**1- Introdução**

**O que é algoritmo?**

Algoritmos podem ser definidos como uma lista de tarefas para ser executada ou para resolver algum problema. E como qualquer lista, os algoritmos devem seguir o padrão e sempre devem ser finitos.

Tudo que é feito no computador é executado por um algoritmo, sendo assim, temos sempre que mantermos algoritmos limpos e bem escritos para que assim a máquina faça o melhor trabalho possível.

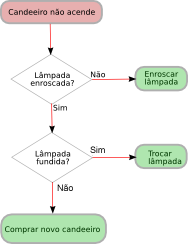
Os algoritmos são geralmente representados de três formas, que são elas:

* Forma de descrição narrativa;
* Fluxograma;
* Linguagem Algoritma

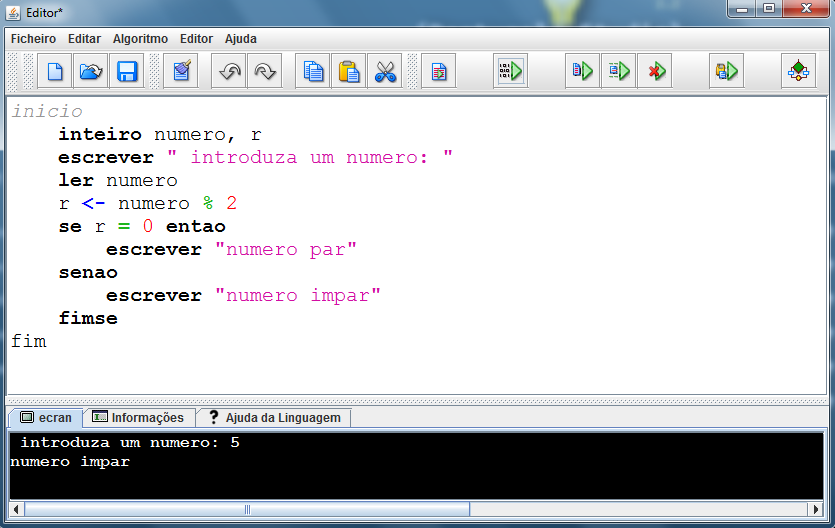
Exemplo de descrição narrativa:



Exemplo de fluxograma:



Exemplo de linguagem algoritma:



**Algoritmos de Ordenação**

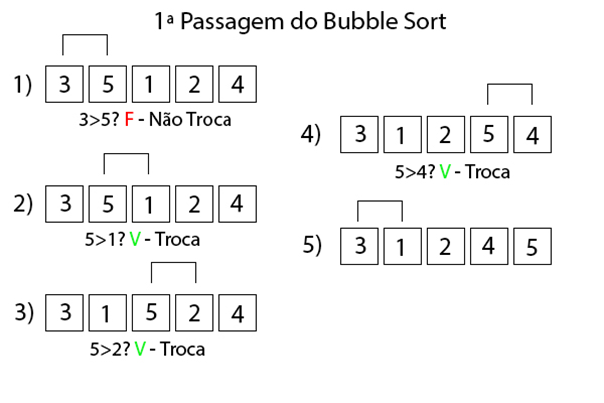
Algoritmos de ordenação, dentro do ambiente de computação serve para posicionar componentes de uma série em uma ordem pré-definida. Resumindo, esse algoritmo faz a ordenação de forma parcial ou de forma completa. São divididos em dois, que são:

* **Ordenação interna:** Onde os componentes que vão ser ordenados cabem na memória principal e qualquer arquivo é acessado diretamente.
* **Ordenação externa:** Onde os componentes que vão ser ordenados não cabem na memória principal e os arquivos são acessados em grandes blocos ou em forma de sequência.

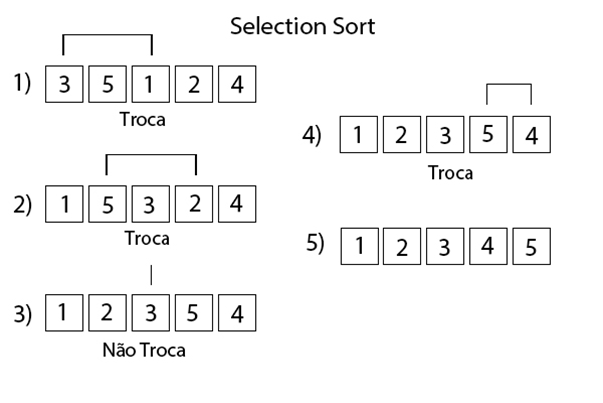
Aqui na introdução falaremos de apenas três que são: Bubble Sort, Selection Sort e Quick Sort.

**Bubble Sort**

Esse algoritmo é considerado o mais simples, e pouco eficiente. No bubble sort cada item da posição x será equiparado com o item da posição x + 1, logo, um item da posição 2 será equiparado com o elemento da posição 3. Se o item da posição 2 for maior do que o da posição 3 eles trocarão de posição e assim sucessivamente. Sendo executado desse jeito o algoritmo terá que ser percorrido várias vezes e assim não é tão eficiente para listas muito grandes.

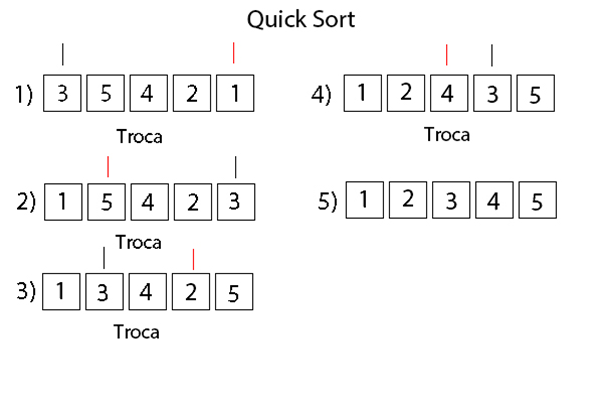


**Selection Sort**

O selection sort é baseado em sempre passar o menor termo do vetor para a primeira posição, também pode ser o maior tudo depende da ordem que foi pedida pelo usuário, depois o segundo menor termo para a segunda posição e assim sequencialmente, até os dois termos finais

**Quick Sort**

Esse algoritmo é o mais eficiente na ordenação por comparação. Quando escolhe um termo chamado pivô, partindo disso é organizada a lista para que todos os números antecedentes a ele sejam menores que ele, e todos os números seguintes a ele sejam maiores que ele. Quando isso chegar ao final o termo pivô está na posição final, e os dois grupos que estavam desordenados recursivamente e assim sofreram o mesmo processo até que a lista esteja em ordem.



**2- Referencial Teórico**

**Quais algoritmos foram escolhidos?**

Os algoritmos escolhidos para o desenvolvimento do trabalho foram:

* Bubble Sort
* Selection Sort
* Insertion Sort

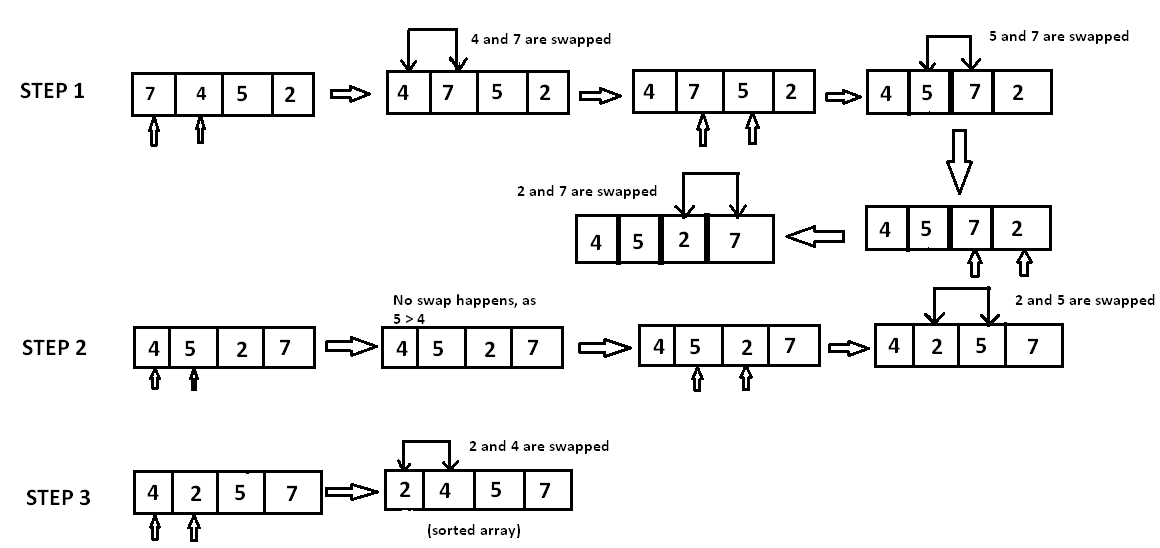
Foram escolhidos esses algoritmos com a intenção de demonstrar a evolução da complexidade lógica de cada um deles, e quais suas principais diferenças. Sendo assim, escolhi algoritmos muito simples e outros mais complexos.

Apresentando e explicando um pouco mais dos algoritmos que foram escolhidos para o desenvolvimento do trabalho, vamos começar pelo mais simples e ir aumentando a complexidade dos algoritmos, logo começaremos pelo *Bubble Sort*.

**2.1 Bubble Sort**

A lógica desse algoritmo pode ser considerada bem simples em comparação com a dos outros. Expondo mais um pouco sobre sua lógica, ela pode ser aplicada tanto em *arrays* quanto em *listas dinâmicas*.

É dada uma certa sequência de termos e o objetivo do algoritmo é ordená-los de forma decrescente, logo, a lógica que será aplicada pelo *Bubble Sort*, é que a posição do atual do termo é comparado com a posição seguinte, se a posição atual for maior que a posição seguinte é feita uma troca de posição dos termos. Caso não seja maior não é realizada nenhuma troca, apenas passa para o termo seguinte e assim ele percorre toda a lista.



Caso o objetivo seja ordenar os termos da lista de forma crescente, logo, a posição atual do termo é comparada com a posição seguinte, caso a posição atual seja menor que a posição seguinte é realizada a troca entre os termos. Caso não seja menor não é realizada nenhuma troca, e o algoritmo segue percorrendo toda a lista.

Aqui um exemplo desse algoritmo na linguagem Java:

**2.2 Selection Sort**

public static void main(String args[]){

    int[] vet = {8, 9, 3, 5, 1};

    int aux = 0;

    int i = 0;

    System.out.println("Vetor desordenado: ");

    for(i = 0; i<5; i++){

        System.out.println(" "+vet[i]);

    }

    System.out.println(" ");

    for(i = 0; i<5; i++){

        for(int j = 0; j<4; j++){

            if(vet[j] > vet[j + 1]){

                aux = vet[j];

                vet[j] = vet[j+1];

                vet[j+1] = aux;

            }

        }

    }

    System.out.println("Vetor organizado:");

    for(i = 0; i<5; i++){

        System.out.println(" "+vet[i]);

    }

}

A lógica desse algoritmo persiste em ordenar uma lista “escolhendo” a cada vez os menores termos possíveis, e assim ordená-los da esquerda para a direita.

Um pouco sobre as vantagens desse algoritmo estão listadas na facilidade de ser implementado e também ocupa menos memória quando comparado com outros algoritmos mais complexos como o *Quick Sort* que tem a lógica de “dividir e conquistar”.

Agora uma das principais desvantagens desse algoritmo é o seu tempo de execução já que em todos os casos é (O(n²)). O que o torna uma má opção para objetos que tem o número de termos muito grande.



Esse algoritmo é composto por dois laços de repetição, o primeiro ou externo é responsável por controlar o índice inicial e o segundo ou interno é responsável por percorrer toda a lista. Falando sobre o processo de seleção que o algoritmo executa, é basicamente ele busca e obtém o menor termo da lista que está desordenada. Abaixo algoritmo implementado na linguagem Java:

public static void main(String[] args) throws IOException {

   int quantidade = 10000;

   int[] vetor = new int[quantidade];

   for (int i = 0; i < vetor.length; i++) {

      vetor[i] = (int) (Math.random()\*quantidade);

   }

   long tempoInicial = System.currentTimeMillis();

   selectionSort(vetor);

   long tempoFinal = System.currentTimeMillis();

   System.out.println("Executado em = " + (tempoFinal - tempoInicial) + " ms");

 }

 public static void selectionSort(int[] array) {

  for (int fixo = 0; fixo < array.length - 1; fixo++) {

    int menor = fixo;

    for (int i = menor + 1; i < array.length; i++) {

       if (array[i] < array[menor]) {

          menor = i;

       }

    }

    if (menor != fixo) {

      int t = array[fixo];

      array[fixo] = array[menor];

      array[menor] = t;

    }

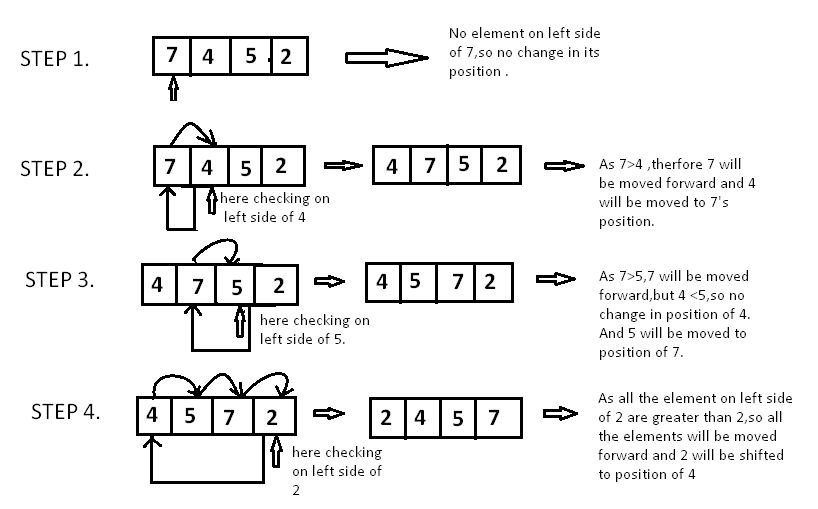
  }

}

**2.3 Insertion Sort**

Esse algoritmo é o mais complexo em comparação com os outros dois que foram escolhidos. Esse método de ordenação em algumas explicações é muito comparado com o modo com que umas pessoas formam o seu baralho.

Por trás desse algoritmo, quando é dado uma lista ele constituí uma matriz final com um termo de cada vez, ou seja, uma inserção por vez.



O *Insertion Sort* se torna um bom algoritmo quando é desejado acrescentar novos termos em uma lista já ordenada, já que seu custo é linear. Outra de suas vantagens é que ele é um algoritmo que tem uma implementação considerada simples e com isso sua leitura e manutenção também se tornam simples.

Abaixo um exemplo do algoritmo implementado na linguagem Java:

public static void insertionSort(int array[]) {

int n = array.length;

for (int j = 1; j < n; j++) {

int key = array[j];

int i = j-1;

while ( (i > -1) && ( array [i] > key ) ) {

array [i+1] = array [i];

i--;

}

array[i+1] = key;

}

}