UNIVERSIDADE PAULISTA (UNIP)

CIÊNCIA DA COMPUTAÇÃO

**ATIVIDADES PRÁTICAS SUPERVISIONADAS**

**DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE ALGORITMOS DE ORDENAÇÃO DE DADOS**

ANDREICK GOMES MAIA - F044289

BEATRIZ ALMEIDA DE LIMA SOUZA - N459JB4

MAICOLN VINICIUS DOS S. SILVA - F0551I7

MATEUS CARVALHO DA SILVA - N508620

SÃO PAULO

2020

**Sumário**

[Objetivo 3](#_Toc56462323)

[Introdução 4](#_Toc56462324)

[Métodos de Ordenação 5](#_Toc56462325)

[Método Selection Sort 6](#_Toc56462326)

[Método Quick Sort 7](#_Toc56462327)

[Método Insertion Sort 8](#_Toc56462328)

[Desenvolvimento do Sistema 9](#_Toc56462329)

[Geração e Obtenção de Dados 9](#_Toc56462330)

[Ordenação de Dados 10](#_Toc56462331)

[Resultados e Comparativos 15](#_Toc56462332)

[Conclusão 18](#_Toc56462333)

[Referências Bibliográficas 19](#_Toc56462334)

[Código Fonte 20](#_Toc56462335)

[Fichas 30](#_Toc56462336)

# Objetivo

O objetivo deste trabalho é aplicar os conceitos de Estruturas de Dados, mais precisamente, as técnicas de ordenação de dados, utilizando-se de algoritmos já conhecidos. Efetuaremos testes práticos, medindo a eficiência de três algoritmos individualmente e, posteriormente, comparando-os entre si.

O plano de fundo para este trabalho é o geoprocessamento de imagens da floresta amazônica, que permite a fiscalização de ações de crimes ambientais. Os satélites geram cerca de 100 mil imagens de toda a região a cada 24 horas, essas imagens devem ser armazenadas o catalogadas. Neste trabalho, desenvolveremos um sistema computacional completo que obtenha os dados catalogados das imagens capturadas dos satélites, efetuando a ordenação e comparando os desempenhos entre eles, por meio de três técnicas de ordenação de dados. A unidade de medida para efeito de comparação será o tempo total de ordenação.

# Introdução

Desde os primórdios, com o surgimento da matemática, a organização de dados se faz necessária para a resolução de problemas. Como preza a estratégia algorítmica: “Primeiro coloque os números em ordem. Depois decidimos o que fazer.”

Na ciência da computação, uma Estrutura de Dados é um meio de armazenar e organizar dados em um computador, de modo que possam ser usados eficientemente, facilitando sua busca e modificação. Na computação existe uma série de algoritmos que utilizam diferentes técnicas de ordenação para organizar um conjunto de dados, eles são conhecidos como Métodos de Ordenação ou Algoritmos de Ordenação. Quando estes dados estão organizados de forma coerente, caracterizam uma estrutura de dados.

Algoritmos de Ordenação permitem colocar os elementos de uma dada sequência em uma certa ordem. Ou seja, efetuam sua ordenação completa ou parcial. O objetivo da ordenação é facilitar a recuperação dos dados de uma lista. Os mais populares algoritmos de ordenação são: *Insertion sort, Selection sort, Bubble sort, Comb sort, Quick sort, Merge sort, Heap sort* e *Shell sort*.

Neste trabalho, serão abordados os algoritmos *Selection Sort, Quick sort e Insertion sort,* explicando o funcionamento e comparando a eficiência de cada um deles.

# Métodos de Ordenação

Os métodos de ordenação podem ser classificados em métodos de ordenação interna ou ordenação externa, onde a diferença está, basicamente, no armazenamento e no acesso aos registros.

**Ordenação Interna:** os elementos a serem ordenados cabem na memória principal e qualquer registro pode ser imediatamente acessado.

**Ordenação Externa:** os elementos a serem ordenados não cabem na memória principal e os registros são acessados sequencialmente ou em grandes blocos.

Nos algoritmos de ordenação as medidas de complexidade relevantes são:

* Número de comparações C(n) entre chaves.
* Número de movimentações M(n) dos registros dos vetores.
* Onde n é o número de registros.

Dentro da ordenação interna temos os Métodos Simples e os Métodos Eficientes.

**Métodos simples:** são adequados para pequenos vetores, são programas pequenos e fáceis de entender. Possuem complexidade C(n) = O(n²), ou seja, requerem O(n²) comparações.

Exemplos*: Insertion Sort, Selection Sort, Bubble Sort, Comb Sort*.

**Métodos Eficientes:** são adequados para vetores maiores e mais complexos. Requerem menos comparações, de complexidade C(n) = O(n log n).

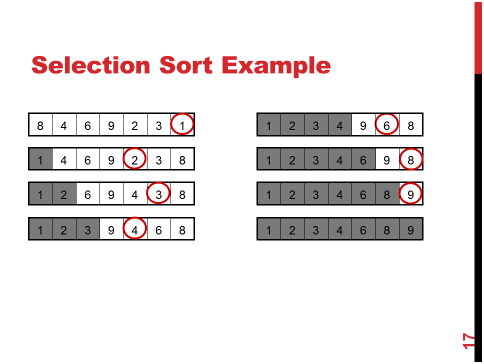
Exemplos: *Shellsort, Heapsort, Mergesort, Quicksort*.

A seguir, será explicado o funcionamento os algoritmos de ordenação interna: *Selection Sort, Quick sort* e *Insertion sort*.

## Método Selection Sort

Na ordenação por seleção ou *Selection Sort*, é escolhido um número a partir do primeiro e comparado com os números a partir da sua direita. Quando encontrado um número menor, o número escolhido ocupa sua posição. Este número encontrado será o próximo número escolhido. Caso não for encontrado nenhum número menor que este escolhido, ele é colocado na posição do primeiro número escolhido, e o próximo número à sua direita vai ser o escolhido para fazer as comparações. É repetido esse processo até que a lista esteja ordenada.

O *Selection Sort* é notado por sua simplicidade e tem vantagens de desempenho sobre algoritmos mais complicados em certas situações, particularmente onde a memória auxiliar é limitada.

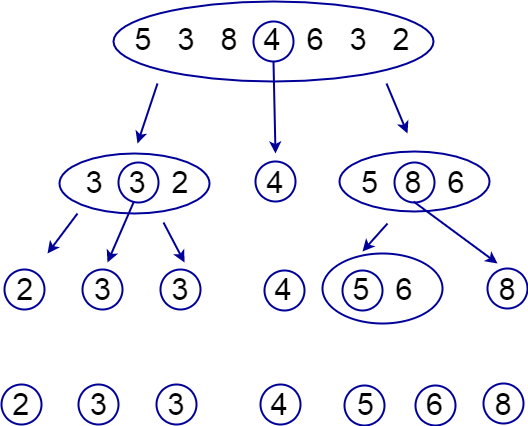


1. A partir do primeiro elemento, pesquisamos o menor elemento da matriz, e o substituímos pelo elemento na primeira posição.
2. A partir do primeiro elemento, pesquisamos o menor elemento da matriz, e o substituímos pelo elemento na primeira posição.
3. Em seguida vamos para a segunda posição e percorremos a lista procurando o menor elemento presente no sub-array, partindo do índice atual até o ultimo, inserindo o segundo menor valor na segunda posição.
4. Isso se repete até tudo estar em ordem.

Para todos os casos possui complexidade C(n) = O(n²) e não é um algoritmo estável.

## Método Quick Sort

O Algoritmo Quick sort, criado por C. A. R. Hoare em 1960, é o método de ordenação interna mais rápido que se conhece para uma ampla variedade de situações. É um algoritmo de comparação que emprega a estratégia de “divisão e conquista”. A ideia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores. Os problemas menores são ordenados independentemente e os resultados são combinados para produzir a solução final.



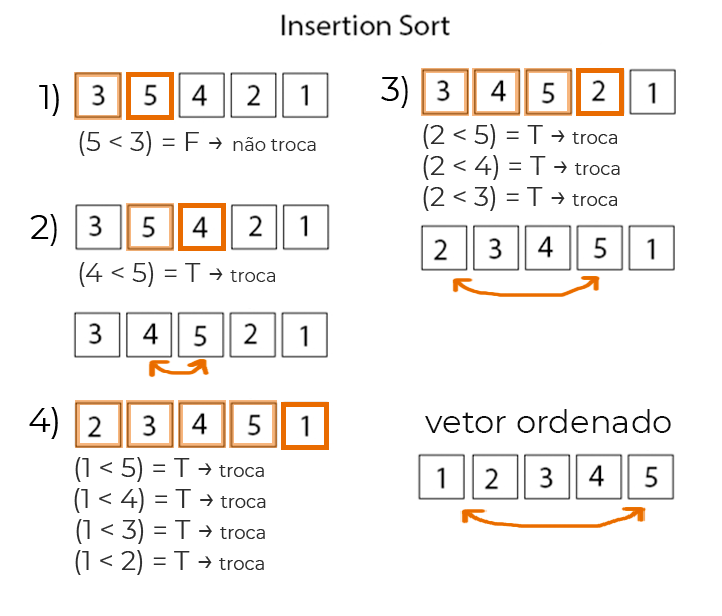
Basicamente a operação do algoritmo pode ser resumida na seguinte estratégia: divide sua lista de entrada em duas sub-listas a partir de um pivô, para em seguida realizar o mesmo procedimento nas duas listas menores até uma lista unitária.

A principal desvantagem deste método é que ele possui uma implementação difícil e delicada, um pequeno engano pode gerar efeitos inesperados para determinadas entradas de dados.

Possui complexidade C(n) = O(n²) no pior caso e C(n) = O(n log n) no melhor e médio caso. Não é um algoritmo estável.

## Método Insertion Sort

O *Insertion sort* é um algoritmo simples e eficiente quando aplicado em pequenas listas. Neste algoritmo a lista é percorrida da esquerda para a direita, à medida que avança vai deixando os elementos mais à esquerda ordenados.

O funcionamento do algoritmo é bem simples: consiste em cada passo a partir do segundo elemento selecionar o próximo item da sequência e colocá-lo no local apropriado de acordo com o critério de ordenação. É semelhante à forma que as pessoas usam para ordenar cartas em um jogo de baralho como o pôquer.

Possui complexidade C(n) = O(n) no melhor caso e C(n) = O(n²) no caso médio e pior caso. É considerado um método de ordenação estável, ou seja, a ordem relativa dos itens iguais não se altera durante a ordenação.

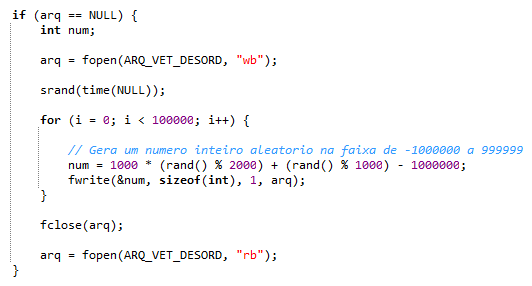
# Desenvolvimento do Sistema

O sistema computacional foi desenvolvido utilizando a linguagem de programação C e visa simular o processo de ordenação de 100 mil imagens geradas por satélite, que serão simbolicamente representadas por 100 mil números inteiros no intervalo de -1.000.000 a 999.999 armazenados em um vetor.

Além testar cada método com um vetor completamente desordenado o sistema também testa os métodos com vetores parcialmente ordenados com 25%, 50% e 75% de ordenação no começo, meio ou fim do vetor.

## Geração e Obtenção de Dados

Os números inteiros são gerados dentro do próprio programa utilizando a função “srand()” e “rand()” da biblioteca padrão da linguagem C “stdlib.h”. Estes valores são escritos em um arquivo binário chamado “vetor\_desordenado.txt”, bastando fazer a leitura deste arquivo para obter a sequência de números.



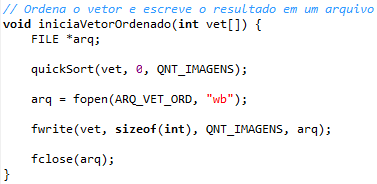
Trecho do código responsável pela geração dos dados

Uma vez criado o arquivo binário o programa faz a leitura dele e guarda os números em um vetor de inteiros de 100 mil posições. Se o arquivo ainda existir nas próximas execuções do programa ele não será sobrescrito, ao invés disso a etapa de geração dos números é pulada e somente é feita a leitura do arquivo, isto permite repetir os testes de ordenação usando uma mesma sequencia de números, para utilizar uma sequência diferente basta alterar o nome ou excluir o arquivo “vetor\_desordenado.txt” antes de executar o programa.

## Ordenação de Dados

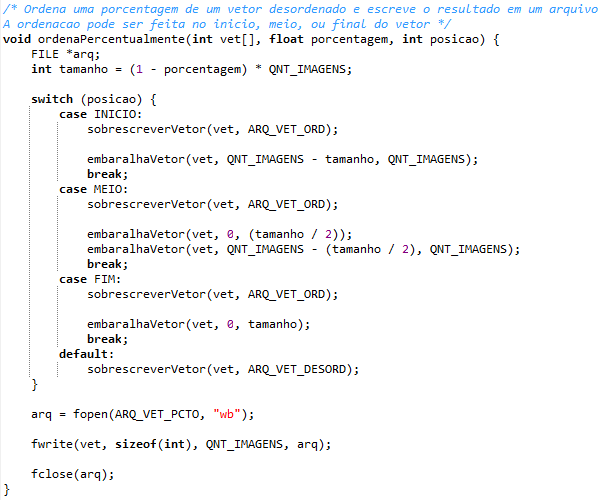
Para evitar a criação de mais de um vetor de 100 mil posições e repetir os processos de forma mais eficiente também são criados alguns arquivos binários auxiliares na realização dos testes.

O primeiro deles é o “vetor\_ordenado.txt” que contém a sequência de números obtidos pelo programa em ordem crescente, bastando aplicar o método de ordenação quicksort no vetor de inteiros do programa e guardar o resultado para obter o arquivo.



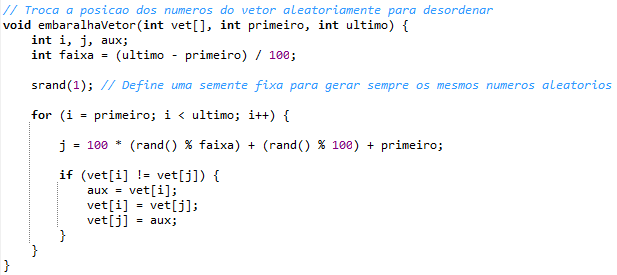
Função que escreve o arquivo “vetor\_ordenado.txt”

O segundo é o “vetor\_percentualmente\_ordenado.txt” que contém os números obtidos pelo programa com apenas uma porcentagem deles ordenados, o conteúdo deste arquivo varia durante a execução do programa de acordo com a porcentagem e a posição em que os números ordenados estarão. O algoritmo para criar este arquivo consiste em uma função que primeiro, com base na porcentagem, calcula quantos números devem estar desordenados e em seguida embaralha parte do vetor de números ordenados, portanto para manter o começo do vetor ordenado basta desordenar todo o restante, o mesmo é feito para manter o meio e o fim do vetor ordenados.



Função que escreve o arquivo “vetor\_percentualmente\_ordenado.txt”

O embaralhamento do vetor é feito trocando aleatoriamente os seus números de posição, e para manter um controle sobre como o vetor está sendo embaralhado é definida uma semente aleatória fixa para que a função “rand()” gere sempre a mesma sequência de números.



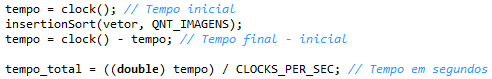
Função que embaralha uma parte de um vetor de inteiros

Para realizar todos os testes de ordenação são utilizados dois loopings encadeados, o primeiro itera sobre a porcentagem de ordenação e o segundo itera sobre a posição em que o vetor está ordenado, assim é possível fazer a comparação dos três métodos de ordenação em todas as situações propostas neste trabalho.



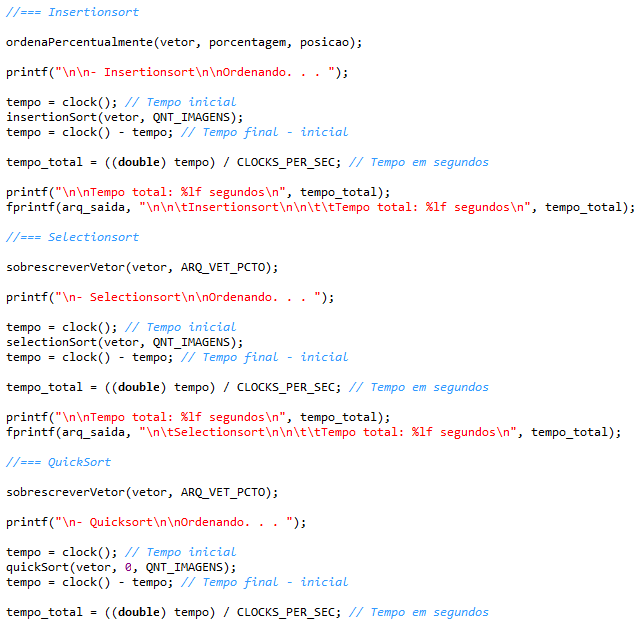
Estruturas de repetição para fazer as comparações

Para calcular o tempo de ordenação dos métodos é necessária a biblioteca padrão da linguagem C “time.h”, primeiro utiliza-se uma variável do tipo “clock\_t” e a função “clock()” para calcular o tempo atual do sistema em clocks antes de iniciar a ordenação e depois calcula-se a diferença entre o tempo final e o tempo inicial após o processo de ordenação, bastando fazer uma conversão do valor obtido em segundos para obter o tempo total de ordenação.



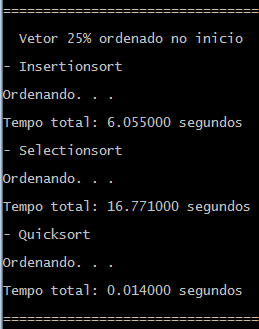
Algoritmo para calcular o tempo de execução do insertion sort

O algoritmo responsável por testar cada método de ordenação dentro dos loopings consiste em escrever o arquivo “vetor\_percentualmente\_ordenado.txt”, já utilizando o vetor que foi escrito para testar o primeiro método, bastando fazer a leitura do arquivo para retornar o vetor à sequência de números utilizada na primeira ordenação e testar o método seguinte, repetindo o processo para o último também.

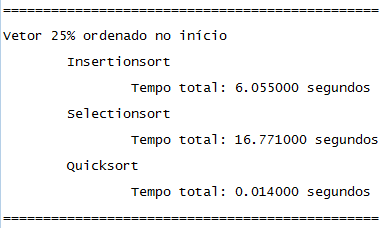


Algoritmo que testa os três métodos

Por fim, os resultados dos testes são exibido no prompt de comando, mostrando o tempo que cada método levou para realizar a ordenação, eles também são guardados em um arquivo texto chamado “resultado.txt”, sendo que a cada execução do programa os novos resultados vão sendo adicionados ao final do arquivo, mantendo os resultados anteriores.



Exemplo de exibição dos resultados no prompt de comando

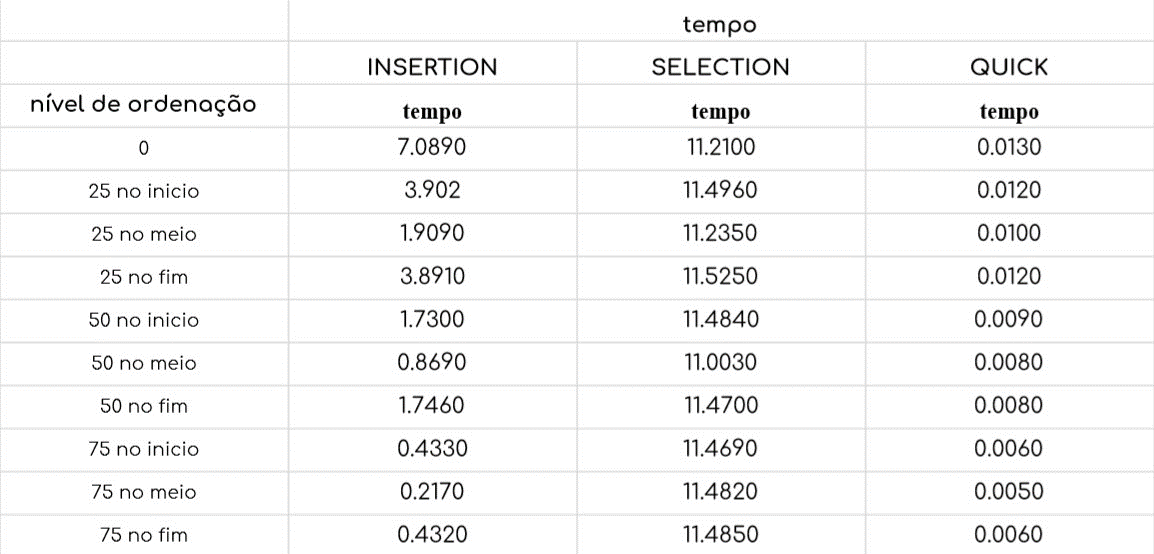


Exemplo de exibição dos resultados no "resultado.txt"

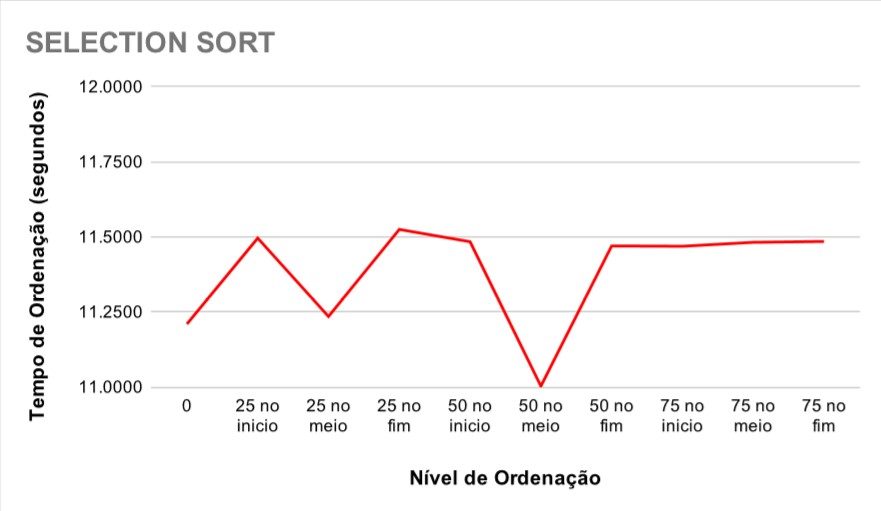
## Resultados e Comparativos

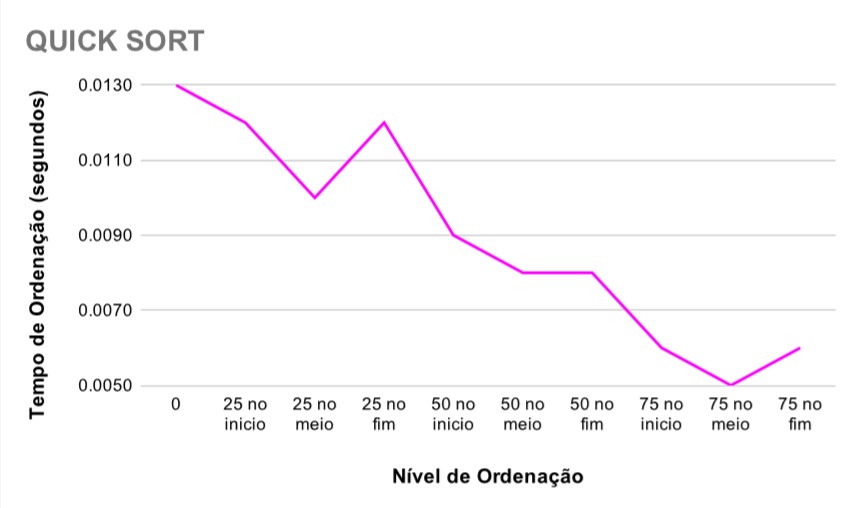
Os algoritmos *Insertion Sort*, *Selection Sort* e *Quick Sort* foram submetidos a um teste, para ordenar 100 mil dados em diferentes condições: 25% ordenados no início, 25% ordenados no meio, 25% ordenados no fim, 50% ordenados no início, 50% ordenados no meio, 50% ordenados no fim, 75% ordenados no início, 75% ordenados no meio e 75% ordenados no fim.

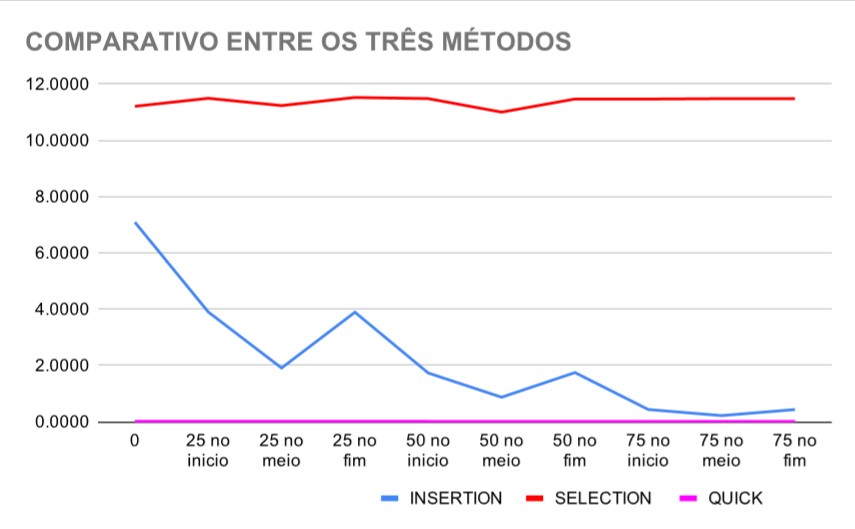
As médias dos resultados obtidos foram registradas em tabela e gráficos, disponíveis a seguir:











# Conclusão

Os três métodos, *Insertion Sort*, *Selection Sort* e *Quick Sort*, mostraram-se capazes de ordenar o vetor com 100 mil dados em poucos segundos. A unidade de medida para efeito de comparação será o tempo total de ordenação.

O método *Insertion Sort* obteve a maior variação, em relação ao nível de ordenação desse vetor. Seu maior tempo de execução foi de **7,08** segundos (vetor totalmente desordenado) e seu menor foi de **0,21** segundos (vetor 75% ordenado no meio). Pode-se afirmar que o tempo de execução diminui conforme o nível de ordenação aumenta. Além disso, esse algoritmo se mostra mais eficiente, preferencialmente, com níveis de ordenação no meio.

O método *Selection Sort* foi o mais demorado entre os três, permanecendo sempre na casa dos 11 segundos. Seu maior tempo de execução foi de **11,52** segundos (vetor 25% ordenado no fim) e seu menor foi de 11 segundos (50% ordenado no meio). Pelo o que foi constatado, o tempo de execução não diminui conforme o nível de ordenação aumenta. Também, a posição do nível de ordenação do vetor não parece ter influenciado no tempo de execução.

O método *Quick Sort* foi o método mais ágil entre os três. Seu maior tempo de execução foi de **0,013** segundos (vetor totalmente desordenado) e seu menor foi de **0,005** segundos (vetor 75% ordenado no meio). Consta-se que o tempo de execução diminui conforme o nível de ordenação aumenta. Existe uma pequena preferência do algoritmo, em relação a sua eficiência com níveis de ordenação no meio.

# Referências Bibliográficas

Bruno. (2013). *Algoritmos de ordenação: análise e comparação*. Fonte: DevMedia: https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261#:~:text=Algoritmo%20de%20ordena%C3%A7%C3%A3o%2C%20em%20ci%C3%AAncia,dos%20dados%20de%20uma%20lista.

Viana, D. (26 de Dezembro de 2016). *Conheça os principais algoritmos de ordenação*. Fonte: Treina Web: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/

# Código Fonte

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Nome do arquivo binario que contem um vetor desordenado com cem mil posicoes

#define ARQ\_VET\_DESORD "vetor\_desordenado.txt"

// Nome do arquivo texto que contem os resultados das ordenacoes

#define ARQ\_SAIDA "resultado.txt"

// Nome do arquivo binario com o vetor ordenado

#define ARQ\_VET\_ORD "vetor\_ordenado.txt"

// Nome do arquivo binario com o vetor percentualmente ordenado

#define ARQ\_VET\_PCTO "vetor\_percentualmente\_ordenado.txt"

// Quantidade de numeros que serao ordenados

enum {QNT\_IMAGENS = 100000};

// Indicadores de qual parte do vetor ordenar

enum {INICIO, MEIO, FIM};

void insertionSort(int arr[], int n) {

int i, j, key;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

void selectionSort(int arr[], int n) {

int i, j, min, aux;

for (i = 0; i < (n - 1); i++) {

min = i;

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

if (arr[j] < arr[min])

min = j;

}

if (arr[i] != arr[min]) {

aux = arr[i];

arr[i] = arr[min];

arr[min] = aux;

}

}

}

void quickSort(int arr[], int first, int last) {

int i, j, pivot, aux;

i = first;

j = last - 1;

pivot = arr[(first + last) / 2];

while(i <= j) {

while(arr[i] < pivot)

i++;

while(arr[j] > pivot)

j--;

if(i <= j) {

if (arr[i] != arr[j]) {

aux = arr[i];

arr[i] = arr[j];

arr[j] = aux;

}

i++;

j--;

}

}

if(j > first)

quickSort(arr, first, j + 1);

if(i < last)

quickSort(arr, i, last);

}

/\* Preenche o vetor com os valores contidos no arquivo do vetor desordenado

Cria o arquivo se ele nao existir \*/

void iniciaVetorDesordenado(int vet[]) {

FILE \*arq;

int i;

arq = fopen(ARQ\_VET\_DESORD, "rb");

if (arq == NULL) {

int num;

arq = fopen(ARQ\_VET\_DESORD, "wb");

srand(time(NULL));

for (i = 0; i < 100000; i++) {

// Gera um numero inteiro aleatorio na faixa de -1000000 a 999999

num = 1000 \* (rand() % 2000) + (rand() % 1000) - 1000000;

fwrite(&num, sizeof(int), 1, arq);

}

fclose(arq);

arq = fopen(ARQ\_VET\_DESORD, "rb");

}

i = fread(vet, sizeof(int), QNT\_IMAGENS, arq);

if (i != QNT\_IMAGENS) {

printf("Erro na leitura do arquivo "ARQ\_VET\_DESORD"!\n\n");

printf("Pressione a tecla Enter para continuar. . . ");

getchar();

exit(1);

}

fclose(arq);

}

// Ordena o vetor e escreve o resultado em um arquivo

void iniciaVetorOrdenado(int vet[]) {

FILE \*arq;

quickSort(vet, 0, QNT\_IMAGENS);

arq = fopen(ARQ\_VET\_ORD, "wb");

fwrite(vet, sizeof(int), QNT\_IMAGENS, arq);

fclose(arq);

}

// Preenche o vetor com os valores contidos no arquivo

void sobrescreverVetor(int vet[], char nome\_arq[]) {

FILE \*arq;

arq = fopen(nome\_arq, "rb");

fread(vet, sizeof(int), QNT\_IMAGENS, arq);

fclose(arq);

}

// Troca a posicao dos numeros do vetor aleatoriamente para desordenar

void embaralhaVetor(int vet[], int primeiro, int ultimo) {

int i, j, aux;

int faixa = (ultimo - primeiro) / 100;

srand(1); // Define uma semente fixa para gerar sempre os mesmos numeros aleatorios

for (i = primeiro; i < ultimo; i++) {

j = 100 \* (rand() % faixa) + (rand() % 100) + primeiro;

if (vet[i] != vet[j]) {

aux = vet[i];

vet[i] = vet[j];

vet[j] = aux;

}

}

}

/\* Ordena uma porcentagem de um vetor desordenado e escreve o resultado em um arquivo

A ordenacao pode ser feita no inicio, meio, ou final do vetor \*/

void ordenaPercentualmente(int vet[], float porcentagem, int posicao) {

FILE \*arq;

int tamanho = (1 - porcentagem) \* QNT\_IMAGENS;

switch (posicao) {

case INICIO:

sobrescreverVetor(vet, ARQ\_VET\_ORD);

embaralhaVetor(vet, QNT\_IMAGENS - tamanho, QNT\_IMAGENS);

break;

case MEIO:

sobrescreverVetor(vet, ARQ\_VET\_ORD);

embaralhaVetor(vet, 0, (tamanho / 2));

embaralhaVetor(vet, QNT\_IMAGENS - (tamanho / 2), QNT\_IMAGENS);

break;

case FIM:

sobrescreverVetor(vet, ARQ\_VET\_ORD);

embaralhaVetor(vet, 0, tamanho);

break;

default:

sobrescreverVetor(vet, ARQ\_VET\_DESORD);

}

arq = fopen(ARQ\_VET\_PCTO, "wb");

fwrite(vet, sizeof(int), QNT\_IMAGENS, arq);

fclose(arq);

}

int main() {

FILE \*arq\_saida;

int vetor[QNT\_IMAGENS];

clock\_t tempo;

double tempo\_total;

float porcentagem;

int posicao;

iniciaVetorDesordenado(vetor);

iniciaVetorOrdenado(vetor);

arq\_saida = fopen(ARQ\_SAIDA, "a");

printf("Comparacao de metodos de ordenacao com um vetor de %d posicoes\n\n", QNT\_IMAGENS);

fprintf(arq\_saida, "\tComparação de métodos de ordenação com um vetor de %d posições\n\n\n", QNT\_IMAGENS);

for (porcentagem = 0; porcentagem < 1; porcentagem += 0.25) {

for (posicao = INICIO; posicao <= FIM; posicao++) {

if (porcentagem == 0)

posicao = FIM + 1;

switch (posicao) {

case INICIO:

printf(" Vetor %.f%% ordenado no inicio", porcentagem \* 100);

fprintf(arq\_saida, "Vetor %.f%% ordenado no início", porcentagem \* 100);

break;

case MEIO:

printf(" Vetor %.f%% ordenado no meio", porcentagem \* 100);

fprintf(arq\_saida, "Vetor %.f%% ordenado no meio", porcentagem \* 100);

break;

case FIM:

printf(" Vetor %.f%% ordenado no fim", porcentagem \* 100);

fprintf(arq\_saida, "Vetor %.f%% ordenado no fim", porcentagem \* 100);

break;

default:

printf(" Vetor desordenado");

fputs("Vetor desordenado", arq\_saida);

}

//=== Insertionsort

ordenaPercentualmente(vetor, porcentagem, posicao);

printf("\n\n- Insertionsort\n\nOrdenando. . . ");

tempo = clock(); // Tempo inicial

insertionSort(vetor, QNT\_IMAGENS);

tempo = clock() - tempo; // Tempo final - inicial

tempo\_total = ((double) tempo) / CLOCKS\_PER\_SEC; // Tempo em segundos

printf("\n\nTempo total: %lf segundos\n", tempo\_total);

fprintf(arq\_saida, "\n\n\tInsertionsort\n\n\t\tTempo total: %lf segundos\n", tempo\_total);

//=== Selectionsort

sobrescreverVetor(vetor, ARQ\_VET\_PCTO);

printf("\n- Selectionsort\n\nOrdenando. . . ");

tempo = clock(); // Tempo inicial

selectionSort(vetor, QNT\_IMAGENS);

tempo = clock() - tempo; // Tempo final - inicial

tempo\_total = ((double) tempo) / CLOCKS\_PER\_SEC; // Tempo em segundos

printf("\n\nTempo total: %lf segundos\n", tempo\_total);

fprintf(arq\_saida, "\n\tSelectionsort\n\n\t\tTempo total: %lf segundos\n", tempo\_total);

//=== QuickSort

sobrescreverVetor(vetor, ARQ\_VET\_PCTO);

printf("\n- Quicksort\n\nOrdenando. . . ");

tempo = clock(); // Tempo inicial

quickSort(vetor, 0, QNT\_IMAGENS);

tempo = clock() - tempo; // Tempo final - inicial

tempo\_total = ((double) tempo) / CLOCKS\_PER\_SEC; // Tempo em segundos

printf("\n\nTempo total: %lf segundos\n", tempo\_total);

fprintf(arq\_saida, "\n\tQuicksort\n\n\t\tTempo total: %lf segundos\n", tempo\_total);

printf("\n================================\n\n");

fputs("\n===============================================\n\n", arq\_saida);

}

}

fclose(arq\_saida);

printf("Pressione a tecla Enter para continuar. . . ");

getchar();

return 0;

}

# Fichas

