# Estruturas de Dados

MÉTODOS DE ORDENAÇÃO

# Métodos de ordenação

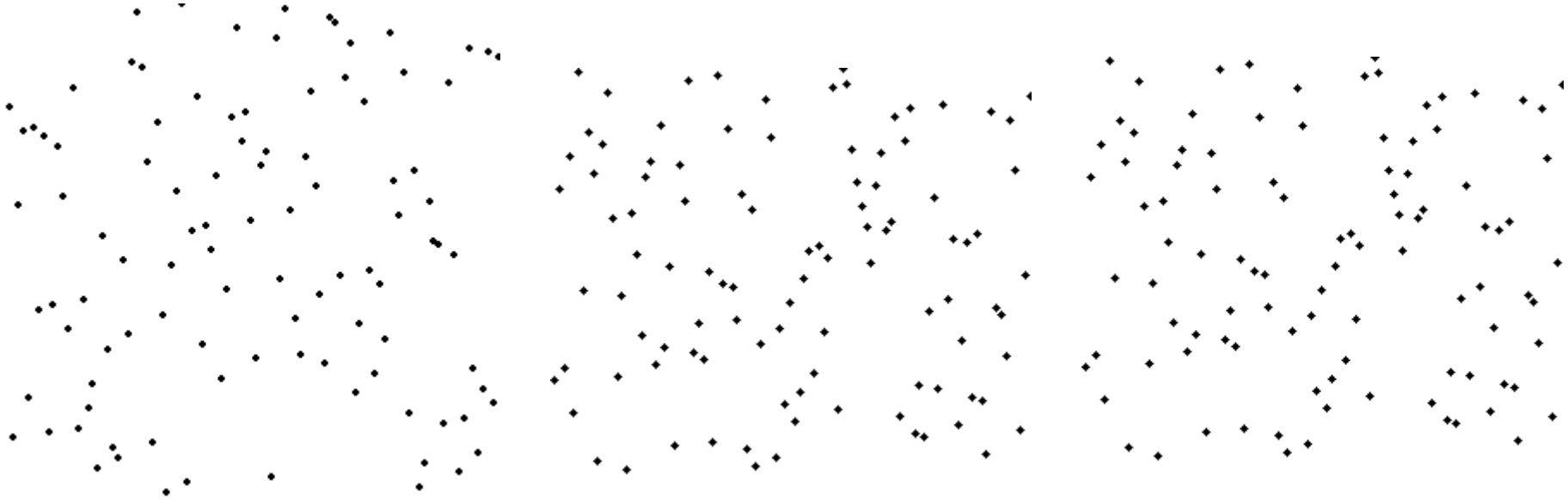
- Ordenar corresponde ao processo de rearranjar um conjunto de objetos em um ordem ascendente ou descendente.
- O objetivo principal da ordenação é facilitar a recuperação dos itens do conjunto ordenado posteriormente.
- A atividade de ordenação está presente na maioria das aplicações onde os objetos tem que ser pesquisados e recuperados

# Métodos de ordenação

- Os métodos de ordenação são classificados em dois grandes grupos.
  - Ordenação interna: Conjunto de dados a ser ordenado cabe todo na memória principal.
  - Ordenação Externa: Conjunto de dados não cabe na memória principal, tendo que ser armazenado, por exemplo, em disco.
- O fator predominante na escolha do algoritmo de ordenação é o tempo.

# Métodos de Ordenação

- Bolha (*BubbleSort*)
- Seleção (*SelectionSort*)
- Inserção (*InsertionSort*)
- Rápido (QuickSort)



# Método bolha

- Os elementos vão “borbulhando” a cada iteração do método até a posição correta para ordenação da lista.
- O método pode parar quando nenhum elemento borbulhar/trocar de posição.
- Como os elementos são trocados (borbulhados) frequentemente, há um alto custo de troca de elementos.

# Exemplo do método bolha

Suponha que se deseja classificar em ordem crescente o seguinte vetor de chaves [28, 26, 30, 24, 25].

**Primeira Varredura**

28 26 30 24 25 compara par (28, 26): **troca**
26 28 30 24 25 compara par (28, 30): não troca
26 28 30 24 25 compara par (30, 24): **troca**
26 28 24 30 25 compara par (30, 25): **troca**
26 28 24 25 **30** Maior chave em sua posição definitiva

fim da primeira varredura

# Exemplo do método bolha

**Segunda Varredura**

26 28 24 25 30 compara par (26, 28) : não troca
26 28 24 25 30 compara par (28, 24) : **troca**
26 24 28 25 30 compara par (28, 25) : **troca**
26 24 25 28 30 (não precisa comparar)

**Terceira Varredura**

26 24 25 28 30 compara par (26, 24) : **troca**
24 26 25 28 30 compara par (26, 25) : **troca**
24 25 26 28 30 (não precisa comparar)

Durante a quarta varredura, nenhuma troca ocorrerá e a execução do algoritmo terminará.

# Análise de Desempenho

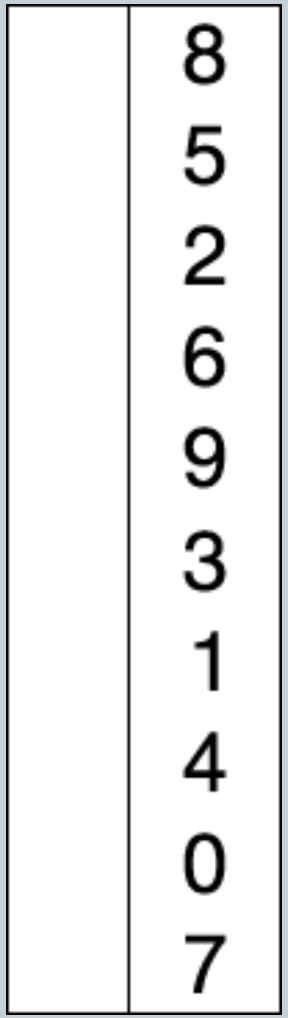
- Melhor caso
  - Quando o vetor já se encontra ordenado → nenhuma troca ocorre na primeira varredura.
  - Custo linear: n - 1 comparações
- Pior caso
  - Quando o vetor se encontra na ordem inversa a desejada.
  - A cada varredura apenas uma chave será colocada em sua posição definitiva.

# Código em C

```
void bubblesort(int vet[], int n) {
 int i, j, cond, temp;
 cond = 1;
 for (i=n-1; (i >= 1) && (cond == 1); i--) {
     cond = 0;
     for (j=0; j < i ;j++) {
         if (vet[j+1] < vet[j]) {
             temp = vet[j];
             vet[j] = vet[j+1];
             vet[j+1] = temp;
             cond = 1;
         }
     }
 }
}
```

# Método de Seleção

- O método de Seleção é um dos algoritmos mais simples de ordenação, cujo princípio de funcionamento é o seguinte:
- Selecione o menor (ou maior) item da lista em seguida troque-o com o que está na primeira posição da lista
- Repita esta operação com os n-1 itens restantes, depois com os n-2 ate que reste apenas um elemento



# Análise de Desempenho

- Vantagens:
  - Custo linear no tamanho da entrada para o número de movimentos de registros.
  - É muito interessante para arquivos pequenos.
- Desvantagens:
  - O fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.
  - O algoritmo não é **estável.**

# Código

```
void selection_sort(int num[], int tam) {
  int i, j, min;
  for (i = 0; i < (tam-1); i++) {
       min = i;
       for (j = (i+1); j < tam; j++) {
              if(num[j] < num[min]) {
                     min = j;
              }
       }
       if (i != min) {
              int swap = num[i];
              num[i] = num[min];
              num[min] = swap;
       }
  }
}
```

# Método de Inserção

- Algoritmo utilizado pelo jogador de cartas
  - As cartas são ordenadas da esquerda para direita uma por uma.
  - O jogador escolhe a segunda carta e verifica se ela deve ficar antes ou na posição que está.
  - Depois a terceira carta é classificada, deslocando-a até sua correta posição
  - O jogador realiza esse procedimento até ordenar todas as cartas

# Análise de Desempenho

- Análise
  - Para arquivos já ordenados o algoritmo tem um custo de O(n).
    - Logo é um método recomendado quando a lista está parcialmente ordenada (pior caso, ordem reversa)
  - Bom método quando se deseja adicionar uns poucos itens a um arquivo já ordenado e depois obter uma lista ordenada (neste caso custo é linear)
  - Algoritmo quase tão simples quanto o algoritmo de ordenação por seleção

# Código em C

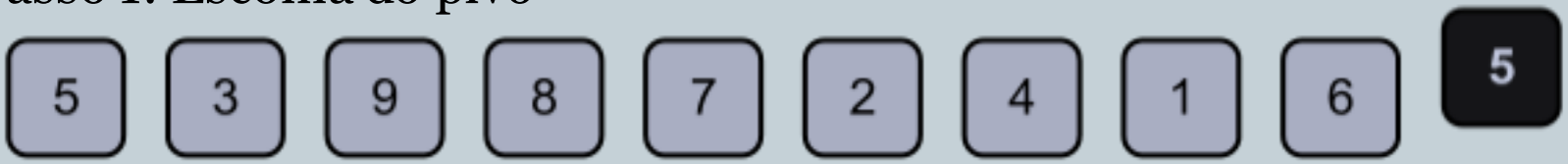
```
void Insertion(int n, int vetor[]){
  int j,i,key;
  for(j = 1; j < n; j++) {
      key = vetor[j];
      i = j - 1;
      while(i >= 0 && vetor[i] > key) {
            vetor[i + 1] = vetor[i];
            i = i - 1;
      }
      vetor[i + 1] = key;
  }
}
```

# Quick Sort

- Método de ordenação rápido e eficiente
- O algoritmo de Quick sort adota a técnica de divisão por conquista, e utiliza os seguintes passos:
  1. Seleciona um elemento da lista, denominado pivô;
  2. Rearranja a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada partição;
  3. Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores.

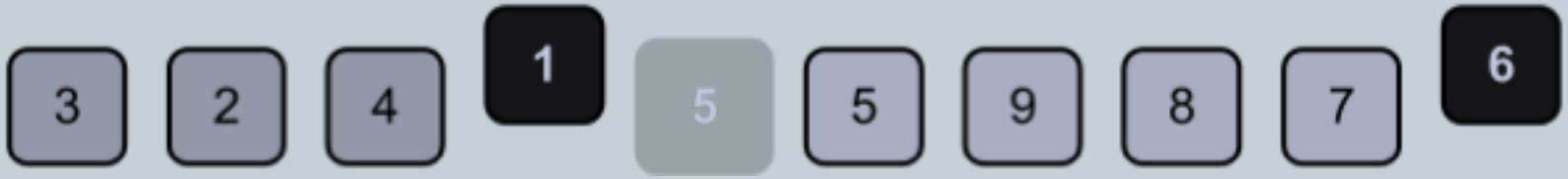
# Quick Sort

- Passo 1: Escolha do pivô




- Passo 2: Valores menores deslocados para esquerda, valores maiores para a direita


- Passo 3: Repetir a partir do passo 1 com as duas sublistas



# Análise de Desempenho

- Vantagens:
  - Implementação recursiva é simples
  - Em geral sua eficiencia é a mesma do método Merge Sort
    - O(n*log(n))
  - Solução elegante e mais simples do que no método Merge Sort
- Desvantagens:
  - Tão lenta quanto o método BubleSort no pior Caso.
    - $O(n^2)$
  - Implementação iterativa não é simples.
  - Existem algoritmos mais rápidos dependendo dos tipos de dados

# Código...

```
#include <cstdlib>
#include <iostream>
using namespace std;
int partition(int vec[], int left, int right) {
   int i, j;
   i = left;
   for (j = left + 1; j <= right; ++j) {
         if (vec[j] < vec[left]) {
                ++i;
                int aux = vec[i];
                vec[i] = vec[j];
                vec[j] = aux;
         }
   }
   int aux = vec[left];
   vec[left] = vec[i];
   vec[i] = aux;
   return i;
} //...
```

# Código

```
void quicksort(int vec[], int left, int right) {
  int r;
  if (right > left) {
      r = partition(vec, left, right);
      quicksort(vec, left, r - 1);
      quicksort(vec, r + 1, right);
  }
}
```