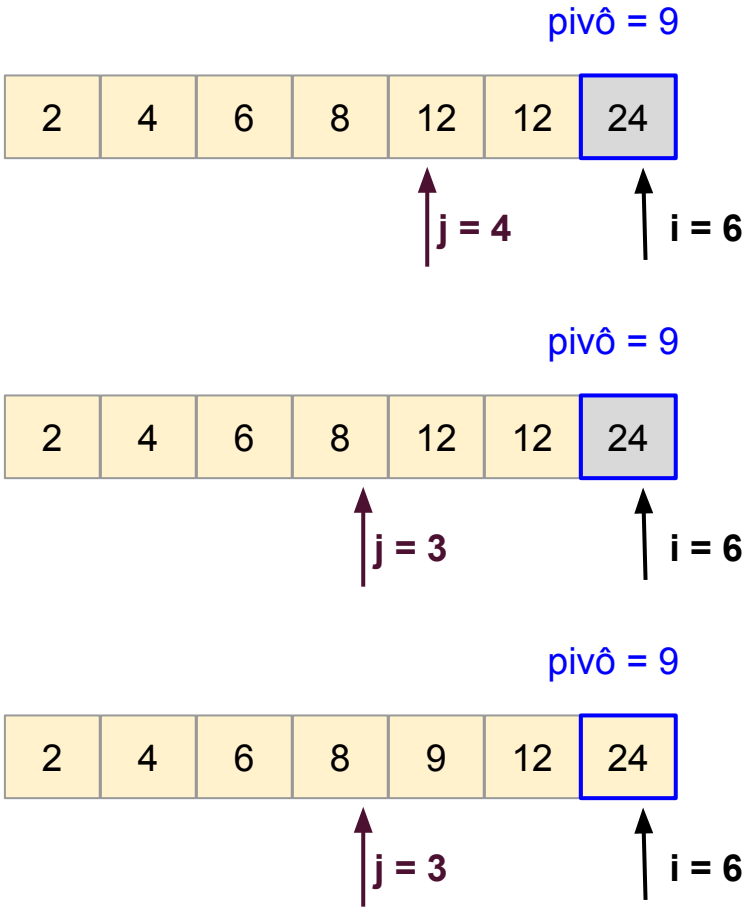
Exemplo Insertion Sort



Exemplo Insertion Sort



Exemplo Insertion Sort



Insertion Sort

- **Subprograma do Insertion Sort**
  - A seguir, no próximo slide, será apresentado um exemplo de procedimento que realiza o algoritmo ***Insertion Sort*** em um vetor de números inteiros.
  - **Atenção**: o exemplo de implementação do próximo slide é apenas isso, um exemplo. É possível implementar o conceito da ordenação por inserção de diferentes formas e utilizando vetores de diferentes tipos de dados.
    - **Recomendação**: não tente decorar a implementação do exemplo. Tente entender como a ordenação por inserção funciona!

Insertion Sort - Algoritmo (Relembrando)

O algoritmo ***Insertion Sort*** pode ser expresso por meio do aninhamento de duas estruturas de repetição:

- Estrutura de repetição externa: pegar um dos valores do conjunto não ordenado como sendo o pivô para achar sua posição correta, fazendo isso da segunda posição do arranjo até a última.
- Estrutura de repetição interna: comparar o pivô com os elementos localizados à sua esquerda no arranjo. Enquanto o subconjunto dos elementos a sua esquerda não acabar e eles forem maiores do que o pivô, move-se os maiores valores para a direita.

Insertion Sort

```
void insertion_sort(int vetor[], int tam){
    int valor_pivo, j;
    for (int i = 1; i < tam; i++) {
        valor_pivo = vetor[i];
        j = i - 1;
        while ((j >= 0) and (valor_pivo < vetor[j])){
            vetor[j+1] = vetor[j];
            j--;
        }
        vetor[j+1] = valor_pivo;
    }
}
```

Variáveis:

vetor: arranjo que armazena os elementos a serem ordenados.

tam: quantidade de elementos armazenados no vetor

i: posição do pivô. Note que a cada iteração do loop exterior o pivô ocupa uma posição diferente.

j: posição dos elementos que serão comparados com o pivô. Note que os elementos a serem comparados sempre estarão à esquerda do pivô.

Valor_pivo: valor do elemento armazenado na posição do pivô.

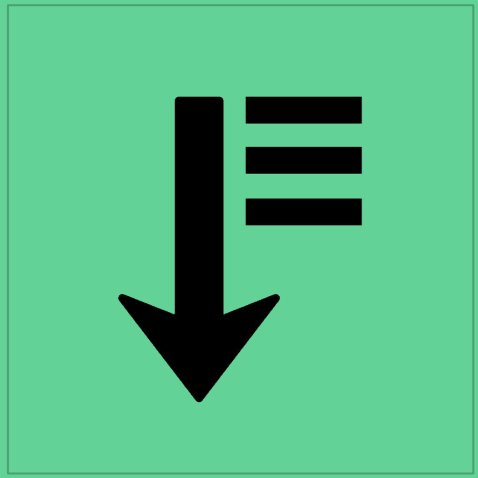
Insertion Sort

```
void insertion_sort(int vetor[], int tam){
    int valor_pivo, j;
    for (int i = 1; i < tam; i++) {
        valor_pivo = vetor[i];
        j = i - 1;
        while ((j >= 0) and (valor_pivo < vetor[j])){
            vetor[j+1] = vetor[j];
            j--;
        }
        vetor[j+1] = valor_pivo;
    }
}
```

Teste este subprograma. Para isso, não se esqueça que você deve construir a função principal **main()**, realizar a leitura de um vetor de números inteiros, chamar o procedimento **insertion_sort()** e exibir o vetor após a chamada do subprograma.

Após este primeiro teste, modifique o subprograma de modo que ele possa ordenar o vetor em ordem decrescente.

Shell Sort



Relembrando o Insertion Sort - i

O algoritmo ***Insertion Sort*** pode ser expresso por meio do aninhamento de duas estruturas de repetição:

- Estrutura de repetição externa: pegar um dos valores do conjunto não ordenado como sendo o pivô para achar sua posição correta, fazendo isso da segunda posição do arranjo até a última.
- Estrutura de repetição interna: comparar o pivô com os elementos localizados à sua esquerda no arranjo. Enquanto o subconjunto dos elementos a sua esquerda não acabar e eles forem maiores do que o pivô, move-se os maiores valores para a direita.

Relembrando o Insertion Sort - ii

```
void insertion_sort(int vetor[], int tam){
    int valor_pivo, j;
    for (int i = 1; i < tam; i++) {
        valor_pivo = vetor[i];
        j = i - 1;
        while ((j >= 0) and (valor_pivo < vetor[j])){
            vetor[j+1] = vetor[j];
            j--;
        }
        vetor[j+1] = valor_pivo;
    }
}
```

Relembrando o Insertion Sort - ii

O insertion sort possui a vantagem de possuir uma implementação muito simples e ser relativamente eficiente para volumes pequenos de dados, o que torna recomendado nesses casos. Em geral, na prática é mais eficiente que outros algoritmos simples, como bubble sort ou selection sort.

Além disso, insertion sort é **adaptativo**, ou seja, **aumenta a eficiência em dados que já estão parcialmente ordenados** e consegue ordenar uma lista à medida que a recebe (não precisa da lista inteira para começar a ordenação).

Relembrando o Insertion Sort - ii

O ins de p n
impl amente e ciente para
volu ecomend do nesses
casc ic que out os
algo rt ou selectio sort.

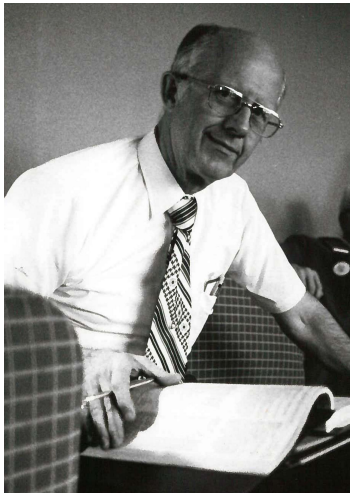
Além vo, ou se aumenta a
efici arcialm **ordenados** e
cons la que ebe (não precisa
da lis ação).

Será que dá para melhorar o Insertion Sort???

Shell Sort

O Shell sort, proposto por Donald Shell, é uma generalização, e melhoria, do insertion sort, utilizando a ideia de um salto (*gap*) entre os elementos. Assim, começando com lacunas maiores, vai-se diminuindo o tamanho do *gap*, até permitir a comparação um a um.

A eficiência do método é, obviamente, dependente da sequência de lacunas escolhidas.



Shell Sort - Exemplo - i

Suponha o vetor
`v = [59, 17, 30, 94, 51, 12, 28, 3, 21, 33, 16, 8]` e a sequência de lacunas dada por `S = [5, 3, 1]`.

59	17	30	94	51	12	28	3	21	33	16	8

Nesse caso, na primeira iteração, serão ordenados por inserção os subvetores [59,12,16], [17,28,8], [30,3], [94,21] e [51,33].

Shell Sort - Exemplo - ii

Assim, após a primeira passada, obtém-se

12	8	3	21	33	16	17	30	94	51	59	28

Na segunda passada os elementos serão ordenados utilizando uma lacuna de tamanho 3:

12	8	3	21	33	16	17	30	94	51	59	28

Nesse caso, serão ordenados por inserção os subvetores `[12,21,17,51]`, `[8,33,30,59]` e `[3,16,94,28]`. *É interessante observar que os subvetores estão semi-ordenados.*

Shell Sort - Exemplo - iii

Após a segunda passada, obtém-se

12	8	3	17	30	16	21	33	28	51	59	94

Por fim, os elementos serão agora ordenados com lacuna de tamanho um, ou seja, um insertion sort normal:

12	8	3	17	30	16	21	33	28	51	59	94

A maioria dos elementos já encontra-se em sua posição ou próxima a ela. Isso significa que haverá menos trocas e o algoritmo terminará mais cedo.

Sequências para o Shell Sort

A velocidade/eficiência do Shell sort é dependente da sequência utilizada para as lacunas. Segue-se algumas sequências utilizadas mais frequentemente:

- $2^k - 1$ => {1, 3, 7, 15, 31, 63, ... }
- $2^k + 1$, prefixado por 1 => {1, 3, 5, 9, 17, 33, 65, ... }
- $(3^k - 1)/2$ => {1, 4, 13, 40, 121, ... }
- a(n) = ceil ($(9 * (9/4)^n - 4) / 5)$) => {1, 4, 9, 20, 46, 103, ... }
- Ciura (experimental) => {1, 4, 10, 23, 57, 132, 301, 701, **1750**}

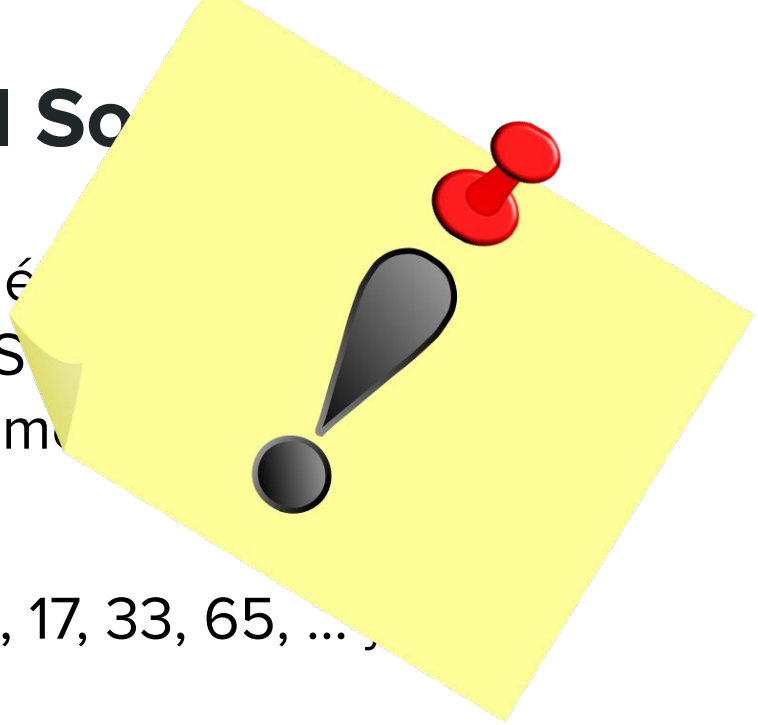
Sequências para o Shell Sort

A velocidade/eficiência do Shell sort é dependente da sequência utilizada para as lacunas. Segue-se algumas sequências utilizadas mais frequentemente:

- $2^k - 1$ => {1, 3, 7, 15, 31, 63, ... }
- $2^k + 1$, prefixado por 1 => {1, 3, 5, 9, 17, 33, 65, ... }
- $(3^k - 1)/2$ => {1, 4, 13, 40, 121, ... }
- a(n) = ceil ($(9 * (9/4)^n - 4) / 5)$) => {1, 4, 9, 20, 46, 103, ... }
- Ciura (experimental) => {1, 4, 10, 23, 57, 132, 301, 701, **1750**}

Sequências para o Shell So

A velocidade/eficiência do Shell sort é
sequência utilizada para as lacunas. S
sequências utilizadas mais frequentem



- A sequência de Ciura foi a que melhor produziu resultados em estudo experimental
- 5, 9, 17, 33, 65, ... }
- }
- => {1, 4, 9, 20, 46, 103, ... }
- Ciura (experimental) => {1, 4, 10, 23, 57, 132, 301, 701, **1750**}

Shell Sort - Algoritmo - i

```
void shell_sort(int vet[], int size) {
    // sequência de Ciura, a que tem os melhores resultados experimentais
    // para vetores maiores usar recursão -> h[k] =  floor( 2.25 * h[k-1] ).

    int gaps[9] = {1, 4, 10, 23, 57, 132, 301, 701, 1750};
    int pos_gap = 8;

    while (gaps[pos_gap] > size) {
        pos_gap--;
    }

    int value;
    int j;

    while ( pos_gap >= 0 ) {...
```

Shell Sort - Algoritmo - ii

```
    while ( pos_gap >= 0 ) {
        int gap = gaps[pos_gap];
        cout << "gap: " << gap << endl;

        for (int i = gap; i < size; i++) {
            value = vet[i];
            j = i;
            while ((j >= gap) and (value < vet[j - gap])) {
                vet[j] = vet[j - gap];
                j = j - gap;
            }
            vet[j] = value;
        }

        pos_gap--;
    }
}
```

Shell Sort - Algoritmo - ii

```
    while ( pos_gap >= 0 ) {
        int gap = gaps[pos_gap];
        cout << "gap: " << gap << endl;

        for (int i = gap; i < size; i++) {
            value = vet[i];
            j = i;
            while ((j >= gap) and (value < vet[j - gap])) {
                vet[j] = vet[j - gap];
                j = j - gap;
            }
            vet[j] = value;
        }

        pos_gap--;
    }
}
```

Insertion sort tradicional, mas usando comparações pulando uma lacuna (gap).

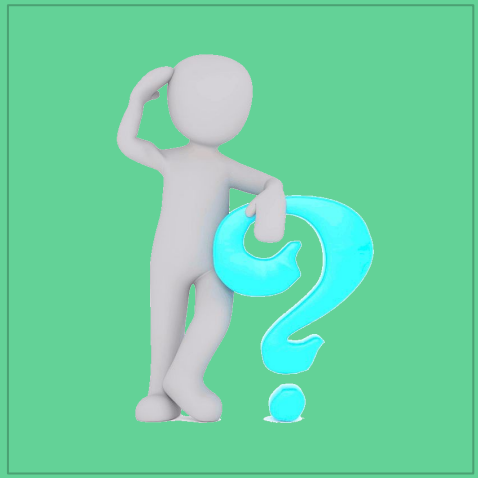
Shell Sort - Algoritmo - ii

```
    while ( pos_gap >= 0 ) {
        int gap = gaps[pos_gap];
        cout << "gap: " << gap << endl;

        for (
        }
        pos_gap--;
    }
}
```

```
[joukim@harpia tmp]$ ./shellsort
gap: 10
gap: 4
gap: 1
-4 -2 0 0 1 1 2 4 8 12 23 55 199
[joukim@harpia tmp]$
```

Análise dos Métodos Simples



Breve Análise dos Métodos Simples - i

Os métodos simples possuem problemas de eficiência em grandes volumes de dados, sendo indicado nesses casos a utilização de algoritmos mais avançados, como quick sort, merge sort ou heap sort, dependendo o caso.

Entretanto, apesar do problema de eficiência, alguns desses métodos possuem espaço em diversas aplicações.

Breve Análise dos Métodos Simples - ii

O insertion sort possui a vantagem de possuir uma implementação muito simples e ser relativamente eficiente para volumes pequenos de dados, o que torna recomendado nesses casos. Em geral, na prática é mais eficiente que outros algoritmos simples, como bubble sort ou selection sort.

Além disso, insertion sort é adaptativo, ou seja, aumenta a eficiência em dados que já estão parcialmente ordenados e consegue ordenar uma lista à medida que a recebe (não precisa da lista inteira para começar a ordenação).

Breve Análise dos Métodos Simples - iii

Selection sort é muito similar ao insertion sort, que geralmente possui mais vantagens. Entretanto, insertion requer mais operações de escrita o que faz com que o selection sort seja preferido em casos em que a escrita na memória é mais caro que a leitura, com em EEPROMs ou memórias flash.

Assim, para pequeno volume de dados, em dispositivos com limitação de escrita (um Arduino ou ESP32, por exemplo), o selection sort é uma opção a ser considerada.

Breve Análise dos Métodos Simples - iv

Infelizmente, quanto ao bubble sort, pesa a favor apenas o fato que ele é popular e relativamente fácil de entender. Alguns autores, como Owen Astrachan, chegam inclusive a recomendar que ele não seja ensinado, o que está sendo adotado nesta disciplina.

Sua única vantagem sobre a maioria dos outros algoritmos é a capacidade de detectar que os dados estão ordenados, interrompendo o processo. Mas exceto em poucos casos usuais, na maioria das vezes ele **é menos eficiente** que o insertion sort ou selection sort, mesmo com essas melhorias.

Breve Análise dos Métodos Simples - iv

Infelizmente, quanto ao bubble sort, pesa a favor apenas o fato que ele é popular e relativamente fácil de entender. Alguns autores, como Owen Astrachan, chegam inclusive a recomendar que ele nã nesta disciplina.

Em síntese:

não use o bubble sort!

Sua única tmos é a capacidade de detectar que os dados estão ordenados, interrompendo o processo. Mas exceto em poucos casos usuais, na maioria das vezes ele **é menos eficiente** que o insertion sort ou selection sort, mesmo com essas melhorias.

Análise do Shell Sort

Shell sort pode ser implementado com pouco código e tem a vantagem de não utilizar uma pilha de chamadas (*call stack*), o que ocorre em métodos recursivos, como quick sort ou merge sort.

Assim, algumas implementações de ordenação em bibliotecas padrões utilizam o shell sort em sistemas embarcados. O kernel do Linux, utiliza uma implementação do Shell sort para ordenação, por razões similares.

Análise do Shell Sort

Shell sort pode ser implementado com pouco código e [illegible] chamadas (*call*) recursivas com

As [illegible] nas bibl[illegible] pac[illegible] embarcados. O kernel do Linux, utiliza uma implementação do Shell sort para ordenação, por razões similares.

É uma boa opção a ser considerada, caso o volume de dados não seja muito grande!

Sobre o Material

Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Janderson Rodrigo de Oliveira
- Joaquim Quinteiro Uchôa
- Juliana Galvani Greghi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.