**UNIVERSIDADE PAULISTA UNIP ANCHIETA**

Claudio Henrique Massone Lavado - G0631d2

Davi Arthur Gonçalves Silva – G511cb3

João Victor de Lelles Souza - N872870

Luis Carlos de Carvalho Junior – G0752A-3

Ruan Cardoso Moreira - N8303c5

**Atividades Práticas Supervisionadas (APS)**

Turma CC2P39 /CC3P

São Paulo - SP

2023

Código da atividade: 77B1

**Atividades Práticas Supervisionadas (APS)**

Turma CC4P / CC3P

Relatório final, apresentado a Universidade UNIP, como parte das exigências para a obtenção do título de Ciência da computação.

São Paulo, 15 de novembro de 2023.

BANCA EXAMINADORA

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. (Olavo Ito)

[Introdução 4](#_Toc1822564857)

[Referencial teórico 6](#_Toc46826505)

[Desenvolvimento 10](#_Toc2057711137)

[Resultados e Discussão 16](#_Toc154574937)

[Considerações finais 21](#_Toc975752512)

[Referências bibliográficas 23](#_Toc334936215)

[Código Fonte 23](#_Toc1038184074)

# **Introdução**

A crescente preocupação com a preservação do meio ambiente e a busca por soluções inovadoras têm impulsionado o desenvolvimento de tecnologias que possam monitorar e proteger ecossistemas de maneira eficiente.

Nesse contexto, a vigilância ambiental por meio de um sistema de geoprocessamento de imagens capturadas via satélite tem emergido como uma possível ferramenta na identificação e mitigação de impactos ambientais. Ao combinar avanços na área de sensoriamento remoto, sistemas de monitoramento e análise de dados por inteligência artificial, a utilização desse sistema proporciona uma visão abrangente e em tempo real de áreas naturais, permitindo a coleta de dados valiosos para a fiscalização de crimes ambientais, vigilância para controlar incêndios florestais e para gestão sustentável de ecossistemas.

Esta abordagem inovadora não apenas potencializa a capacidade de resposta a eventos críticos, como desmatamento e poluição, mas também contribui para a promoção de práticas de conservação mais eficazes, porém esta técnica necessita de tecnologias avançadas e grande poder computacional para analisar e armazenar os dados obtidos.

Como esse sistema utiliza enormes quantidades de dados, capturando cerca de 100mil imagens por dia, parte da dificuldade da implementação do mesmo é a organização e armazenamento dos dados analisados.

Após as imagens serem capturadas pelo satélite elas são enviadas para uma inteligência artificial que avalia cada imagem e dá uma porcentagem de chance de ela conter um possível acidente futuro.

Depois de analisada as imagens são organizadas em ordem de quais possuem maior chance de conter um perigo e então são poder ser reportadas para as autoridades responsáveis.

Em função disso este experimento tem como objetivo aliviar um pouco a carga computacional desse sistema utilizando e comparando diferentes algoritmos de ordenação para sabermos qual algoritmo é mais eficiente, levando em consideração uso de memória, tempo gasto, complexidade e facilidade de implementação

# **Referencial teórico**

Neste experimento usaremos quatro algoritmos de ordenação.

**Bubble sort**, ou o método da bolha, tem esse nome pois a cada troca vai subindo os maiores números em direção ao topo lembrando o movimento de bolhas.

A lógica por trás do Bubble Sort é bastante simples. O algoritmo percorre a lista de elementos várias vezes, comparando pares de elementos adjacentes e trocando-os caso estejam fora de ordem. O processo é repetido até que a lista esteja completamente ordenada.

No melhor caso, o algoritmo executa n operações relevantes, onde n representa o número de elementos do vetor. No pior caso e em média são feitas n² operações, (esperamos que o bubble sort não seja o mais eficiente, porém escolhemos ele para participar do experimento por sua simplicidade e facilidade de implementação.)

**Selection Sort** ou a ordenação por seleção funciona percorrendo todo o vetor, encontrando o menor valor e colocando-o na primeira posição, então repete o processo para o segundo menor valor e assim a diante.

O Selection Sort é interessante pois para ele não existe melhor ou pior caso, ele sempre faz a mesma quantidade de comparações (n² - n) / 2 de novo n sendo o tamanho do vetor.

Além de ser extremamente consistente o Selection Sort

Também é muito simples e fácil de implementar, porém ele não escala tão bem com listas enormes quanto o próximo algoritmo.

**Quick Sort** “*quicksort é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960, quando visitou a Universidade de Moscovo como estudante. Naquela época, Hoare trabalhou em um projeto de tradução de máquina para o National Physical Laboratory. Ele criou o quicksort ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais fácil e rápido. Foi publicado em 1962 após uma série de refinamentos.”* (<https://pt.wikipedia.org/wiki/Quicksort>)

O Quick Sort é o algoritmo mais complexo do experimento, mas esperamos que ele seja o mais rápido com uma quantidade média de comparações de (n log n).

Ele funciona com a ideia de dividir o vetor em vetores menores a partir de um pivô depois rearranja a lista para que todos os elementos menores que o pivô estejam à esquerda e todos os maiores a direita. Ao fim desse processo o pivô estará em sua posição correta e haverá duas sub listas não ordenadas. Essa operação é denominada partição. Agora repita o processo de partição nas sub listas até que todas as sub listas tenham tamanho 1, nesse ponto, remonte o vetor usando as sub listas e ele estará em ordem.

**Bogo Sort** o Bogo Sort “(*também conhecido como CaseSort ou Estou com Sort), é um algoritmo de ordenação extremamente ineficiente. É baseado na reordenação aleatória dos elementos. Não é utilizado na prática, mas pode ser usado no ensino de algoritmos mais eficientes. Seu nome veio do engraçado termo quantum bogodynamics e, ultimamente, a palavra bogus.*” (<https://pt.wikipedia.org/wiki/Bogosort>).

Não esperamos que o Bogo Sort seja remotamente útil, ele é praticamente o pior algoritmo de ordenação que existe, seu próprio nome, assim como sua performance é uma piada, escolhemos ele para o experimento para demonstrar a diferença máxima que os algoritmos podem ter.

O Bogo Sort é um algoritmo recursivo que funciona pegando o vetor e colocando todos os elementos em ordem aleatória e então verificando se está ordenado, se sim ele para a execução, se não repita o processo.

Apesar de ser o algoritmo mais simples e teoricamente capaz de ser o mais rápido e eficiente (ordenando qualquer vetor na primeira tentativa), não tem nenhuma utilidade pratica devido a sua natureza aleatória sendo capaz também de nunca ordenar qualquer vetor com mais de um elemento.

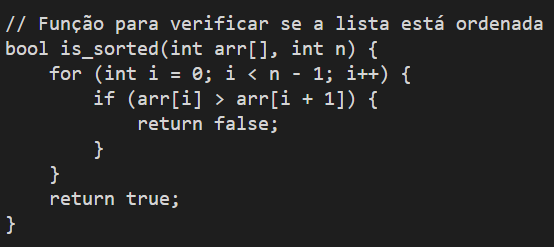
O tempo médio e número de comparações necessárias para completar sua execução é dado por (n \* n!).

# **Desenvolvimento**

O desenvolvimento do experimento começa com a criação de um ambiente estável. Usamos a ferramental do google colaboratory que cria uma máquina virtual capaz de gerar resultados relativamente constantes.

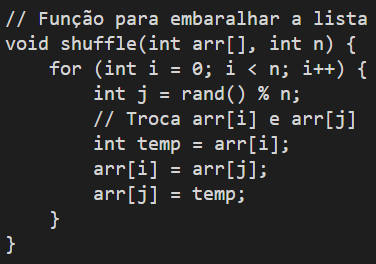
Dentro do ambiente virtual criamos um programa simples com apenas as funções necessárias para o experimento sendo elas

**is\_sorted:**



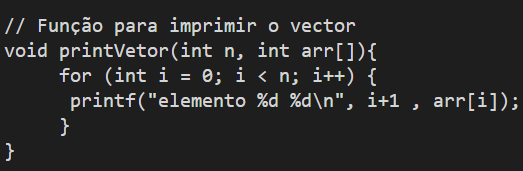
Essa função é usada dentro de outras para verificar se o vetor está ordenado percorrendo todo o vetor e verificando se cada elemento adjacente é maior que o anterior.

**Shuffle:**



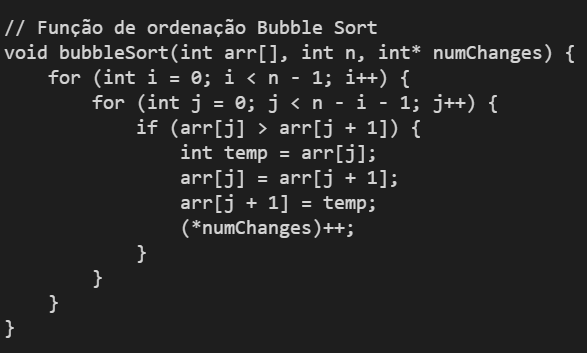
Embaralha os elementos do vetor. Recebe um vetor (arr) e o tamanho do vetor (n). Utiliza a função rand() para gerar números aleatórios e troca elementos aleatórios no vetor.

**Printvetor:**



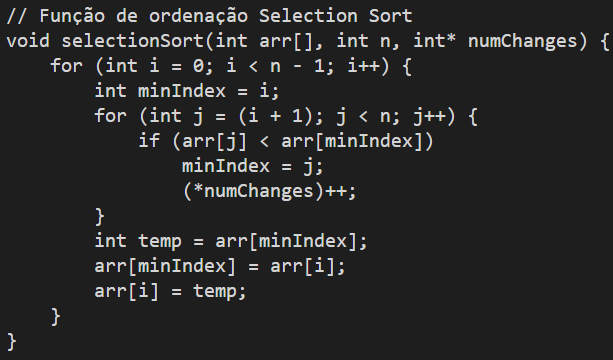
Imprime os elementos do vetor com seus índices. Recebe o tamanho do vetor (n) e o vetor (arr). Imprime cada elemento com o formato "elemento [índice] [valor]".

**bubbleSort:**



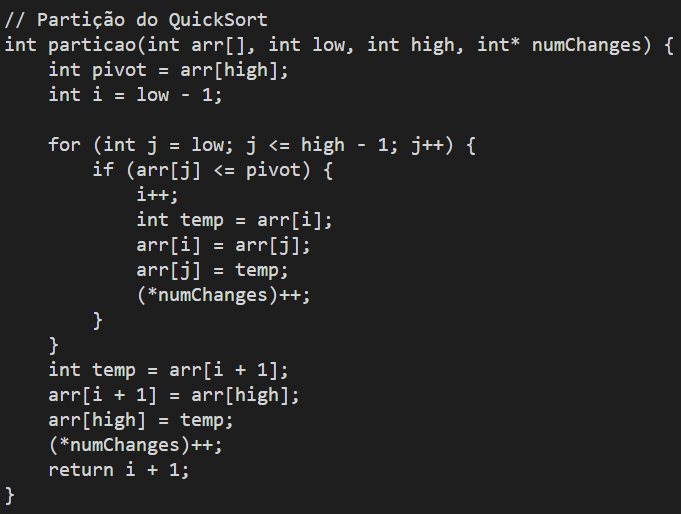
Implementa o algoritmo Bubble Sort. Recebe um vetor (arr), o tamanho do vetor (n) e um ponteiro para o número de trocas (numChanges). Ordena o vetor usando o método de comparação e troca de elementos adjacentes.

**selectionSort:**



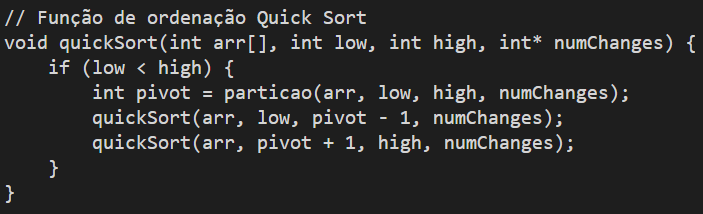
Implementa o algoritmo Selection Sort. Recebe um vetor (arr), o tamanho do vetor (n) e um ponteiro para o número de trocas (numChanges). Ordena o vetor selecionando repetidamente o menor elemento restante e trocando-o com o primeiro elemento não ordenado.

**particao:**



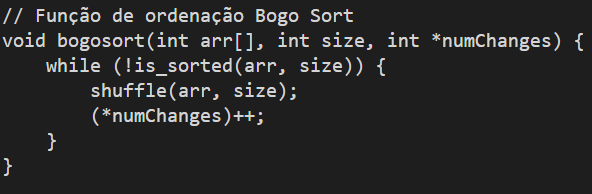
Esta função é parte do algoritmo Quick Sort. Ela recebe um vetor (arr), os índices inferior (low) e superior (high) e um ponteiro para o número de trocas (numChanges). A função escolhe o último elemento do vetor como pivô e rearranja os elementos de forma que todos os elementos menores que o pivô estejam à sua esquerda e todos os elementos maiores estejam à sua direita. A função retorna o índice do pivô após a rearranjo.

**quickSort:**



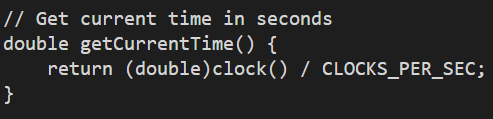
Implementa o algoritmo Quick Sort. Recebe um vetor (arr), os índices inferior (low) e superior (high), e um ponteiro para o número de trocas (numChanges). Usa a função particao para dividir o vetor em duas partes, ordenando recursivamente cada parte.

**bogoSort:**



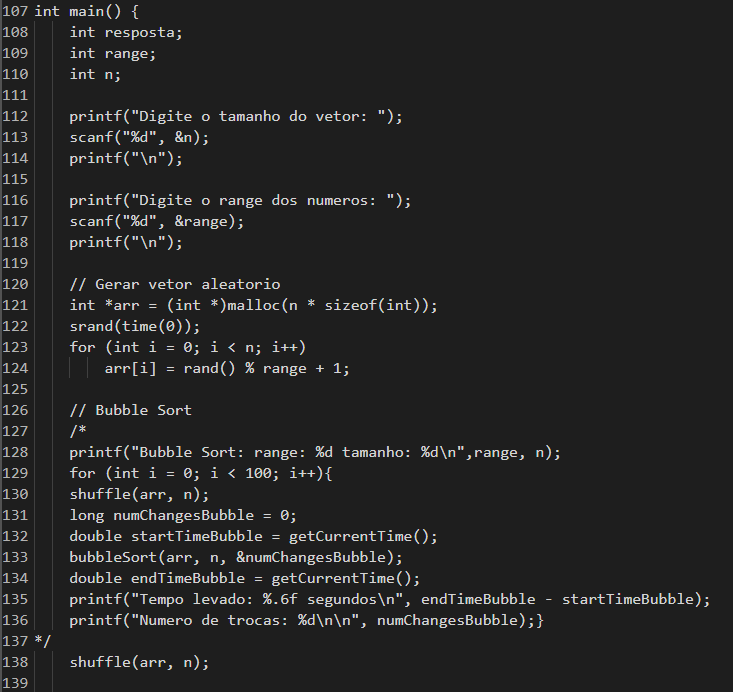
Implementa o algoritmo Bogo Sort. Recebe um vetor (arr), o tamanho do vetor (size), e um ponteiro para o número de trocas (numChanges). Este algoritmo embaralha o vetor repetidamente até que o vetor esteja ordenado.

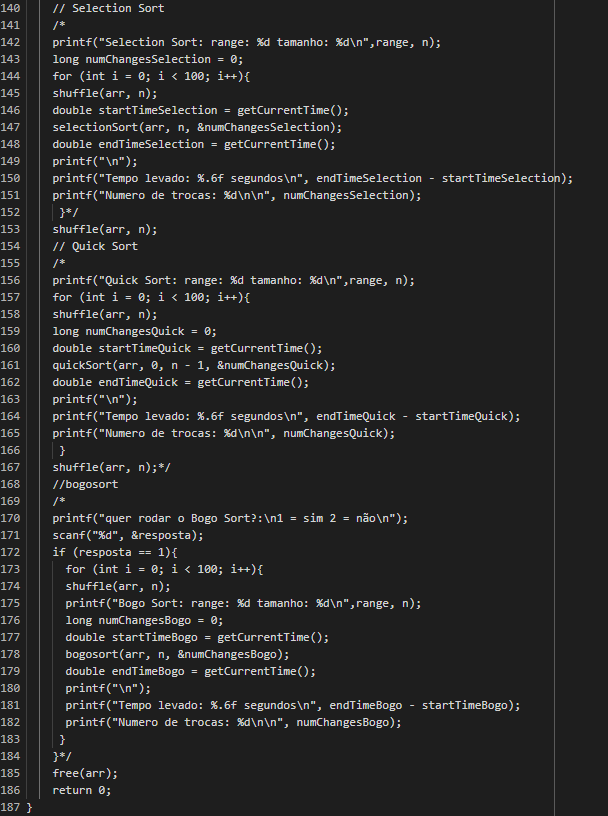
**getCurrentTime:**



Retorna o tempo atual em segundos, utilizando a função clock().

**main:**

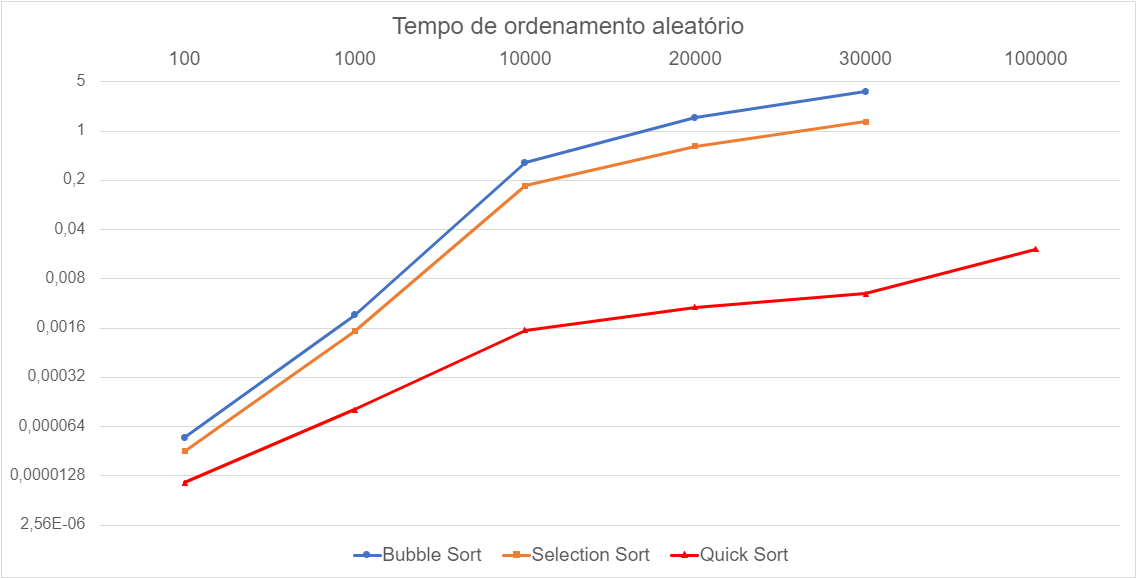




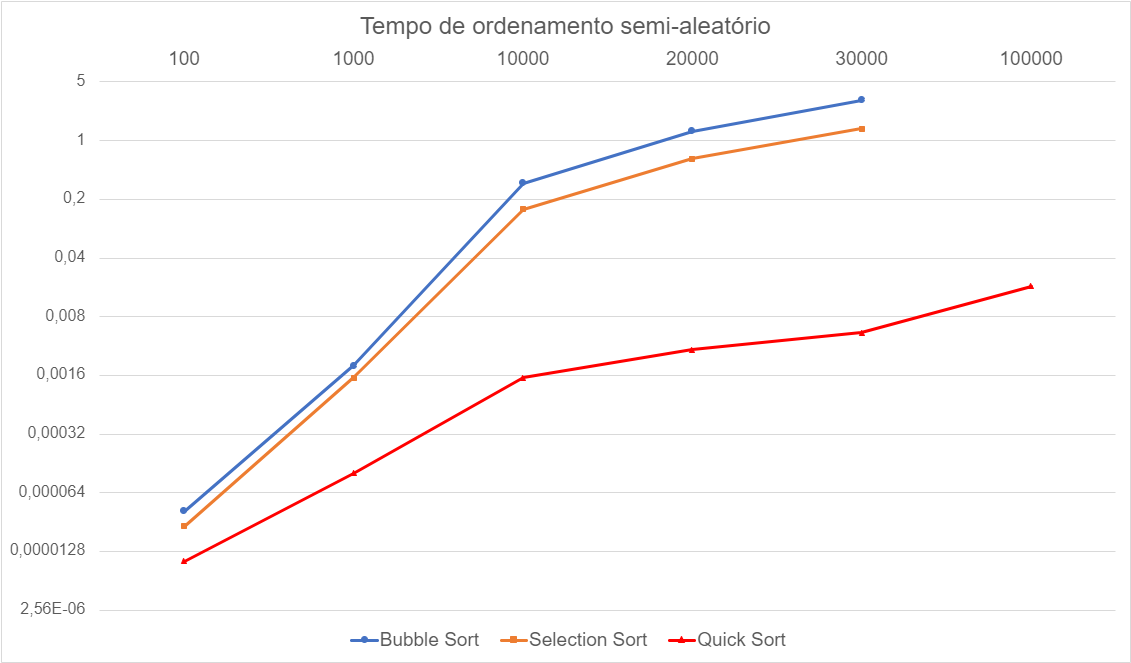
Função principal do programa. Onde entramos com o tamanho do vetor e o range dos números, gera um vetor aleatório, e em seguida, executa os algoritmos de ordenação (Bubble Sort, Selection Sort, Quick Sort, e Bogo Sort) várias vezes, medindo o tempo de execução e o número de trocas em cada execução.

# **Resultados e Discussão**

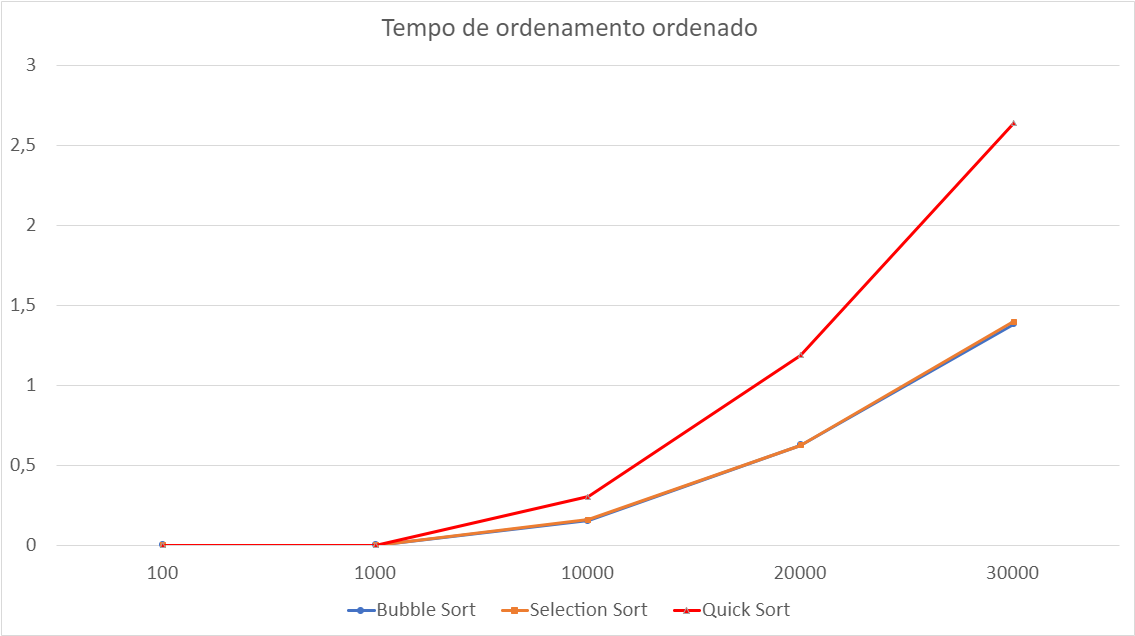
Abaixo estão os gráficos comparando as médias de cem testes realizados medindo o tempo para ordenar um vetor de tamanho n e range de números (n\*10), semi-ordenado e ordenado:



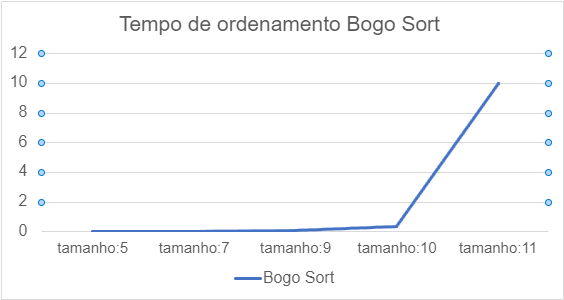
Tempo de ordenamento aleatório



Tempo de ordenamento semi-ordenado

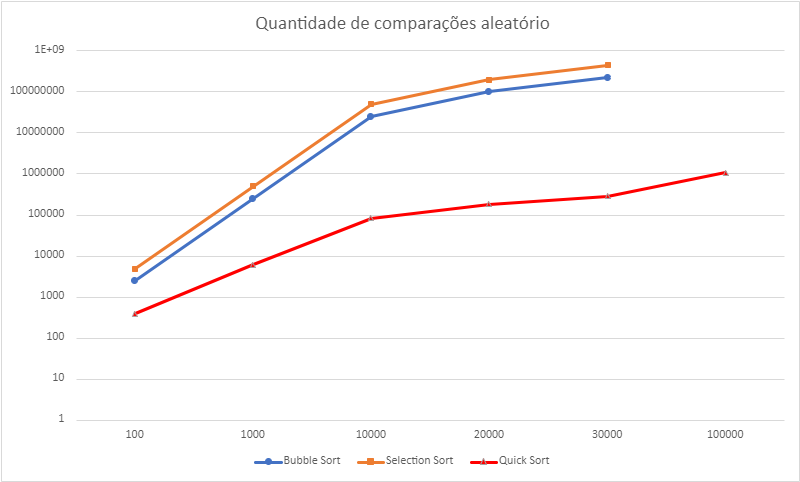


Tempo de ordenamento ordenado

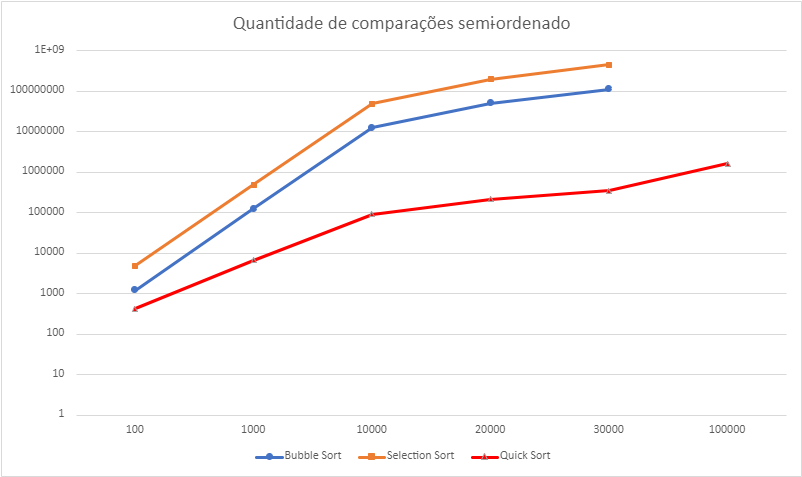


Tempo de ordenamento Bogo Sort

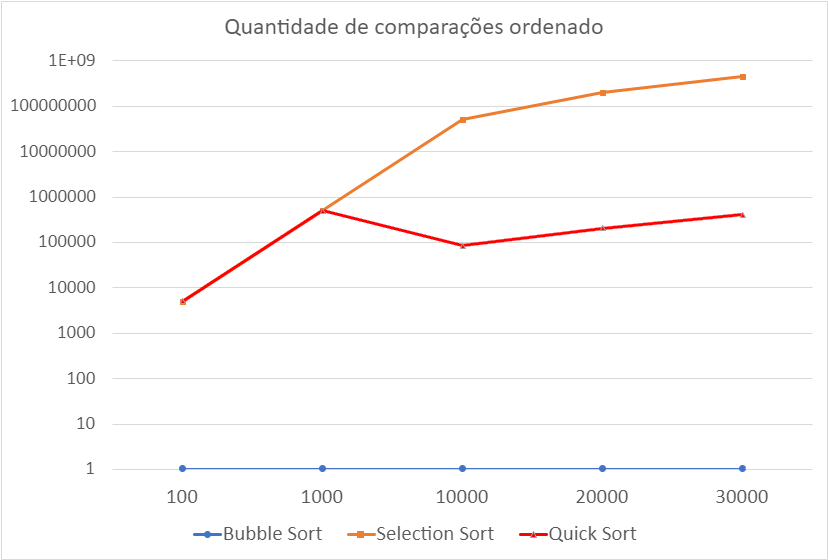
Abaixo estão os gráficos comparando as médias de cem testes realizados medindo a quantidade de comparações necessárias para ordenar um vetor de tamanho n e range de números (n\*10), semi-ordenado e ordenado:



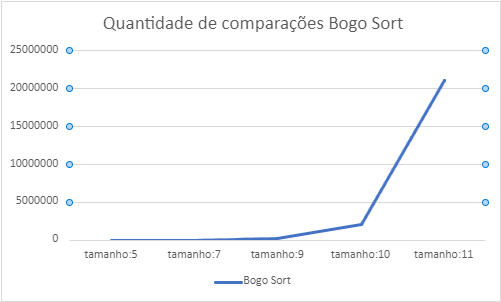
Quantidade de comparações aleatório



Quantidade de comparações semi-ordenado



Quantidade de comparações ordenado



Quantidade de comparações Bogo Sort:

Nesse experimento descobrimos as peculiaridades dos algoritmos de ordenação como o fato de que o Selection Sort sempre faz a mesma quantidade de comparações para ordenar um vetor de tamanho n, ou que a performance do Quick Sort despenca quando lhe é dado um vetor ordenado.

Em conclusão: O experimento deu resultados parecidos com o que era esperado, o Quick Sort é vastamente superior em questão de tempo e quantidade de comparações necessárias com vetores aleatórios e semi-ordenados mas seu pior caso é infelizmente bem comum sendo vetores ordenados, neste caso sua performance é comparável aos demais (exceto o Bogo Sort).

Apesar disso o Quick Sort continua sendo o melhor algoritmo de ordenação do experimento.

# **Considerações finais**

O projeto foi bem interessante, foi aprofundando uma parte da matéria ensinada em sala de aula, e dessa vez foi executado em um programa para obtenção de resultados e comparações. Uma etapa legal das pesquisas, foi descobrir que em meio a tantas opções de ordenações existe uma que seja de fato ineficiente, diferente de algumas outras que nos foi apresentada, o bogo sort consegue ser único, obviamente que escolhemos ele com intuito de servir como exemplo para demonstrar um resultado ineficiente e comparar com um pior resultado ou um método eficiente. Seguindo essa ordem nos surpreendemos com a ineficiência do bubble sort, tivemos alguns exemplos de como ele responde a pequenas e grandes quantidades de dados, mas ao pôr em prática e medir com outros métodos fica provado que por mais simples e fácil que ele indique ser, não vai ser o melhor e mais prático sem todos os casos. Referente aos outros métodos não temos muitos comentários, com exceção do Quick sort, que nos fez mudar o tipo de gráfico que iriamos usar dado a sua eficiência, em comparação com os outros ficaria somente uma linha reta, com os métodos demonstrados nessa pesquisa, o quick sort é o melhor em todas as situações exceto em casos de vetor já ordenado.

# **Referências bibliográficas**

<https://colab.research.google.com/>

<https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao>

<https://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o>

<https://pt.wikipedia.org/wiki/Bubble_sort>

<https://pt.wikipedia.org/wiki/Selection_sort>

<https://pt.wikipedia.org/wiki/Quicksort>

<https://pt.wikipedia.org/wiki/Bogosort>

# **Código Fonte**

%%writefile Sauce.c

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

int particao(int arr[], int low, int high, int\* numChanges) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high - 1; j++) {

if (arr[j] <= pivot) {

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

(\*numChanges)++;

}

}

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

(\*numChanges)++;

return i + 1;

}

// Função de ordenação Bubble Sort

void bubbleSort(int arr[], int n, int\* numChanges) {

shuffle(arr, n);

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

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

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

int temp = arr[j];

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

arr[j + 1] = temp;

(\*numChanges)++;

}

}

}

}

// Função de ordenação Selection Sort

void selectionSort(int arr[], int n, int\* numChanges) {

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

int minIndex = i;

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

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

minIndex = j;

(\*numChanges)++;

}

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

// Função de ordenação Quick Sort

void quickSort(int arr[], int low, int high, int\* numChanges) {

if (low < high) {

int pivot = particao(arr, low, high, numChanges);

quickSort(arr, low, pivot - 1, numChanges);

quickSort(arr, pivot + 1, high, numChanges);

}

}

// Função para verificar se a lista está ordenada

bool is\_sorted(int arr[], int size) {

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

if (arr[i] > arr[i + 1]) {

return false;

}

}

return true;

}

// Função para embaralhar a lista

void shuffle(int arr[], int size) {

for (int i = 0; i < size; i++) {

int j = rand() % size;

// Troca arr[i] e arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Função de ordenação Bogo Sort

void bogosort(int arr[], int size, int \*numChanges) {

shuffle(arr, size);

while (!is\_sorted(arr, size)) {

shuffle(arr, size);

(\*numChanges)++;

}

}

// Get current time in seconds

double getCurrentTime() {

return (double)clock() / CLOCKS\_PER\_SEC;

}

int main() {

int resposta = 1;

int range;

printf("Enter the range of the vector: ");

scanf("%d", &range);

printf("\n");

int n;

printf("Enter the size of the vector: ");

scanf("%d", &n);

printf("\n");

// Generate random vector

int \*arr = (int \*)malloc(n \* sizeof(int));

srand(time (0));

for (int i = 0; i < n; i++)

arr[i] = rand() % range + 1;

// Bubble Sort

printf("Bubble Sort:\n");

long numChangesBubble = 0;

double startTimeBubble = getCurrentTime();

bubbleSort(arr, n, &numChangesBubble);

double endTimeBubble = getCurrentTime();

printf("\n");

printf("Time taken: %.6f seconds\n", endTimeBubble - startTimeBubble);

printf("Number of changes: %d\n\n", numChangesBubble);

// Selection Sort

printf("Selection Sort:\n");

long numChangesSelection = 0;

double startTimeSelection = getCurrentTime();

selectionSort(arr, n, &numChangesSelection);

double endTimeSelection = getCurrentTime();

printf("\n");

printf("Time taken: %.6f seconds\n", endTimeSelection - startTimeSelection);

printf("Number of changes: %d\n\n", numChangesSelection);

// Quick Sort

printf("Quick Sort:\n");

long numChangesQuick = 0;

double startTimeQuick = getCurrentTime();

quickSort(arr, 0, n - 1, &numChangesQuick);

double endTimeQuick = getCurrentTime();

printf("\n");

printf("Time taken: %.6f seconds\n", endTimeQuick - startTimeQuick);

printf("Number of changes: %d\n\n", numChangesQuick);

//bogosort

/\*

printf("quer rodar o Bogo Sort?:\n1 = sim 2 = não\n");

scanf("%d", &resposta);

if (resposta == 1) {

printf("Bogo Sort:\n");

long numChangesBogo = 0;

double startTimeBogo = getCurrentTime();

bogosort(arr, n, &numChangesBogo);

double endTimeBogo = getCurrentTime();

printf("\n");

printf("Time taken: %.6f seconds\n", endTimeBogo - startTimeBogo);

printf("Number of changes: %d\n\n", numChangesBogo);

}

\*/

free(arr);

return 0;

}

%%shell

gcc Sauce.c -w -o programa

./programa