INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO

CAMPUS GUARULHOS

IGOR BANDEIRA DE SOUZA - GU3026744
JADSON HENRIQUE DE LIMA - GU3026761
JASMINE PINHEIRO DE SOUZA - GU302704X
KETLYN MORAES CAMPOS - GU3027678
LUCAS BARBOSA CALIXTO - GU3015041

PROJETO ORDENAÇÃO

GUARULHOS - SP
2023

IGOR BANDEIRA DE SOUZA - GU3026744
JADSON HENRIQUE DE LIMA - GU3026761
JASMINE PINHEIRO DE SOUZA - GU302704X
KETLYN MORAES CAMPOS - GU3027678
LUCAS BARBOSA CALIXTO - GU3015041

PROJETO ORDENAÇÃO

Trabalho apresentado à disciplina de Estrutura de Dados II do Instituto Federal de Educação, Ciência e Tecnologia Campus Guarulhos como nota parcial para aprovação na disciplina do curso Análise e Desenvolvimento de Sistemas, terceiro semestre.

Professor: Antonio Angelo De Souza Tartaglia.

GUARULHOS - SP
2023

SUMÁRIO

1. INTRODUÇÃO

Este trabalho foi criado a partir de uma pesquisa anterior, sobre os algoritmos de ordenação, que foi separada em grupos em sala de aula, cada grupo havia ficado com um algoritmo de ordenação, e este grupo em específico ficou com o Heap Sort.

Um Algoritmo de ordenação, na programação, é um método que organiza os elementos de uma determinada sequência em uma ordem específica. O propósito da organização é simplificar a recuperação dos dados de um array, e reduzir a complexidade de algum problema. Além disso, existem uma variedade de algoritmos, cada um representa um tipo de eficiência e limitações quando se trata de memória e tempo de execução.

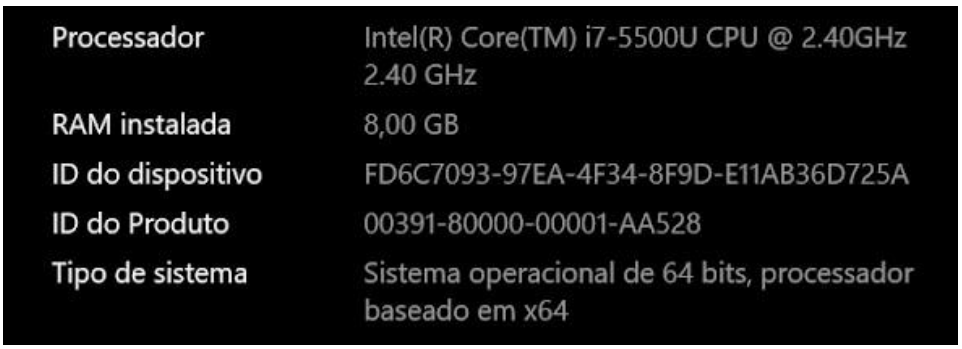
Logo, como já foi citado acima os conceitos, a finalidade do presente relatório é descrever quais foram os procedimentos usados e as dificuldades encontradas para a realização do projeto de ordenação, na qual possui o intento de apresentar o desempenho sobre os nove algoritmos Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Merge Sort, Quick Sort, Heap Sort, Radix Sort (somente LSD) e o Tim Sort. De acordo com os conhecimentos adquiridos ao decorrer das aulas, com ênfase em fazer uma comparação no tempo de execução entre os mesmos.

2. MATERIAIS E MÉTODOS

2.1 Especificações técnicas da primeira máquina utilizada

A figura acima trata-se das configurações do equipamento utilizado para realização do projeto. O processador utilizado foi um AMD Ryzen 5 3500U com o desempenho de 2.10 GHz, a memória ram é de 12,00 GB e com SSD de um 1 TB, seu sistema operacional é o Windows 10 de 64 bits. Ademais o fabricante é a Lenovo

2.2 Especificações técnicas da segunda máquina utilizada



A figura acima trata-se das configurações do equipamento utilizado para realização do projeto. O processador utilizado foi um Intel ® Core(™) i7-5500U com o desempenho de 2.40 GHz, a memória ram é de 8,00 GB, e com HD de 220 GB, seu sistema operacional é o Windows 10 de 64 bits. Ademais o fabricante é a Lenovo.

2.3 Métodos

Devido às múltiplas ocupações dos integrantes deste grupo, incluindo trabalho, estágio e outros projetos, foi preciso encontrar uma forma de reunir todos para produzir o trabalho. A melhor solução encontrada foi realizar videochamadas.

O primeiro encontro virtual foi crucial para o andamento do trabalho. Nele, foi feito uma leitura geral do projeto e dividido as tarefas em várias etapas para garantir que cada membro pudesse contribuir da melhor forma possível.

As fases do projeto foram divididas da seguinte forma:

Fase 1: Adequação dos algoritmos para funcionar com a função srand;

Fase 2: Criação do esqueleto principal do programa e separação em Tipo Abstrato de Dados;

Fase 3: Criação de um novo switch case com os números de elementos desejados; Fase 4: Criação do vetor dinâmico e automação das “baterias de testes”;

Fase 5: Adequação do programa para a função gettimeofday, e criação do vetor com comportamento natural e do vetor pior caso.

Durante as fases foi percebido que em alguns momentos não era preciso seguir o cronograma à risca, a função gettimeofday foi adequada logo após o fim da Fase 3.

3. DESCRIÇÃO DAS ATIVIDADES

Utilizou-se como material de apoio os três algoritmos (Bubble Sort, Insertion Sort, Selection Sort) fornecidos nas aulas, e os outros seis (Shell Sort, Merge Sort, Quick Sort, Heap Sort, Radix Sort (somente LSD) e o Tim Sort) apresentados pelos grupos no seminário sobre algoritmos de ordenação que estão disponíveis na plataforma Moodle, conforme os requisitos.

Primeiramente, foram realizados vários testes a fim de certificar que todos os algoritmos tinham sido criados corretamente. Além disso, foram inseridos juntos ao switch case, para que o usuário consiga escolher qual tipo de algoritmo deseja para realizar a ordenação dos elementos e também possa escolher a quantidade de elementos que deseja ordenar, sendo seis tipos de opções: 1.000, 5.000, 10.000, 20.000, 50.000 ou 100.000 elementos.

Para proceder, houve a implementação do modelo Tipo Abstrato de Dados (TAD), foram criados três tipos de arquivos, o main, projeto.c e o projeto.h, cada um com a sua finalidade. O main é o arquivo principal, o projeto.h com os protótipos das funções e o tipo de dado armazenado nas ordenações, já no projeto.c encontra-se às implementações de todas as funções usadas no programa.

Além disso, foi de extrema importância respeitar uma sequência de passos para não acarretar em problemas futuros como por exemplo, o ponteiro direcionar a um endereço que não existe, ou se alguma função estiver de divergência com o protótipo da mesma. No entanto, quando foram realizados os testes com todas as quantidades de elementos em cada método de ordenação, infelizmente foi obtido um problema com as funções do algoritmo Tim Sort, lembrando que trata-se de um algoritmo de ordenação híbrido, que combina os conceitos do Insertion Sort e do Merge Sort. Ele primeiro divide a lista em blocos de tamanhos variáveis e, em seguida, aplica o Insertion Sort para ordenar cada um desses blocos individualmente. Em seguida, os blocos são mesclados utilizando uma abordagem semelhante ao Merge Sort, porém uma certa otimização.

Sendo assim, para resolver o seguinte impasse, foi modificada a quantidade de runs que teria no define, a princípio foi atribuída com 25000 para resolvê-lo, porém a raiz do problema, encontrava-se na função Merge Sort, após ler diversos

artigos, fóruns e comunidades de programação, como o Stack Overflow, em busca de soluções, foi analisado que a função Merge Sort, tinha sido feita de maneira errônea, estava faltando partes do algoritmo, por fim ao identificá-lo, foi possível prosseguir com o restante do projeto.

Após a resolução dos problemas relacionados ao Tim Sort, procedeu-se para resolver a questão das “baterias de testes”. Para cada algoritmo, o programa cria um novo vetor dinâmico, com uma função malloc para a alocação de memória. Este vetor será preenchido com elementos aleatórios gerados de acordo com a quantidade escolhida no menu anterior. O processo se repete 10 vezes para cada quantidade escolhida, reinicializando a alocação e, dessa forma, preenchendo o novo vetor com elementos diferentes do anterior.

A função utilizada na geração dos elementos é a rand(), com valores entre 0 e 100. O principal problema foi justamente desenvolver uma função que sempre gerasse valores diferentes para cada vetor, pois sempre que um novo vetor era gerado, ele tinha exatamente a mesma sequência de valores do anterior. Logo foi percebido um problema: os dois loops “for” estavam sendo inicializados com a mesma variável, no caso, a variável “i”. Após a alteração da variável do segundo loop para “j”, a função passou a retornar três arrays diferentes. A segunda solução foi alterar o parâmetro da função srand() para: srand(time(NULL) + j). Dessa forma, além da função se basear no tempo, também é considerado o índice atual do loop “for” para garantir valores aleatórios, mesmo que o array seja preenchido várias vezes dentro de um curto espaço de tempo.

Posteriormente, foi notado que o algoritmo Radix Sort não estava funcionando com 50 mil e nem com 100 mil elementos. Este problema existia desde o começo do projeto, mas pelo fato de não ter sido testado, não foi percebido. Basicamente, o programa parava antes de ordenar os elementos do vetor, ou seja, não cumpria com seu papel principal. Após diversas análises e testes, chegou-se a seguinte solução: no lugar das já conhecidas “buckets”, que costumam ser utilizadas para coletar os valores no Radix, usou-se um vetor “count” para contar quantos algarismos possuía cada dígito em sua respectiva posição. Em seguida, o mesmo vetor “count” foi usado para determinar as posições corretas dos números no momento da saída. Esse processo é repetido em cada posição do número até

que o vetor seja completamente ordenado. Com esta solução, o Radix Sort passou a funcionar com 50 mil e 100 mil elementos.

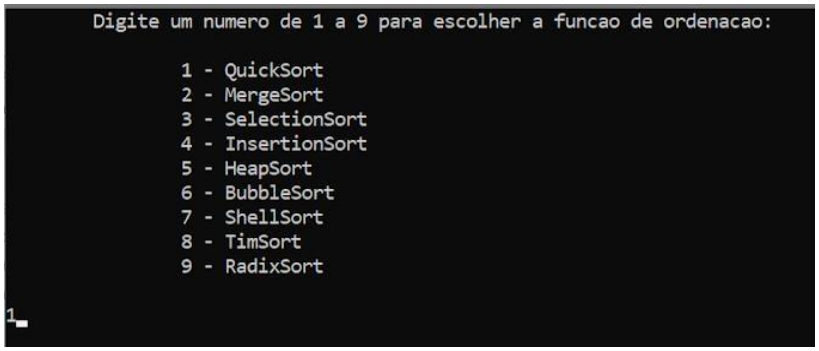
Finalizadas as medições dos 10 vetores de números aleatórios, o programa realiza mais duas medições: o tempo gasto quando a entrada é um vetor já ordenado de forma crescente (melhor caso) e quando a entrada é um vetor decrescente (pior caso). Nesta etapa, utiliza-se o último vetor gerado tanto na forma crescente quanto na forma decrescente. Isto é apenas para efeitos de comparação entre o melhor e o pior caso. Em contrapartida, a maior dificuldade nesta etapa foi estruturar o código para que as variáveis não dessem como indefinidas, ou seja, extrair o maior número de variáveis e colocar fora do escopo do switch case. Ademais as variáveis que não poderiam está fora do escopo, foi necessário renomeá-las, para garantir o funcionamento de todos os algoritmos.

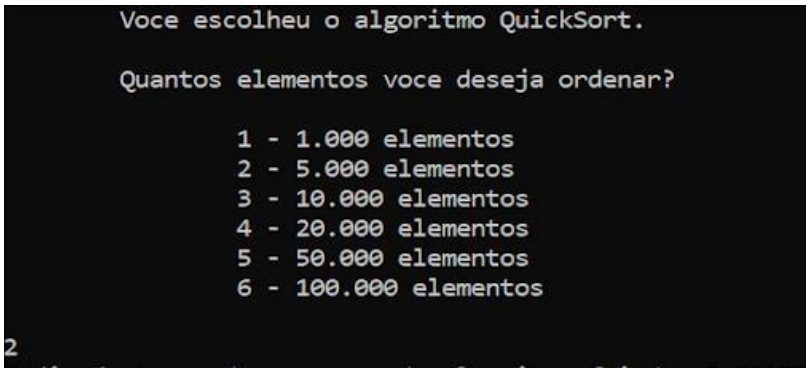
Portanto, após certificar que todas as etapas do projeto estavam concluídas, foi feita uma revisão minuciosa, verificando-se que todos os requisitos tinham sido atendidos, para evitar quaisquer erros ou inconsistência.

4. ANÁLISE DE DESEMPENHO DOS ALGORITMOS DE ORDENAÇÃO

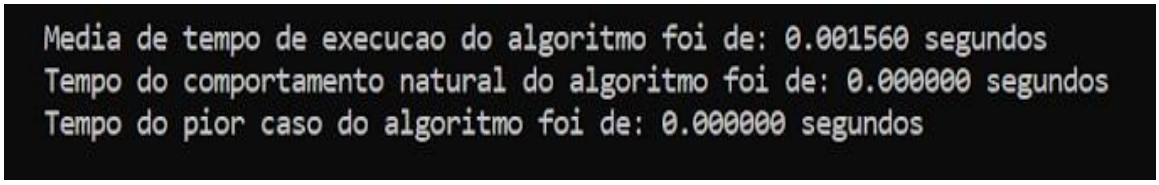
4.1 Execução do Programa

As figuras a seguir, demonstram como é a execução do programa:





A simulação acima tem início com um menu de seleção dos algoritmos de ordenação, o valor de exemplo neste caso e para seleção do Quick Sort no menu é o número 01. Após a escolha no menu, é informado um segundo menu, que tem por objetivo apresentar os valores que podem ser informados na simulação aleatória de algoritmos, nesta etapa foi escolhido a opção 02.



A figura acima mostra o resultado final do algoritmo (Quick Sort) que foi feito a simulação. Sendo assim, foi repetido esse mesmo procedimento para cada um dos algoritmos, e suas respectivas quantidades, e foram analisadas apenas as médias.

4.2 Bubble Sort:

- 1 mil elementos - Máquina 1 - Tempo Médio: 0.003126 segundos
- 1 mil elementos - Máquina 2 - Tempo Médio: 0.007812 segundos

- 5 mil elementos - Máquina 1 - Tempo Médio: 0.072581 segundos
- 5 mil elementos - Máquina 2 - Tempo Médio: 0.221065 segundos

- 10 mil elementos - Máquina 1 - Tempo Médio: 0.294495 segundos
- 10 mil elementos - Máquina 2 - Tempo Médio: 1.057912 segundos

- 20 mil elementos - Máquina 1 - Tempo Médio: 1.239335 segundos
- 20 mil elementos - Máquina 2 - Tempo Médio: 5.596992 segundos

- 50 mil elementos - Máquina 1 - Tempo Médio: 8.204178 segundos
- 50 mil elementos - Máquina 2 - Tempo Médio: 35.178744 segundos

- 100 mil elementos - Máquina 1 - Tempo Médio: 36.932495 segundos
- 100 mil elementos - Máquina 2 - Tempo Médio: 145.051247 segundos

4.3 Insertion Sort:

- 1 mil elementos - Máquina 1 - Tempo Médio: 0.001564 segundos
- 1 mil elementos - Máquina 2 - Tempo Médio: 0.003115 segundos

- 5 mil elementos - Máquina 1 - Tempo Médio: 0.017975 segundos
- 5 mil elementos - Máquina 2 - Tempo Médio: 0.070308 segundos

- 10 mil elementos - Máquina 1 - Tempo Médio: 0.070384 segundos
- 10 mil elementos - Máquina 2 - Tempo Médio: 0.335268 segundos

- 20 mil elementos - Máquina 1 - Tempo Médio: 0.282582 segundos
- 20 mil elementos - Máquina 2 - Tempo Médio: 1.224387 segundos

- 50 mil elementos - Máquina 1 - Tempo Médio: 1.734698 segundos
- 50 mil elementos - Máquina 2 - Tempo Médio: 8.365254 segundos

- 100 mil elementos - Máquina 1 - Tempo Médio: 7.080685 segundos
- 100 mil elementos - Máquina 2 - Tempo Médio: 32.968995 segundos

4.4 Selection Sort:

- 1 mil elementos - Máquina 1 - Tempo Médio: 0.001582 segundos
- 1 mil elementos - Máquina 2 - Tempo Médio: 0.006239 segundos

- 5 mil elementos - Máquina 1 - Tempo Médio: 0.027625 segundos
- 5 mil elementos - Máquina 2 - Tempo Médio: 0.132793 segundos

- 10 mil elementos - Máquina 1 - Tempo Médio: 0.114270 segundos
- 10 mil elementos - Máquina 2 - Tempo Médio: 0.539623 segundos

- 20 mil elementos - Máquina 1 - Tempo Médio: 0.443638 segundos
- 20 mil elementos - Máquina 2 - Tempo Médio: 2.256088 segundos

- 50 mil elementos - Máquina 1 - Tempo Médio: 3.213878 segundos
- 50 mil elementos - Máquina 2 - Tempo Médio: 13.458308 segundos

- 100 mil elementos - Máquina 1 - Tempo Médio: 12.005516 segundos
- 100 mil elementos - Máquina 2 - Tempo Médio: 59.056014 segundos

4.5 Shell Sort:

- 1 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 1 mil elementos - Máquina 2 - Tempo Médio: 0.000000 segundos

- 5 mil elementos - Máquina 1 - Tempo Médio: 0.001412 segundos
- 5 mil elementos - Máquina 2 - Tempo Médio: 0.004682 segundos

- 10 mil elementos - Máquina 1 - Tempo Médio: 0.001563 segundos
- 10 mil elementos - Máquina 2 - Tempo Médio: 0.007812 segundos

- 20 mil elementos - Máquina 1 - Tempo Médio: 0.002866 segundos
- 20 mil elementos - Máquina 2 - Tempo Médio: 0.012490 segundos

- 50 mil elementos - Máquina 1 - Tempo Médio: 0.008466 segundos
- 50 mil elementos - Máquina 2 - Tempo Médio: 0.035266 segundos

- 100 mil elementos - Máquina 1 - Tempo Médio: 0.016800 segundos
- 100 mil elementos - Máquina 2 - Tempo Médio: 0.067422 segundos

4.6 Merge Sort:

- 1 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 1 mil elementos - Máquina 2 - Tempo Médio: 0.001552 segundos

- 5 mil elementos - Máquina 1 - Tempo Médio: 0.000992 segundos
- 5 mil elementos - Máquina 2 - Tempo Médio: 0.003115 segundos

- 10 mil elementos - Máquina 1 - Tempo Médio: 0.001511 segundos
- 10 mil elementos - Máquina 2 - Tempo Médio: 0.006239 segundos

- 20 mil elementos - Máquina 1 - Tempo Médio: 0.003095 segundos
- 20 mil elementos - Máquina 2 - Tempo Médio: 0.009364 segundos

- 50 mil elementos - Máquina 1 - Tempo Médio: 0.006660 segundos
- 50 mil elementos - Máquina 2 - Tempo Médio: 0.031239 segundos

- 100 mil elementos - Máquina 1 - Tempo Médio: 0.016895 segundos
- 100 mil elementos - Máquina 2 - Tempo Médio: 0.062502 segundos

4.7 Quick Sort:

- 1 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 1 mil elementos - Máquina 2 - Tempo Médio: 0.000000 segundos

- 5 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 5 mil elementos - Máquina 2 - Tempo Médio: 0.001556 segundos

- 10 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 10 mil elementos - Máquina 2 - Tempo Médio: 0.003115 segundos

- 20 mil elementos - Máquina 1 - Tempo Médio: 0.002168 segundos
- 20 mil elementos - Máquina 2 - Tempo Médio: 0.006246 segundos

- 50 mil elementos - Máquina 1 - Tempo Médio: 0.003730 segundos
- 50 mil elementos - Máquina 2 - Tempo Médio: 0.015969 segundos

- 100 mil elementos - Máquina 1 - Tempo Médio: 0.008464 segundos
- 100 mil elementos - Máquina 2 - Tempo Médio: 0.033370 segundos

4.8 Heap Sort:

- 1 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 1 mil elementos - Máquina 2 - Tempo Médio: 0.000000 segundos

- 5 mil elementos - Máquina 1 - Tempo Médio: 0.001518 segundos
- 5 mil elementos - Máquina 2 - Tempo Médio: 0.004679 segundos

- 10 mil elementos - Máquina 1 - Tempo Médio: 0.003081 segundos
- 10 mil elementos - Máquina 2 - Tempo Médio: 0.007809 segundos

- 20 mil elementos - Máquina 1 - Tempo Médio: 0.004039 segundos
- 20 mil elementos - Máquina 2 - Tempo Médio: 0.015611 segundos

- 50 mil elementos - Máquina 1 - Tempo Médio: 0.011489 segundos
- 50 mil elementos - Máquina 2 - Tempo Médio: 0.045671 segundos

- 100 mil elementos - Máquina 1 - Tempo Médio: 0.021938 segundos
- 100 mil elementos - Máquina 2 - Tempo Médio: 0.092369 segundos

4.9 Radix Sort (somente LSD):

- 1 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 1 mil elementos - Máquina 2 - Tempo Médio: 0.000000 segundos

- 5 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 5 mil elementos - Máquina 2 - Tempo Médio: 0.000000 segundos

- 10 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 10 mil elementos - Máquina 2 - Tempo Médio: 0.001556 segundos

- 20 mil elementos - Máquina 1 - Tempo Médio: 0.001565 segundos
- 20 mil elementos - Máquina 2 - Tempo Médio: 0.003115 segundos

- 50 mil elementos - Máquina 1 - Tempo Médio: 0.003852 segundos
- 50 mil elementos - Máquina 2 - Tempo Médio: 0.009377 segundos

- 100 mil elementos - Máquina 1 - Tempo Médio: 0.003877 segundos
- 100 mil elementos - Máquina 2 - Tempo Médio: 0.018745 segundos

4.10 Tim Sort:

- 1 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 1 mil elementos - Máquina 2 - Tempo Médio: 0.000000 segundos

- 5 mil elementos - Máquina 1 - Tempo Médio: 0.000000 segundos
- 5 mil elementos - Máquina 2 - Tempo Médio: 0.003112 segundos

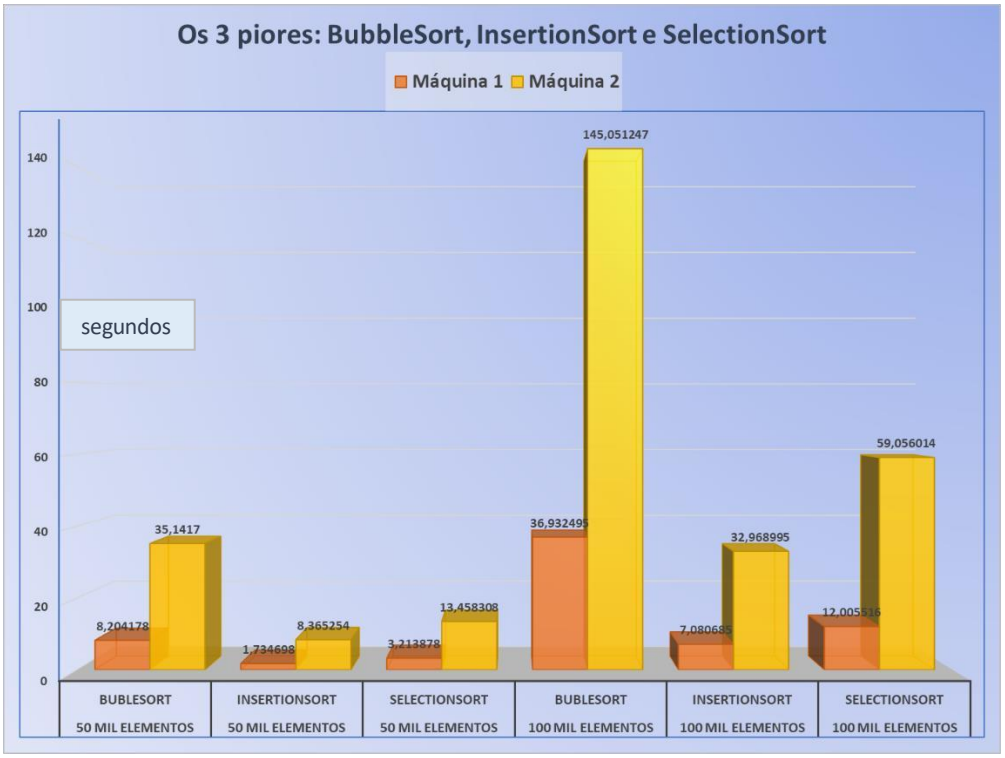
- 10 mil elementos - Máquina 1 - Tempo Médio: 0.001565 segundos
- 10 mil elementos - Máquina 2 - Tempo Médio: 0.004672 segundos

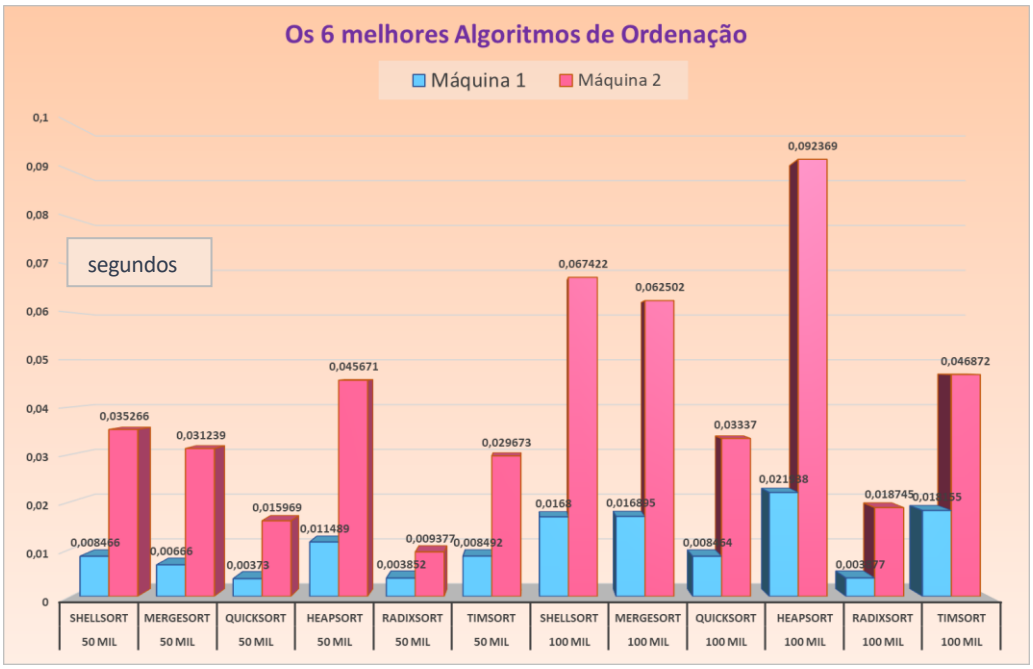
- 20 mil elementos - Máquina 1 - Tempo Médio: 0.003080 segundos
- 20 mil elementos - Máquina 2 - Tempo Médio: 0.010927 segundos

- 50 mil elementos - Máquina 1 - Tempo Médio: 0.008492 segundos
- 50 mil elementos - Máquina 2 - Tempo Médio: 0.029673 segundos

- 100 mil elementos - Máquina 1 - Tempo Médio: 0.018155 segundos
- 100 mil elementos - Máquina 2 - Tempo Médio: 0.046872 segundos

5. COMPARAÇÃO DE DESEMPENHO ENTRE OS ALGORITMOS DE ORDENAÇÃO





6. CÓDIGOS

Neste capítulo será apresentado o código realizado e seus respectivos aspectos, objetivos e funções desempenhadas.

6.1 main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include "projeto.h"

int main() {
    int teste;
    do {
    struct timeval tempo_inicial, tempo_final;
    double deltaTempo;
    int tamanhoVetor;

        system("cls");
        printf("\tDigite um numero de 1 a 9 para escolher a funcao de ordenacao:\n\n");
        printf("\t\t1 - QuickSort\n");
        printf("\t\t2 - MergeSort\n");
        printf("\t\t3 - SelectionSort\n");
        printf("\t\t4 - InsertionSort\n");
        printf("\t\t5 - HeapSort\n");
        printf("\t\t6 - BubbleSort\n");
        printf("\t\t7 - ShellSort\n");
        printf("\t\t8 - TimSort\n");
        printf("\t\t9 - RadixSort\n\n");

        int opcaoAlgoritmo;
        scanf("%d", &opcaoAlgoritmo);

        switch (opcaoAlgoritmo) {
```

```
case 1:
    system("cls"); // ou "clear" se estiver usando Linux ou MacOS
    printf("\tVoce escolheu o algoritmo QuickSort.\n\n");
    break;
case 2:
    system("cls"); // ou "clear" se estiver usando Linux ou MacOS
    printf("\tVoce escolheu o algoritmo MergeSort.\n\n");
    break;
case 3:
    system("cls"); // ou "clear" se estiver usando Linux ou MacOS
    printf("\tVoce escolheu o algoritmo SelectionSort.\n\n");
    break;
case 4:
    system("cls"); // ou "clear" se estiver usando Linux ou MacOS
    printf("\tVoce escolheu o algoritmo InsertionSort.\n\n");
    break;
case 5:
    system("cls"); // ou "clear" se estiver usando Linux ou MacOS
    printf("\tVoce escolheu o algoritmo HeapSort.\n\n");
    break;
case 6:
    system("cls"); // ou "clear" se estiver usando Linux ou MacOS
    printf("\tVoce escolheu o algoritmo BubbleSort.\n");
    break;
case 7:
    system("cls"); // ou "clear" se estiver usando Linux ou MacOS
    printf("\tVoce escolheu o algoritmo ShellSort.\n\n");
    break;
case 8:
    system("cls"); // ou "clear" se estiver usando Linux ou MacOS
    printf("\tVoce escolheu o algoritmo TimSort.\n\n");
    break;
case 9:
    system("cls"); // ou "clear" se estiver usando Linux ou MacOS
```

```
    printf("\tVoce escolheu o algoritmo RadixSort.\n\n");
    break;
default:
    printf("\tOpcao invalida.\n");
    return 1;
}


printf("\tQuantos elementos voce deseja ordenar?\n\n");
printf("\t\t1 - 1.000 elementos\n");
printf("\t\t2 - 5.000 elementos\n");
printf("\t\t3 - 10.000 elementos\n");
printf("\t\t4 - 20.000 elementos\n");
printf("\t\t5 - 50.000 elementos\n");
printf("\t\t6 - 100.000 elementos\n\n");

int opcaoTamanho;
scanf("%d", &opcaoTamanho);

switch (opcaoTamanho) {
case 1:
    tamanhoVetor = 1000;
    break;
case 2:
    tamanhoVetor = 5000;
    break;
case 3:
    tamanhoVetor = 10000;
    break;
case 4:
    tamanhoVetor = 20000;
    break;
case 5:
    tamanhoVetor = 50000;
```

```
    break;
case 6:
    tamanhoVetor = 100000;
    break;
default:
    printf("Opcao invalida.\n");
    return 1;
}

int vetor[tamanhoVetor];

double somaDeltaTempo = 0.0;
double mediaDeltaTempo;

// Ordena o vetor de acordo com a escolha do usuario
switch (opcaoAlgoritmo) {
case 1:
    // Algoritmo QuickSort
    for (int j = 0; j < 10; j++) {
        int *vetor = (int *)malloc(tamanhoVetor * sizeof(int));

        // Preenche o vetor com valores aleatorios


        srand(time(NULL) + j); // Adiciona um valor diferente  seed do srand
        for (int i = 0; i < tamanhoVetor; i++) {
            vetor[i] = rand() % 100;
        }

        // QuickSort no vetor e contando o tempo
        gettimeofday(&tempo_inicial, NULL);
        quickSort(vetor, 0, tamanhoVetor - 1);
        gettimeofday(&tempo_final, NULL);
```

```
            // Somando os 10 tempos
            double deltaTempo = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
            somaDeltaTempo += deltaTempo;

            free(vetor);
        }
        // Realizando a media dos 10 tempos
        mediaDeltaTempo = somaDeltaTempo / 10.0;
        printf("Media de tempo de execucao do algoritmo foi de: %f segundos \n", mediaDeltaTempo);

        // Tempo do Comportamento Natural
        int *vetorNatural = (int *)malloc(tamanhoVetor * sizeof(int));

        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorNatural[i] = i;
        }

        // Mede o tempo de execucao do algoritmo para o comportamento natural
        gettimeofday(&tempo_inicial, NULL);
        quickSort(vetorNatural, 0, tamanhoVetor - 1);
        gettimeofday(&tempo_final, NULL);

        // Calcula o tempo decorrido
        double deltaTempoNatural = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        printf("Tempo do comportamento natural do algoritmo foi de: %f segundos \n", deltaTempoNatural);

        // Tempo do Pior Caso:
                // Isso significa que os valores no vetor estao em ordem decrescente,
```

```
comecando com "tamanhoVetor" na primeira posicao e terminando com "1" na ultima posicao
        int *vetorPiorCaso = (int *)malloc(tamanhoVetor * sizeof(int));
        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorPiorCaso[i] = tamanhoVetor - i;
        }

        // Mede o tempo de execucao do algoritmo para o pior caso

        gettimeofday(&tempo_inicial, NULL);
        quickSort(vetorPiorCaso, 0, tamanhoVetor - 1);
        gettimeofday(&tempo_final, NULL);

        // Calcula o tempo decorrido
        double deltaTempoPior = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        printf("Tempo do pior caso do algoritmo foi de: %f segundos \n", deltaTempoPior);

        // Libera a memoria alocada para os vetores
        free(vetorNatural);
        free(vetorPiorCaso);

        break;
    case 2:
        // Algoritmo MergeSort
        for (int j = 0; j < 10; j++) {
            int *vetor = (int *)malloc(tamanhoVetor * sizeof(int));

            // Preenche o vetor com valores aleatorios
            srand(time(NULL) + j); // Adiciona um valor diferente  seed do srand
            for (int i = 0; i < tamanhoVetor; i++) {
                vetor[i] = rand() % 100;
```

```
        }

        // MergeSort no vetor e contando o tempo
        gettimeofday(&tempo_inicial, NULL);
        mergeSort(vetor, 0, tamanhoVetor - 1);
        gettimeofday(&tempo_final, NULL);

        // Somando os 10 tempos
        double deltaTempo = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        somaDeltaTempo += deltaTempo;

        free(vetor);
    }
    // Realizando a media dos 10 tempos
    mediaDeltaTempo = somaDeltaTempo / 10.0;
    printf("Media de tempo de execucao do algoritmo foi de: %f segundos \n", mediaDeltaTempo);

    // Tempo do Comportamento Natural
    int *vetorNatural2 = (int *)malloc(tamanhoVetor * sizeof(int));
    for (int i = 0; i <= tamanhoVetor; i++) {
        vetorNatural2[i] = i;
    }


    // Mede o tempo de execucao do algoritmo para o comportamento natural
    gettimeofday(&tempo_inicial, NULL);
    mergeSort(vetorNatural2, 0, tamanhoVetor - 1);
    gettimeofday(&tempo_final, NULL);

    // Calcula o tempo decorrido
    double deltaTempoNatural2 = (tempo_final.tv_sec + tempo_final.tv_usec /
```

```
1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        printf("Tempo do comportamento natural do algoritmo foi de: %f segundos \n", deltaTempoNatural2);

        // Tempo do Pior Caso:
            // Isso significa que os valores no vetor estao em ordem decrescente, comecando com "tamanhoVetor" na primeira posicao e terminando com "1" na ultima posicao
        int *vetorPiorCaso2 = (int *)malloc(tamanhoVetor * sizeof(int));
        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorPiorCaso2[i] = tamanhoVetor - i;
        }

        // Mede o tempo de execucao do algoritmo para o pior caso
        gettimeofday(&tempo_inicial, NULL);
        mergeSort(vetorPiorCaso2, 0, tamanhoVetor - 1);
        gettimeofday(&tempo_final, NULL);


        // Calcula o tempo decorrido
        double deltaTempoPior2 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        printf("Tempo do pior caso do algoritmo foi de: %f segundos \n", deltaTempoPior2);

        // Libera a memoria alocada para os vetores
        free(vetorNatural2);
        free(vetorPiorCaso2);

        break;
    case 3:
        // Algoritmo SelectionSort
```

```
        for (int j = 0; j < 10; j++) {
            int *vetor = (int *)malloc(tamanhoVetor * sizeof(int));

            // Preenche o vetor com valores aleatorios
            srand(time(NULL) + j); // Adiciona um valor diferente  seed do srand
            for (int i = 0; i < tamanhoVetor; i++) {
                vetor[i] = rand() % 100;
            }

            // SelectionSort no vetor e contando o tempo

            gettimeofday(&tempo_inicial, NULL);
            selectionSort(vetor, tamanhoVetor);
            gettimeofday(&tempo_final, NULL);

            // Somando os 10 tempos
            double deltaTempo = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
            somaDeltaTempo += deltaTempo;

            free(vetor);
        }
        // Realizando a media dos 10 tempos
        mediaDeltaTempo = somaDeltaTempo / 10.0;
        printf("Media de tempo de execucao do algoritmo foi de: %f segundos \n", mediaDeltaTempo);

        // Tempo do Comportamento Natural
        int *vetorNatural3 = (int *)malloc(tamanhoVetor * sizeof(int));
        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorNatural3[i] = i;
        }
```

```
// Mede o tempo de execucao do algoritmo para o comportamento natural
gettimeofday(&tempo_inicial, NULL);


selectionSort(vetorNatural3, tamanhoVetor);
gettimeofday(&tempo_final, NULL);

// Calcula o tempo decorrido
double deltaTempoNatural3 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
printf("Tempo do comportamento natural do algoritmo foi de: %f segundos \n", deltaTempoNatural3);

// Tempo do Pior Caso:
// Isso significa que os valores no vetor estao em ordem decrescente, comecando com "tamanhoVetor" na primeira posicao e terminando com "1" na ultima posicao
int *vetorPiorCaso3 = (int *)malloc(tamanhoVetor * sizeof(int));
for (int i = 0; i <= tamanhoVetor; i++) {
   vetorPiorCaso3[i] = tamanhoVetor - i;
}

// Mede o tempo de execucao do algoritmo para o pior caso
gettimeofday(&tempo_inicial, NULL);
selectionSort(vetorPiorCaso3, tamanhoVetor);
gettimeofday(&tempo_final, NULL);

// Calcula o tempo decorrido

double deltaTempoPior3 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
printf("Tempo do pior caso do algoritmo foi de: %f segundos \n",
```

```
deltaTempoPior3);

        // Libera a memoria alocada para os vetores
        free(vetorNatural3);
        free(vetorPiorCaso3);

        break;
    case 4:
        // Algoritmo InsertionSort
        for (int j = 0; j < 10; j++) {
            int *vetor = (int *)malloc(tamanhoVetor * sizeof(int));

            // Preenche o vetor com valores aleatorios
            srand(time(NULL) + j); // Adiciona um valor diferente  seed do srand
            for (int i = 0; i < tamanhoVetor; i++) {
                vetor[i] = rand() % 100;
            }

            // InsertionSort no vetor e contando o tempo
            gettimeofday(&tempo_inicial, NULL);
            insertionSort(vetor, tamanhoVetor);

            gettimeofday(&tempo_final, NULL);

            // Somando os 10 tempos
            double deltaTempo = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
            somaDeltaTempo += deltaTempo;

            free(vetor);
        }
        // Realizando a media dos 10 tempos
        mediaDeltaTempo = somaDeltaTempo / 10.0;
```

```
printf("Media de tempo de execucao do algoritmo foi de: %f segundos \n", mediaDeltaTempo);

// Tempo do Comportamento Natural
int *vetorNatural4 = (int *)malloc(tamanhoVetor * sizeof(int));
for (int i = 0; i <= tamanhoVetor; i++) {
    vetorNatural4[i] = i;
}

// Mede o tempo de execucao do algoritmo para o comportamento natural
gettimeofday(&tempo_inicial, NULL);
insertionSort(vetorNatural4, tamanhoVetor);
gettimeofday(&tempo_final, NULL);

// Calcula o tempo decorrido
double deltaTempoNatural4 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
printf("Tempo do comportamento natural do algoritmo foi de: %f segundos \n", deltaTempoNatural4);

// Tempo do Pior Caso:
// Isso significa que os valores no vetor estao em ordem decrescente, comecando com "tamanhoVetor" na primeira posicao e terminando com "1" na ultima posicao
int *vetorPiorCaso4 = (int *)malloc(tamanhoVetor * sizeof(int));
for (int i = 0; i <= tamanhoVetor; i++) {
    vetorPiorCaso4[i] = tamanhoVetor - i;
}

// Mede o tempo de execucao do algoritmo para o pior caso
gettimeofday(&tempo_inicial, NULL);
insertionSort(vetorPiorCaso4, tamanhoVetor);
gettimeofday(&tempo_final, NULL);
```

```
        // Calcula o tempo decorrido
        double deltaTempoPior4 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos

        printf("Tempo do pior caso do algoritmo foi de: %f segundos \n", deltaTempoPior4);

        // Libera a memoria alocada para os vetores
        free(vetorNatural4);
        free(vetorPiorCaso4);

        break;
    case 5:
        // Algoritmo HeapSort
        for (int j = 0; j < 10; j++) {
            int *vetor = (int *)malloc(tamanhoVetor * sizeof(int));

            // Preenche o vetor com valores aleatorios
            srand(time(NULL) + j); // Adiciona um valor diferente o seed do srand
            for (int i = 0; i < tamanhoVetor; i++) {
                vetor[i] = rand() % 100;
            }

            // HeapSort no vetor e contando o tempo
            gettimeofday(&tempo_inicial, NULL);
            heapSort(vetor, tamanhoVetor);
            gettimeofday(&tempo_final, NULL);

            // Somando os 10 tempos

            double deltaTempo = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o
```

```
calculo para deixar em segundos
            somaDeltaTempo += deltaTempo;

            free(vetor);
        }
        // Realizando a media dos 10 tempos
        mediaDeltaTempo = somaDeltaTempo / 10.0;
        printf("Media de tempo de execucao do algoritmo foi de: %f segundos \n", mediaDeltaTempo);

        // Tempo do Comportamento Natural
        int *vetorNatural5 = (int *)malloc(tamanhoVetor * sizeof(int));
        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorNatural5[i] = i;
        }

        // Mede o tempo de execucao do algoritmo para o comportamento natural
        gettimeofday(&tempo_inicial, NULL);
        heapSort(vetorNatural5, tamanhoVetor);
        gettimeofday(&tempo_final, NULL);

        // Calcula o tempo decorrido


        double deltaTempoNatural5 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        printf("Tempo do comportamento natural do algoritmo foi de: %f segundos \n", deltaTempoNatural5);

        // Tempo do Pior Caso:
                // Isso significa que os valores no vetor estao em ordem decrescente, comecando com "tamanhoVetor" na primeira posicao e terminando com "1" na ultima posicao
```

```
        int *vetorPiorCaso5 = (int *)malloc(tamanhoVetor * sizeof(int));
        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorPiorCaso5[i] = tamanhoVetor - i;
        }

        // Mede o tempo de execucao do algoritmo para o pior caso
        gettimeofday(&tempo_inicial, NULL);
        heapSort(vetorPiorCaso5, tamanhoVetor);
        gettimeofday(&tempo_final, NULL);

        // Calcula o tempo decorrido
        double deltaTempoPior5 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        printf("Tempo do pior caso do algoritmo foi de: %f segundos \n", deltaTempoPior5);

        // Libera a memoria alocada para os vetores
        free(vetorNatural5);
        free(vetorPiorCaso5);

        break;
    case 6:
        // Algoritmo BubbleSort
        for (int j = 0; j < 10; j++) {
            int *vetor = (int *)malloc(tamanhoVetor * sizeof(int));

            // Preenche o vetor com valores aleatorios
            srand(time(NULL) + j); // Adiciona um valor diferente  seed do srand
            for (int i = 0; i < tamanhoVetor; i++) {
                vetor[i] = rand() % 100;
            }

            // BubbleSort no vetor e contando o tempo
```

```
        gettimeofday(&tempo_inicial, NULL);
        bubbleSort(vetor, tamanhoVetor);
        gettimeofday(&tempo_final, NULL);

        // Somando os 10 tempos
        double deltaTempo = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
38

        somaDeltaTempo += deltaTempo;

        free(vetor);
    }
    // Realizando a media dos 10 tempos
    mediaDeltaTempo = somaDeltaTempo / 10.0;
    printf("Media de tempo de execucao do algoritmo foi de: %f segundos \n", mediaDeltaTempo);

    // Tempo do Comportamento Natural
    int *vetorNatural6 = (int *)malloc(tamanhoVetor * sizeof(int));
    for (int i = 0; i <= tamanhoVetor; i++) {
        vetorNatural6[i] = i;
    }

    // Mede o tempo de execucao do algoritmo para o comportamento natural
    gettimeofday(&tempo_inicial, NULL);
    bubbleSort(vetorNatural6, tamanhoVetor);
    gettimeofday(&tempo_final, NULL);

    // Calcula o tempo decorrido
    double deltaTempoNatural6 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
```

39

```
        printf("Tempo do comportamento natural do algoritmo foi de: %f segundos \n", deltaTempoNatural6);

        // Tempo do Pior Caso:
        // Isso significa que os valores no vetor estao em ordem decrescente, comecando com "tamanhoVetor" na primeira posicao e terminando com "1" na ultima posicao
        int *vetorPiorCaso6 = (int *)malloc(tamanhoVetor * sizeof(int));
        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorPiorCaso6[i] = tamanhoVetor - i;
        }

        // Mede o tempo de execucao do algoritmo para o pior caso
        gettimeofday(&tempo_inicial, NULL);
        bubbleSort(vetorPiorCaso6, tamanhoVetor);
        gettimeofday(&tempo_final, NULL);

        // Calcula o tempo decorrido
        double deltaTempoPior6 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        printf("Tempo do pior caso do algoritmo foi de: %f segundos \n", deltaTempoPior6);

        // Libera a memoria alocada para os vetores
        free(vetorNatural6);

        free(vetorPiorCaso6);

        break;
    case 7:
        // Algoritmo ShellSort
```

```
        for (int j = 0; j < 10; j++) {
            int *vetor = (int *)malloc(tamanhoVetor * sizeof(int));

            // Preenche o vetor com valores aleatorios
            srand(time(NULL) + j); // Adiciona um valor diferente  seed do srand
            for (int i = 0; i < tamanhoVetor; i++) {
                vetor[i] = rand() % 100;
            }

            // ShellSort no vetor e contando o tempo
            gettimeofday(&tempo_inicial, NULL);
            shellSort(vetor, tamanhoVetor);
            gettimeofday(&tempo_final, NULL);

            // Somando os 10 tempos
            double deltaTempo = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
            somaDeltaTempo += deltaTempo;



            free(vetor);
        }
        // Realizando a media dos 10 tempos
        mediaDeltaTempo = somaDeltaTempo / 10.0;
        printf("Media de tempo de execucao do algoritmo foi de: %f segundos \n", mediaDeltaTempo);

        // Tempo do Comportamento Natural
        int *vetorNatural7 = (int *)malloc(tamanhoVetor * sizeof(int));
        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorNatural7[i] = i;
        }
```

```
// Mede o tempo de execucao do algoritmo para o comportamento natural
gettimeofday(&tempo_inicial, NULL);
shellSort(vetorNatural7, tamanhoVetor);
gettimeofday(&tempo_final, NULL);

// Calcula o tempo decorrido
double deltaTempoNatural7 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
printf("Tempo do comportamento natural do algoritmo foi de: %f segundos \n", deltaTempoNatural7);


// Tempo do Pior Caso:
// Isso significa que os valores no vetor estao em ordem decrescente, comecando com "tamanhoVetor" na primeira posicao e terminando com "1" na ultima posicao
int *vetorPiorCaso7 = (int *)malloc(tamanhoVetor * sizeof(int));
for (int i = 0; i <= tamanhoVetor; i++) {
   vetorPiorCaso7[i] = tamanhoVetor - i;
}

// Mede o tempo de execucao do algoritmo para o pior caso
gettimeofday(&tempo_inicial, NULL);
shellSort(vetorPiorCaso7, tamanhoVetor);
gettimeofday(&tempo_final, NULL);

// Calcula o tempo decorrido
double deltaTempoPior7 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
printf("Tempo do pior caso do algoritmo foi de: %f segundos \n", deltaTempoPior7);
```

```
        // Libera a memoria alocada para os vetores
        free(vetorNatural7);
        free(vetorPiorCaso7);

        break;

    case 8:
        // Algoritmo TimSort
        for (int j = 0; j < 10; j++) {
            int *vetor = (int *)malloc(tamanhoVetor * sizeof(int));

            // Preenche o vetor com valores aleatorios
            srand(time(NULL) + j); // Adiciona um valor diferente  seed do srand
            for (int i = 0; i < tamanhoVetor; i++) {
                vetor[i] = rand() % 100;
            }

            // TimSort no vetor e contando o tempo
            gettimeofday(&tempo_inicial, NULL);
            timSort(vetor, tamanhoVetor);
            gettimeofday(&tempo_final, NULL);

            // Somando os 10 tempos
            double deltaTempo = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
            somaDeltaTempo += deltaTempo;

            free(vetor);
        }
        // Realizando a media dos 10 tempos

        mediaDeltaTempo = somaDeltaTempo / 10.0;
        printf("Media de tempo de execucao do algoritmo foi de: %f segundos \n",
```

```
mediaDeltaTempo);

        // Tempo do Comportamento Natural
        int *vetorNatural8 = (int *)malloc(tamanhoVetor * sizeof(int));
        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorNatural8[i] = i;
        }

        // Mede o tempo de execucao do algoritmo para o comportamento natural
        gettimeofday(&tempo_inicial, NULL);
        timSort(vetorNatural8, tamanhoVetor);
        gettimeofday(&tempo_final, NULL);

        // Calcula o tempo decorrido
        double deltaTempoNatural8 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        printf("Tempo do comportamento natural do algoritmo foi de: %f segundos \n", deltaTempoNatural8);

        // Tempo do Pior Caso:
        // Isso significa que os valores no vetor estao em ordem decrescente, comecando com "tamanhoVetor" na primeira posicao e terminando com "1" na ultima posicao

        int *vetorPiorCaso8 = (int *)malloc(tamanhoVetor * sizeof(int));
        for (int i = 0; i <= tamanhoVetor; i++) {
            vetorPiorCaso8[i] = tamanhoVetor - i;
        }

        // Mede o tempo de execucao do algoritmo para o pior caso
        gettimeofday(&tempo_inicial, NULL);
        timSort(vetorPiorCaso8, tamanhoVetor);
        gettimeofday(&tempo_final, NULL);
```

```
        // Calcula o tempo decorrido
        double deltaTempoPior8 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
        printf("Tempo do pior caso do algoritmo foi de: %f segundos \n", deltaTempoPior8);

        // Libera a memoria alocada para os vetores
        free(vetorNatural8);
        free(vetorPiorCaso8);

        break;
    case 9:
        // Algoritmo RadixSort
        for (int j = 0; j < 10; j++) {

            int *vetor = (int *)malloc(tamanhoVetor * sizeof(int));

            // Preenche o vetor com valores aleatorios
            srand(time(NULL) + j); // Adiciona um valor diferente  seed do srand
            for (int i = 0; i < tamanhoVetor; i++) {
                vetor[i] = rand() % 100;
            }

            // RadixSort no vetor e contando o tempo
            gettimeofday(&tempo_inicial, NULL);
            radix_sort(vetor, tamanhoVetor);
            gettimeofday(&tempo_final, NULL);

            // Somando os 10 tempos
            double deltaTempo = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
```

```
        somaDeltaTempo += deltaTempo;

        free(vetor);
    }
    // Realizando a media dos 10 tempos
    mediaDeltaTempo = somaDeltaTempo / 10.0;
    printf("Media de tempo de execucao do algoritmo foi de: %f segundos \n", mediaDeltaTempo);

    // Tempo do Comportamento Natural
    int *vetorNatural9 = (int *)malloc(tamanhoVetor * sizeof(int));
    for (int i = 0; i <= tamanhoVetor; i++) {
        vetorNatural9[i] = i;
    }

    // Mede o tempo de execucao do algoritmo para o comportamento natural
    gettimeofday(&tempo_inicial, NULL);
    radix_sort(vetorNatural9, tamanhoVetor);
    gettimeofday(&tempo_final, NULL);

    // Calcula o tempo decorrido
    double deltaTempoNatural9 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
    printf("Tempo do comportamento natural do algoritmo foi de: %f segundos \n", deltaTempoNatural9);

    // Tempo do Pior Caso:
    // Isso significa que os valores no vetor estao em ordem decrescente, comecando com "tamanhoVetor" na primeira posicao e terminando com "1" na ultima posicao
    int *vetorPiorCaso9 = (int *)malloc(tamanhoVetor * sizeof(int));
    for (int i = 0; i <= tamanhoVetor; i++) {
        vetorPiorCaso9[i] = tamanhoVetor - i;
```

```
            }

            // Mede o tempo de execucao do algoritmo para o pior caso
            gettimeofday(&tempo_inicial, NULL);
            radix_sort(vetorPiorCaso9, tamanhoVetor);
            gettimeofday(&tempo_final, NULL);

            // Calcula o tempo decorrido
            double deltaTempoPior9 = (tempo_final.tv_sec + tempo_final.tv_usec / 1000000.0) - (tempo_inicial.tv_sec + tempo_inicial.tv_usec / 1000000.0); // Faz o calculo para deixar em segundos
            printf("Tempo do pior caso do algoritmo foi de: %f segundos \n", deltaTempoPior9);

            // Libera a memoria alocada para os vetores
            free(vetorNatural9);
            free(vetorPiorCaso9);

            break;
        default:
            printf("Escolha invalida\n");
            return 1;

        }

        printf("\nDeseja um novo teste?\nsim = 1 nao = 2\n");//loop para utilizacao do menu

        scanf("%d", &teste);

    } while (teste == 1);
        system("cls");
    return 0;
```

```
}
```

6.2 projeto.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include "projeto.h"

#define RUN 2

// Algoritmo QuickSort
void quickSort(int vetor[], int esquerda, int direita) {
    int i, j, pivo, aux;
    i = esquerda;
    j = direita;
    pivo = vetor[(esquerda + direita) / 2];

    while (i <= j) {
        while (vetor[i] < pivo && i < direita) {
            i++;
        }
        while (vetor[j] > pivo && j > esquerda) {
            j--;
        }
        if (i <= j) {
            aux = vetor[i];
            vetor[i] = vetor[j];

            vetor[j] = aux;
            i++;
            j--;
        }
    }
    if (j > esquerda) {
```

```
        quickSort(vetor, esquerda, j);
    }
    if (i < direita) {
        quickSort(vetor, i, direita);
    }
}

// Algoritmo MergeSort
void merge(int arr[], int inicio, int meio, int fim) {
    int i, j, k;
    int n1 = meio - inicio + 1;
    int n2 = fim - meio;

    int INICIO[n1], FIM[n2];

    for (i = 0; i < n1; i++)
        INICIO[i] = arr[inicio + i];
    for (j = 0; j < n2; j++)

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

    i = 0;
    j = 0;
    k = inicio;

    while (i < n1 && j < n2) {
        if (INICIO[i] <= FIM[j]) {
            arr[k] = INICIO[i];
            i++;
        } else {
            arr[k] = FIM[j];
            j++;
        }
        k++;
```

```
    }

    while (i < n1) {
        arr[k] = INICIO[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = FIM[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int inicio, int fim) {
    if (inicio < fim) {
        int meio = inicio + (fim - inicio) / 2;

        mergeSort(arr, inicio, meio);
        mergeSort(arr, meio + 1, fim);

        merge(arr, inicio, meio, fim);
    }
}

// Algoritmo InsertionSort
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
```

```
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// Algoritmo SelectionSort
void selectionSort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n - 1; i++) {
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

// Algoritmo HeapSort

void heapify(int vet[], int n, int i) {
    int max = i;

    int e = 2 * i + 1;
    int d = 2 * i + 2;

    if (e < n && vet[e] > vet[max])
        max = e;

    if (d < n && vet[d] > vet[max])
```

```
        max = d;

    if (max != i) {
        int aux = vet[i];
        vet[i] = vet[max];
        vet[max] = aux;

        heapify(vet, n, max);
    }
}

void heapSort(int vet[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(vet, n, i);

    for (int i = n - 1; i >= 0; i--) {
        int aux = vet[0];
        vet[0] = vet[i];
        vet[i] = aux;

        heapify(vet, i, 0);
    }
}

// Algoritmo BubbleSort
void bubbleSort(int *v, int n) {
    int i, continua, aux, fim = n;
    do {
        continua = 0;
        for (i = 0; i < fim - 1; i++) {
            if (v[i] > v[i + 1]) {
                aux = v[i];
                v[i] = v[i + 1];
                v[i + 1] = aux;
```

```
                continua = i;
            }
        }
        fim--;

    } while (continua != 0);
}

// Algoritmo ShellSort
void shellSort(int L[], int n) {
    int temp, i, j, m;
    for (m = n / 2; m > 0; m /= 2) {
        for (j = m; j < n; j++) {
            for (i = j - m; i >= 0; i -= m) {
                if (L[i + m] >= L[i])
                    break;
                else {
                    temp = L[i];
                    L[i] = L[i + m];
                    L[i + m] = temp;
                }
            }
        }
    }
}

// Algoritmo TimSort
void insertion(int arr[], int l, int r) {
    if (arr == NULL || l>=r) return;

    int key;
    int i, j;
    for (i=l+1; i<=r; i++) {
        key = arr[i];
```

```
        j = i-1;
        while (j>=l && arr[j]>key) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }

}

void mergesort(int arr[], int temp[], int l, int m, int r) {
    if (arr == NULL || temp == NULL) return;
    if (l > m || m+1 > r) return;

    int i = l;
    int j = m + 1;
    int start = l;

    while (i <= m && j <= r) {

        if (arr[i] < arr[j]) {
            temp[start++] = arr[i++];
        } else if (arr[i] == arr[j]) {

            temp[start++] = arr[i++];
            temp[start++] = arr[j++];
        } else {
            temp[start++] = arr[j++];
        }

    }

    while (i <= m) {
        temp[start++] = arr[i++];
```

```
    }


    while (j <= r) {
        temp[start++] = arr[j++];
    }

    for (i = l; i <= r; i++) {
        arr[i] = temp[i];
    }

}

int minsort(int arg1, int arg2) {
    if (arg1 <= arg2) {
        return arg1;
    } else {
        return arg2;
    }
}

void timSort(int arr[], unsigned int size) {
    if (arr == NULL || size <= 1) return;
    for (int i=0; i < size; i+=RUN) {
        insertion(arr, i, minsort(i+RUN-1, size-1));
    }

    int* temp = (int*)malloc(sizeof(int)*size);


    int l, m, r, n;
    for (n=RUN; n < size; n*=2) {
        for (l=0; l < size; l+=2*n) {
            m = l+n-1;
```

```
            r = minsort(l+2*n-1, size-1);

            if (m<r) {
                mergesort(arr, temp, l, m, r);
            }
        }
    }
    free(temp);
}

// Algoritmo RadixSort
int find_max(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

void radix_sort(int arr[], int n) {
    int max = find_max(arr, n);
    int exp = 1;
    int *temp = malloc(n * sizeof(int));
    while (max / exp > 0) {

        int count[10] = {0};
        for (int i = 0; i < n; i++) {
            count[(arr[i] / exp) % 10]++;
        }
        for (int i = 1; i < 10; i++) {
            count[i] += count[i - 1];
        }
```

```
        for (int i = n - 1; i >= 0; i--) {
            temp[count[(arr[i] / exp) % 10] - 1] = arr[i];
            count[(arr[i] / exp) % 10]--;
        }
        for (int i = 0; i < n; i++) {
            arr[i] = temp[i];
        }
        exp *= 10;
    }
    free(temp);
}
```

6.3 projeto.h

```
// Algoritmo QuickSort
void quickSort(int vetor[], int esquerda, int direita);

// Algoritmo MergeSort
void merge(int arr[], int inicio, int meio, int fim);

void mergeSort(int arr[], int inicio, int fim);

// Algoritmo InsertionSort
void insertionSort(int arr[], int n);

// Algoritmo SelectionSort
void selectionSort(int arr[], int n);

// Algoritmo HeapSort
void heapify(int vet[], int n, int i);

void heapSort(int vet[], int n);

// Algoritmo BubbleSort
```

```
void bubbleSort(int *v, int n);

// Algoritmo ShellSort
void shellSort(int L[], int n);

// Algoritmo TimSort
void insertion(int arr[], int l, int r);

void mergesort(int arr[], int temp[], int l, int m, int r);

int minsort(int arg1, int arg2);

void timSort(int arr[], unsigned int size);

// Algoritmo RadixSort
int find_max(int arr[], int n);

void radix_sort(int arr[], int n);
```

7. REFERÊNCIAS

DIAS, Cayo. **Algoritmos de ordenação explicados com exemplos em Python, Java e C++.** Disponível em: <https://www.freecodecamp.org/portuguese/news/algoritmos-de-ordenacao-explicados-com-exemplos-em-python-java-e-c/> Acesso em: 21 de abril de 2023.

ROSA, Rodrigo Pansolim. ALMEIDA, William. **TimSort.** Disponível em: <https://deinfo.uepg.br/~alunoso/2019/AEP/TIMSORT/REA-TimSort.htm#:~:text=Resumindo%3A,ordenada%20apenas%20usando%20o%20InsertionSort.> Acesso em: 22 de abril de 2023.

BORIN, Edson. **Algoritmos e Programação de Computadores.** Disponível em: <https://www.ic.unicamp.br/~edson/disciplinas/mc102/2019-1s/ef/slides/MC102-Aula01.pdf > Acesso em: 28 de abril de 2023.

ALFREDO, Antonio. **Análise de Complexidade** Disponível em:<https://homepages.dcc.ufmg.br/~loureiro/alg/071/paa_Analise_4pp.pdf> Acesso em: 29 de abril de 2023.