Trabalho 2 P.O.T.A: Teste Prático com Algoritmos de Ordenação

Euler de Azevedo Costa, Danrley Gomes dos Santos, Fernando Custódio Santiago, Heudmann Osmídio Lima, Jhonatan Tibiqueira Sarmento, José Felipe Albuquerque

Centro Universitário do Norte (Uninorte) – Laureate International Universities.
Escola de Ciências Exatas e Tecnologia – Ciência da Computação/Engenharia da Computação
CEP 69020-220 –Manaus –Amazonas

euller.azevedo@gmail.com, danrleyifam@gmail.com, nando.custodio@yahoo.com.br, heudmannlima@gmail.com, jhonatan.sarmento@gmail.com, josefelipeas1@gmail.com

Abstract. *This document consists of a series of tests with the following sorting algorithms: Selection Sort, Sort Insertion, Bubble Sort, Sort Sort, Heap Sort, Merge Sort and Quick Sort, using metric established as number of comparisons and number of exchanges, through of randomly filled vectors, assigning the following results in graphs and tables to the appropriate comparisons.*

Keywords: Ordering Algorithms, Data Structure.

Resumo. *O trabalho consiste em uma série de testes com os seguintes algoritmos de ordenação: Selection Sort, Insertion Sort, Bubble Sort, Shell Sort, Heap Sort, Merge Sort e Quick Sort, utilizando métricas estabelecidas como número de comparações e número de trocas, através de vetores preenchidos aleatoriamente, atribuindo os seguintes resultados em gráficos e tabelas para as devidas comparações*

Palavras-chave: Algoritmos de Ordenação, Estrutura de Dados.

1. Introdução

O termo ordenação, não se delimita somente na computação. Ele está presente nas outras áreas da concepção do conhecimento humano. É comum encontrar exemplos que se aplicam em outras áreas de estudo no dia a dia como a gramática, onde um dicionário composto por centenas de palavras é organizado de forma sequencial, começando pelas letras, primeira sílaba, segunda sílaba respectivamente para melhor controle de pesquisas das palavras e, neste contexto, a eficiência do manuseio de dados muitas das vezes pode ser substancialmente aumentada se os dados forem dispostos de acordo com algum critério de ordem.

Neste sentido, existem vários métodos de ordenação. Na computação esses algoritmos tem como finalidade, organizar os dados em uma determinada ordem. São

métodos que possuem o mesmo objetivo, mas existem algumas diferenças que serão abordadas logo a seguir. Geralmente o que impacta muito nas diferenças é a capacidade de processamento, dependendo do tipo e do modelo do processador. Existem alguns métodos que usam um pouco mais de tempo para ordenar um conjunto de dados, devido ao número de comparações e trocas que os mesmos executam em um determinado processo. Já outros demais métodos realizam essa tarefa com muito mais rapidez, por executarem subdivisões e menos comparações na mesma sequência.

Com base na aplicação destes algoritmos de ordenação, no presente trabalho será demonstrado uma série de testes com os algoritmos de ordenação Selection Sort, Insertion Sort, Bubble Sort, Shell Sort, Heap Sort, Merge Sort e Quick Sort, visando extrair diferentes tipos de dados como número de comparações e número de trocas, através de vetores preenchidos aleatoriamente, atribuindo os seguintes resultados em gráficos e tabelas para serem comparados.

Ao fim, iremos analisar todos os resultados dentre uma gama de dados aleatórios para poder comparar a eficiência de cada algoritmo de ordenação aplicado.

3. Procedimento Experimental

3.1 Especificações da máquina de testes:

Fabricante: Acer

Modelo: Aspire 574Z-4459

Processador: Intel(R) Core(TM) i5 CPU M430 - 2.27GHz - Dual Core x64

Memória RAM: 4,00GB

Sistema Operacional: Microsoft Windows 10 Professional 64-bit

Compilador: DEV C++ Version: 5.11 TDM-GCC Compiler

3.2 Experimentos em Prática

Os testes consistiram em extrair diferentes dados: Número de comparações e número de trocas. Em cada rodada de execução foram utilizados 50 tipos de vetores de testes, cada um com inserção aleatória de elementos e, conforme as rodadas aumentavam, o tamanho do vetor era alocado de 50 em 50 vezes para verificar o compostamento dos algoritmos de ordenação, obtendo assim a média dos resultados e inserindo-os em tabelas.

4. Análise dos Resultados

Na 1ª Rodada, obteve-se os seguintes resultados inseridos na tabela para a montagem do primeiro gráfico:

Algoritmo	Tamanho dos Vetores	Quantidade de Vetores	Média de Comparações	Média de Trocas
Selection Sort	10	50	45	12
Insertion Sort	10	50	32	23
Bubble Sort	10	50	81	23
Shell Sort	10	50	28	13
Heap Sort	10	50	28	26
Merge Sort	10	50	22	10
Quick Sort	10	50	22	11

Tabela 1. 1ª Rodada de Execução do Programa de Testes

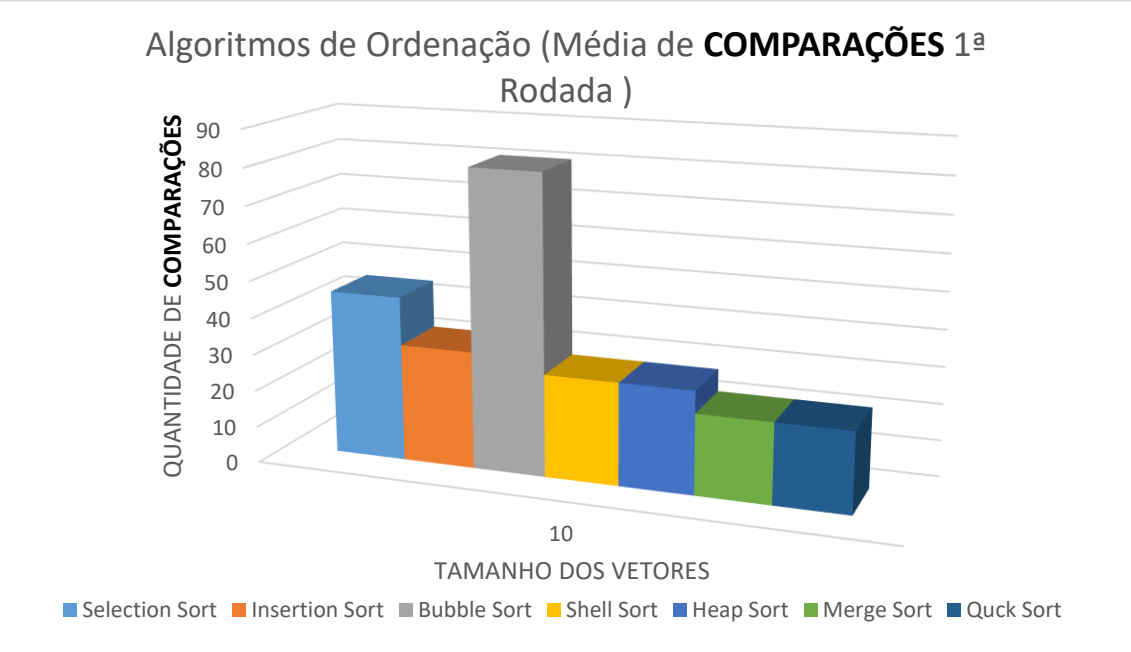

Gráfico 1. Média de Comparações da 1ª Rodada

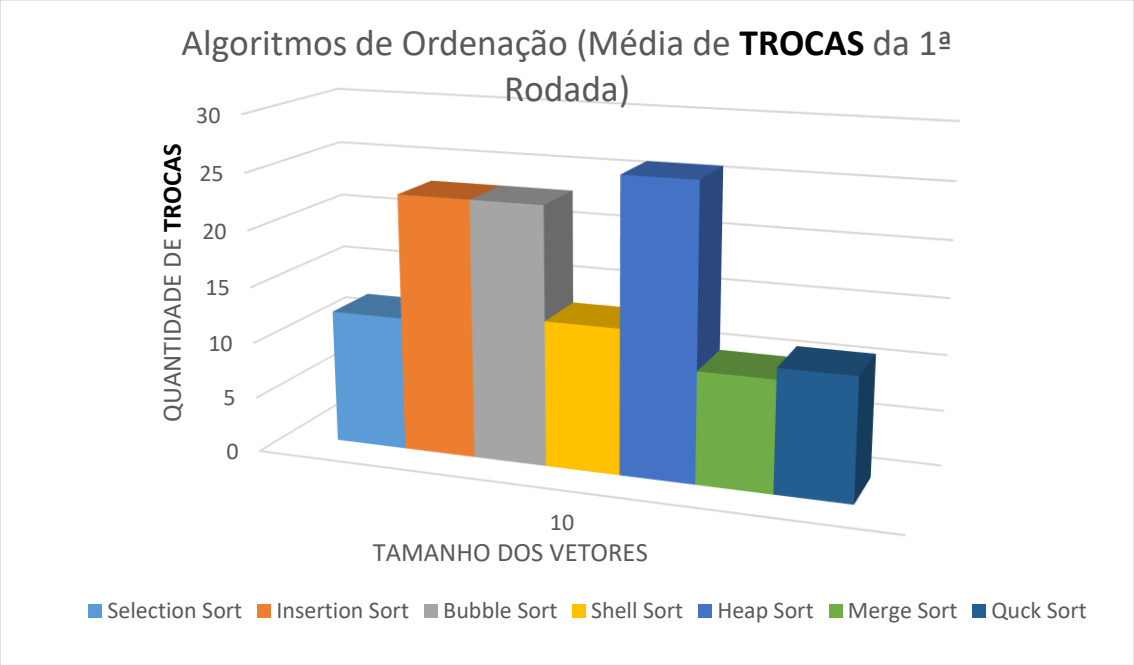

Gráfico 2. Média de Trocas da 1ª Rodada

Na tabela e nos gráficos acima, pode-se observar claramente que com vetores pequenos o Quick Sort e o Merge Sort já levam uma leve vantagem sobre os demais, mesmo com um tamanho de vetor muito pequeno. O Bubble por sua vez já assusta com uma média muito alta no número de comparações, apesar do baixo tamanho dos vetores da rodada. Já o Selection Sort e o Insertion Sort se comportam muito bem, se comparando ao Quick Sort e Merge Sort. O Heap Sort começa executando muito mais trocas do que os demais, porém está empatado com o Shell Sort na média de comparações.

Ao chegar na 5ª Rodada, já com um aumento significativo no tamanho dos vetores, obteve-se os seguintes resultados inseridos na tabela:

Algoritmo	Tamanho dos Vetores	Quantidade de Vetores	Média de Comparações	Média de Trocas
Selection Sort	210	50	21945	822
Insertion Sort	210	50	11157	10948
Bubble Sort	210	50	43681	10948
Shell Sort	210	50	2021	1150
Heap Sort	210	50	1912	1442
Merge Sort	210	50	1311	438
Quick Sort	210	50	1360	273

Tabela 2. 5ª Rodada de Execução do Programa de Testes

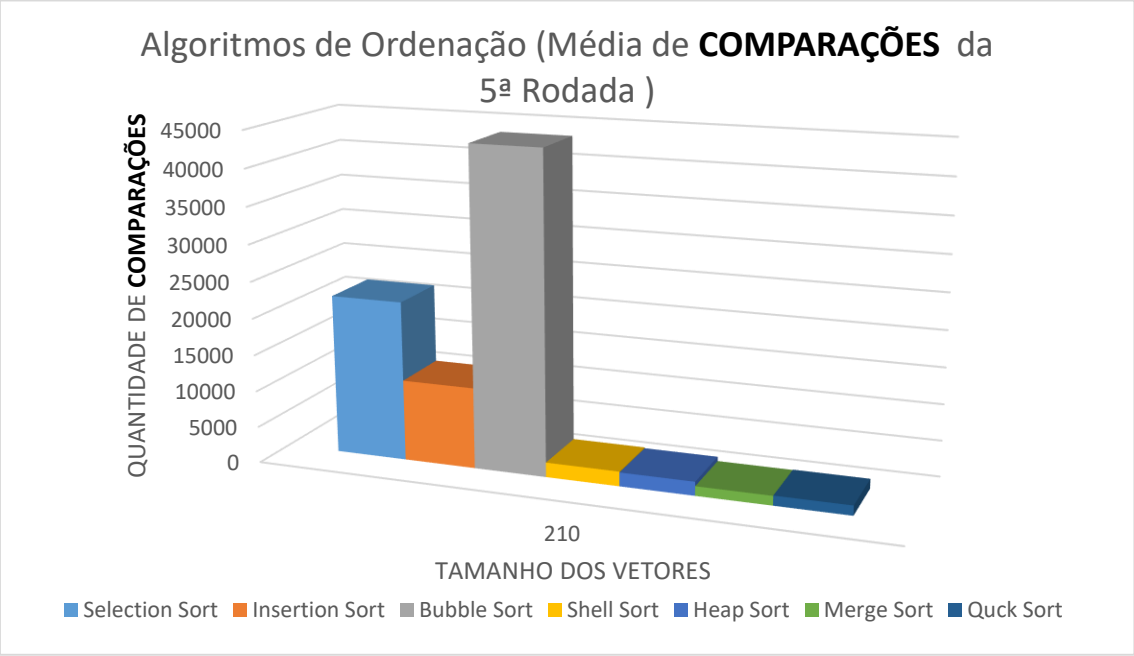

Gráfico 3. Média de Comparações da 5ª Rodada

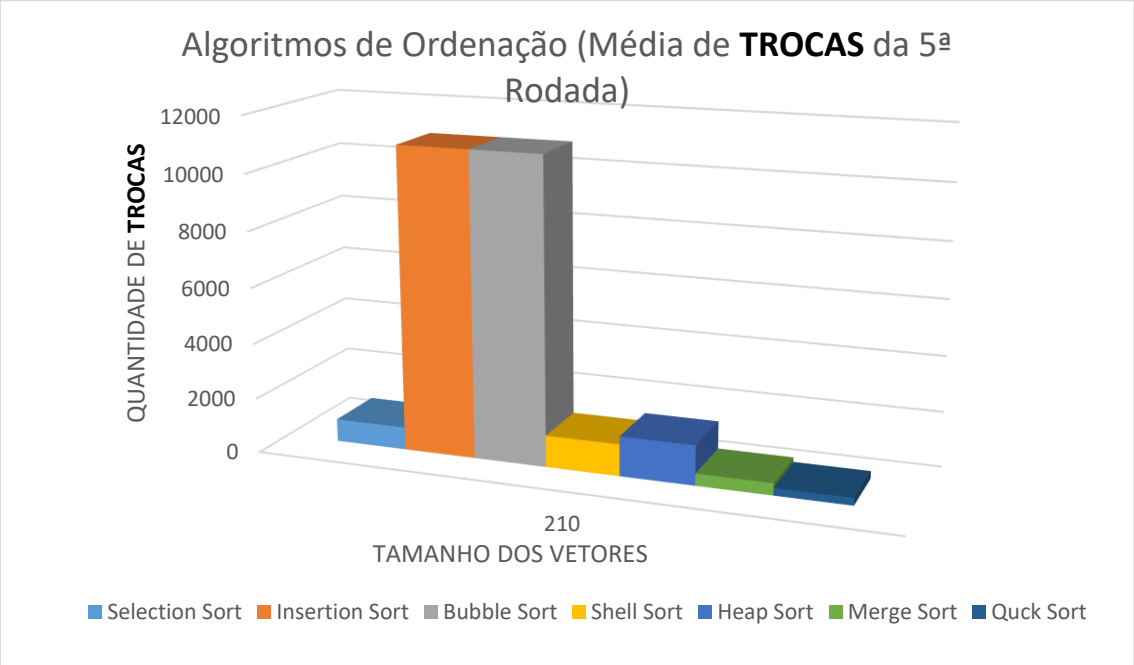

Gráfico 4. Média de Trocas da 5ª Rodada

Na tabela e gráficos acima, o Quick Sort e o Merge Sort já conseguem levar uma vantagem maior sobre os outros. Apesar de o Bubble Sort apresentar uma média de trocas idêntica ao do Insertion Sort, já perde muito com sua média de comparações chegando a ser absurda em relação aos seus concorrentes.

Nota-se que o Shell Sort encontra-se em uma zona de meio-termo comparando-se aos demais, pois trata-se de um algoritmo bom para ordenar um número moderado de elementos, pois quando encontra um arquivo parcialmente ordenado trabalha menos.

O Selection Sort e o Insertion Sort, se comparando ao Quick Sort e Merge Sort, já começam perder sua eficiência com o aumento do tamanho dos vetores. O Heap Sort apesar de executar muitas trocas, em relação ao Quick Sort e Merge Sort, se mantém em vantagem em relação aos demais.

Chegando na 10ª Rodada, com o tamanho dos vetores aumentados ainda mais, obteve-se os seguintes resultados inseridos na tabela:

Algoritmo	Tamanho dos Vetores	Quantidade de Vetores	Média de Comparações	Média de Trocas
Selection Sort	460	50	105570	2121
Insertion Sort	460	50	52593	52134
Bubble Sort	460	50	210681	52134
Shell Sort	460	50	5400	3183
Heap Sort	460	50	4974	3674
Merge Sort	460	50	3353	1089
Quick Sort	460	50	3495	592

Tabela 3. 10ª Rodada de Execução do Programa de Testes

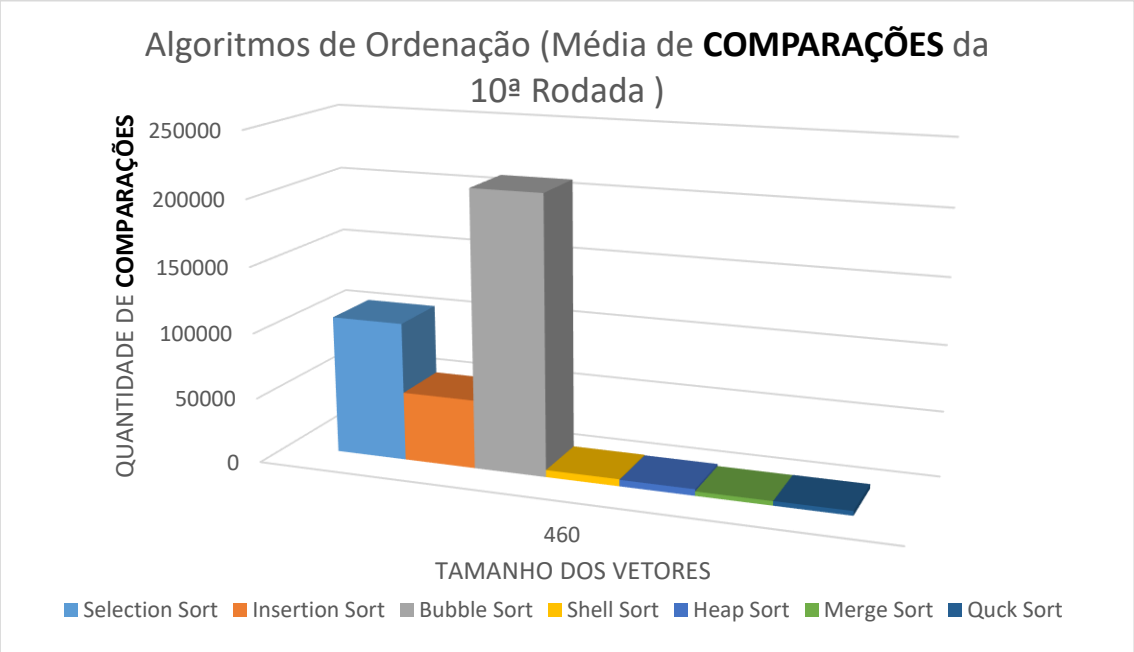

Gráfico 5. Média de Comparações da 10ª Rodada

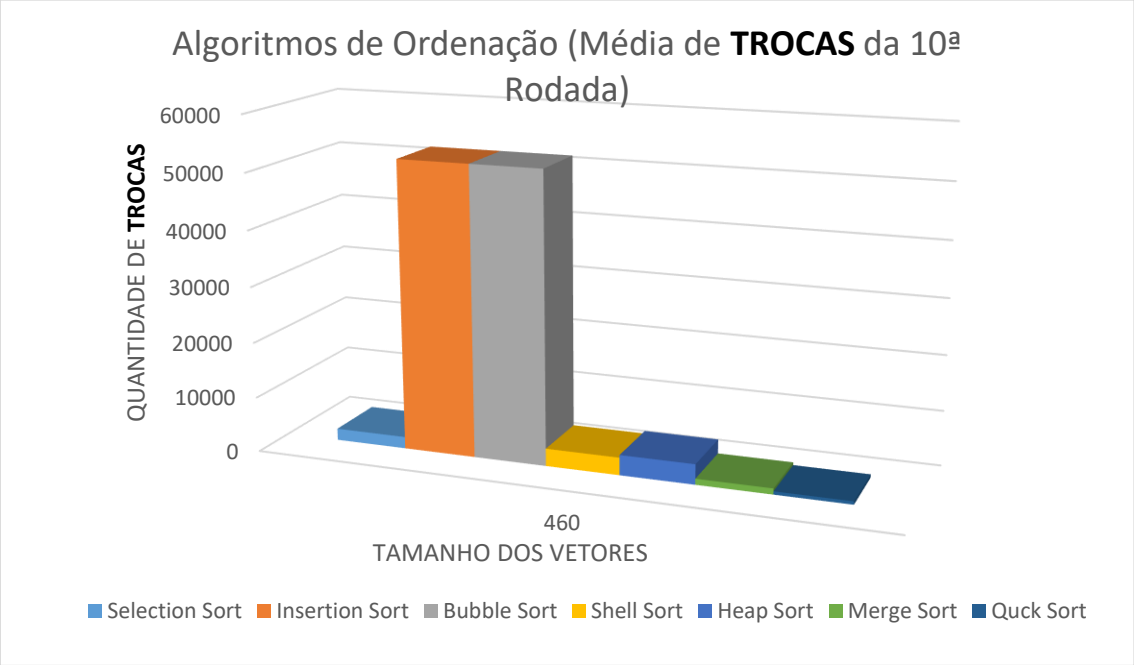

Gráfico 6. Média de Trocas da 10ª Rodada

Na tabela acima, o Quick Sort e o Merge Sort já se sobressaem em relação aos outros. O Shell Sort, Quick Sort e o Heap Sort têm a mesma ordem de grandeza para arranjos, sendo que o Quicksort é o mais eficiente para todos os tamanhos aleatórios experimentados até aqui.

Tanto o Bubble Sort quanto o Selection Sort possuem complexidade $O(n^2)$ em todos os casos. O Insertion Sort, por outro lado, roda em O(n) no melhor caso (como por exemplo, se o vetor já estiver em ordem crescente). Assim, ele acaba sendo um algoritmo melhor para propósitos específicos, apesar de suas limitações.

O Quick Sort certamente é o algoritmo mais eficiente em listas totalmente desordenadas, ele se torna muito eficiente em relação aos outros nos dois quesitos. Nos testes de execução, a diferença entre do Quick Sort em comparação aos outros foi absurdamente grande.

Já o Merge Sort, em comparação a outros algoritmos de divisão e conquista, como o Quick sort, apresenta a mesma complexidade, mas com algoritmos mais básicos de ordenação por comparação e troca (Bubble, Insertion e Selection Sort), o Merge é mais rápido e eficiente quando é utilizado sobre uma grande quantidade de dados. Para entradas pequenas os algoritmos de ordenação por comparação mais básicos são pró-eficientes.

O Bubble sort apresenta melhor caso como O(n) porque o algoritmo pode ser modificado de forma que, se a lista já estiver ordenada, basta apenas uma verificação básica que custa O(n). O Quick sort pode atingir $O(n^2)$ em um caso específico quando o particionamento é desequilibrado.

Com base nos resultados obtidos, construiu-se dois gráficos gerais das rodadas, colocando todos os dados para comparações

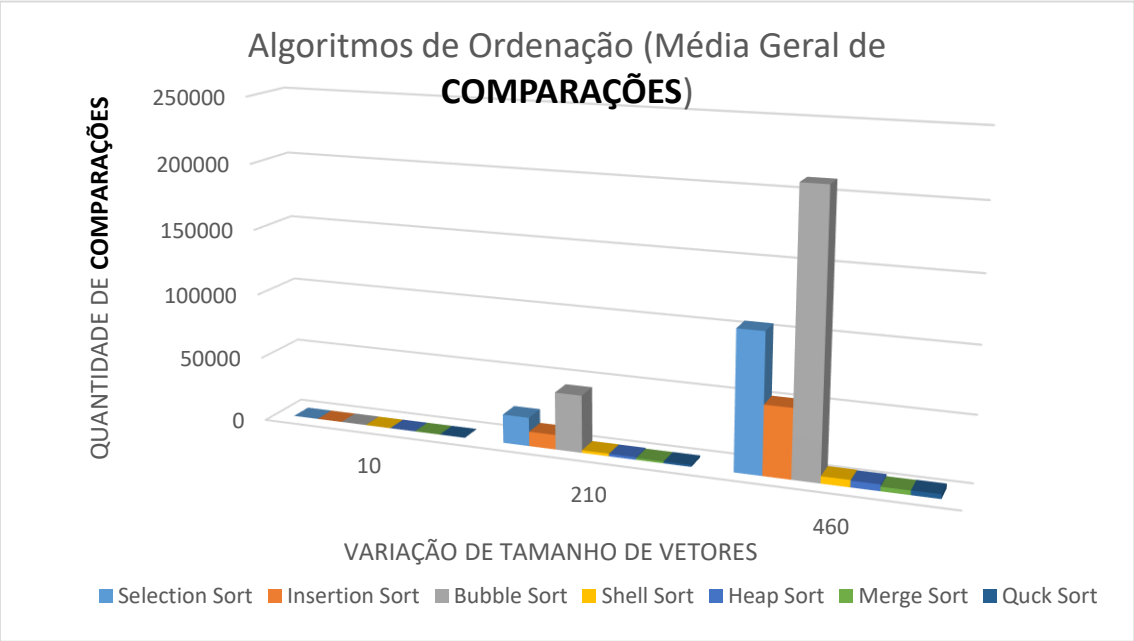

Gráfico 7. Média Geral de Comparações

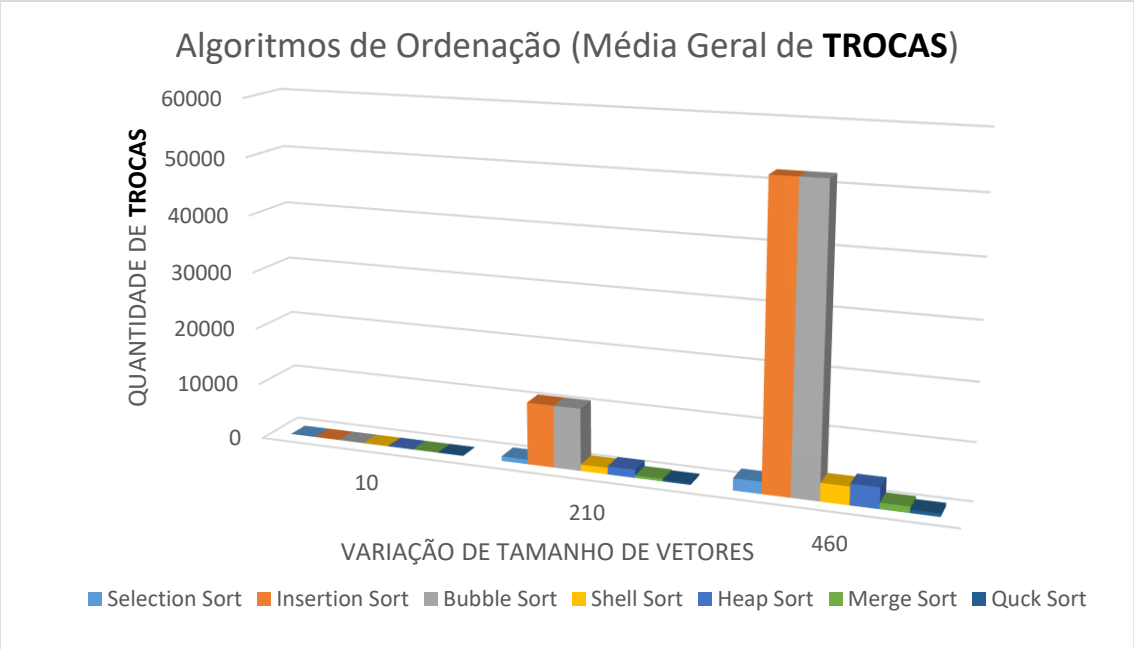

Gráfico 8. Média Geral de Trocas

Nos gráficos acima pode ser observado a eficiência dos algoritmos de ordenação em relação ao número de comparações e trocas em todas as rodadas, mostrando que qunato maior o tamanho do vetor, uns permanecem eficientes como é o caso do Quick

Sort, e outros vão perdendo sua eficiência como o Bubble Sort. Através destas observações é que se pode concluir sobre o comportamento dos algoritmos.

5. Conclusões

Os objetivos do trabalho foram alcançados com sucesso, ou seja, foi possível construir todos os algoritmos e apresenta-los neste relatório juntamente com estatísticas de desempenho de cada um. Os testes unitários foram imprescindíveis para a perfeita conclusão de todos os algoritmos. Mesmo depois de concluído cada um, foi necessário passar por uma fase de revisão e refinação o que permitiu serem feitas de forma muito mais rápidas e assertivas, visto que existiam testes validando a ordenação de todos os algoritmos.

Verificou-se que com valores aleatórios e de grandes vetores, os algoritmos de ordenação Quick Sort e Merge Sort se sobressairam aos outros com uma larga vantagem. Por fim o trabalho foi de grande valia para entendimento, principalmente, das diferenças de desempenho entre os algoritmos de ordenação e compreender que sempre existem métodos mais eficientes de resolver um problema recorrente.

6. Referências

Drozdek, Adam. Estrutura de Dados em C++ / Adam Drozdek. São Paulo: Cengage Learning, 2010.

https://pt.wikipedia.org/wiki/Ordenação_(computação)

https://pt.wikipedia.org/wiki/Algoritmo_de_ordenação

Anexo 01 : Código Fonte em C do Trabalho

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int comp_insert = 0, troca_insert = 0, ac_insert_troc = 0, ac_insert_comp =
0, md_insert_comp = 0, md_insert_troc = 0;
int comp_bubble = 0, troca_bubble = 0, ac_bubble_troc = 0, ac_bubble_comp =
0, md_bubble_comp = 0, md_bubble_troc = 0;
int comp_select = 0, troca_select = 0, ac_select_troc = 0, ac_select_comp =
0, md_select_comp = 0, md_select_troc = 0;
int comp_shell  = 0, troca_shell  = 0, ac_shell_troc  = 0, ac_shell_comp  =
0, md_shell_comp =  0, md_shell_troc  = 0;
int comp_heap   = 0, troca_heap   = 0, ac_heap_troc   = 0, ac_heap_comp   =
0, md_heap_comp   = 0, md_heap_troc   = 0;
int comp_quick  = 0, troca_quick  = 0, ac_quick_troc  = 0, ac_quick_comp  =
0, md_quick_comp  = 0, md_quick_troc  = 0;
int comp_merge  = 0, troca_merge  = 0, ac_merge_troc  = 0, ac_merge_comp  =
0, md_merge_comp  = 0, md_merge_troc  = 0;

void bubble_sort(int *nrs, int tam);
void selection_sort(int *nrs, int tam);
void insertion_sort(int *original, int n);
void heap_sort(int *a, int n);
void shell_sort(int *vet, int size);
void quick_sort(int *nrs, int esquerda, int direita);
void merge_sort(int *vetor, int comeco, int fim);
void merge(int *vetor, int comeco, int meio, int fim);

int main(void) {
int *vet, *cop, range, i, j, k, x, z;
int num = 10, rodada = 0, qtd = 50; //(qtd) - Quantidade de vetores por
rodada | (num) - qtd de rodadas
int tam = 10;//(tam) - vetor inicial tamanho 10 posições
range=(1000-0)+1;//(range) - valores aleatórios entre 0 a 1000 para preencher
vetor
srand((unsigned)time(NULL));
vet = (int*)calloc(tam,sizeof(int));
cop = (int*)calloc(tam,sizeof(int));

while(rodada < num)
{
rodada++;
printf("\n\n\n\n - - Iniciando %da. Rodada - - \n",rodada);

for(k = 0; k < qtd; k++)
{
printf("\n\n\n Vetor %d de [%d] posicoes:",k+1,tam);
for (j = 0; j < tam ; j++)
{
vet[j] = (0+(rand()%range));
//printf("  %d",vet[j]);//Printa cada vetor
}

for(z=0; z<tam; z++) cop[z]=vet[z];
```

```
selection_sort(cop,tam);
printf("\n - SELECTION:\tTroca: %d\tComparacao:
%d",troca_select,comp_select);
ac_select_troc += troca_select;
ac_select_comp += comp_select;
troca_select = 0;
comp_select = 0;

for(z=0; z<tam; z++) cop[z]=vet[z];
insertion_sort(cop,tam);
printf("\n - INSERTION:\tTroca: %d\tComparacao:
%d",troca_insert,comp_insert);
ac_insert_troc += troca_insert;
ac_insert_comp += comp_insert;
troca_insert = 0;
comp_insert = 0;

for(z=0; z<tam; z++) cop[z]=vet[z];
bubble_sort(cop,tam);
printf("\n - BUBBLE:\tTroca: %d\tComparacao: %d",troca_bubble,comp_bubble);
ac_bubble_troc += troca_bubble;
ac_bubble_comp += comp_bubble;
troca_bubble = 0;
comp_bubble = 0;

for(z=0; z<tam; z++) cop[z]=vet[z];
shell_sort(cop,tam);
printf("\n - SHELL:\tTroca: %d\tComparacao: %d",troca_shell,comp_shell);
ac_shell_troc += troca_shell;
ac_shell_comp += comp_shell;
troca_shell = 0;
comp_shell = 0;

for(z=0; z<tam; z++) cop[z]=vet[z];
heap_sort(cop,tam);
printf("\n - HEAP:\tTroca: %d\tComparacao: %d",troca_heap,comp_heap);
ac_heap_troc += troca_heap;
ac_heap_comp += comp_heap;
troca_heap = 0;
comp_heap = 0;

for(z=0; z<tam; z++) cop[z]=vet[z];
quick_sort(cop,0,tam-1);
printf("\n - QUICK:\tTroca: %d\tComparacao: %d",troca_quick,comp_quick);
ac_quick_troc += troca_quick;
ac_quick_comp += comp_quick;
troca_quick = 0;
comp_quick = 0;

for(z=0; z<tam; z++) cop[z]=vet[z];
merge_sort(cop,0,tam-1);
printf("\n - MERGE:\tTroca: %d\tComparacao: %d",troca_merge,comp_merge);
ac_merge_troc += troca_merge;
ac_merge_comp += comp_merge;
troca_merge = 0;
comp_merge = 0;
```

```
}
printf("\n\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- -  \n");
printf(" - Resultado das Medias da %da Rodada com Vetores de [%d]
posicoes:\n",rodada,tam);

md_select_troc = ac_select_troc/qtd;
md_select_comp = ac_select_comp/qtd;
printf("\n - MEDIA SELECTION:\tTroca: %d\tComparacao:
%d",md_select_troc,md_select_comp);

md_insert_troc = ac_insert_troc/qtd;
md_insert_comp = ac_insert_comp/qtd;
printf("\n - MEDIA INSERTION:\tTroca: %d\tComparacao:
%d",md_insert_troc,md_insert_comp);

md_bubble_troc = ac_bubble_troc/qtd;
md_bubble_comp = ac_bubble_comp/qtd;
printf("\n - MEDIA BUBBLE:\tTroca: %d\tComparacao:
%d",md_bubble_troc,md_bubble_comp);

md_shell_troc = ac_shell_troc/qtd;
md_shell_comp = ac_shell_comp/qtd;
printf("\n - MEDIA SHELL:\t\tTroca: %d\tComparacao:
%d",md_shell_troc,md_shell_comp);

md_heap_troc = ac_heap_troc/qtd;
md_heap_comp = ac_heap_comp/qtd;
printf("\n - MEDIA HEAP:\t\tTroca: %d\tComparacao:
%d",md_heap_troc,md_heap_comp);

md_quick_troc = ac_quick_troc/qtd;
md_quick_comp = ac_quick_comp/qtd;
printf("\n - MEDIA QUICK:\t\tTroca: %d\tComparacao:
%d",md_quick_troc,md_quick_comp);

md_merge_troc = ac_merge_troc/qtd;
md_merge_comp = ac_merge_comp/qtd;
printf("\n - MEDIA MERGE:\t\tTroca: %d\tComparacao:
%d\n\n",md_merge_troc,md_merge_comp);

tam += 50;
vet = (int*)realloc(vet,tam*sizeof(int));
cop = (int*)realloc(cop,tam*sizeof(int));

ac_select_troc = 0; ac_select_comp = 0;
ac_insert_troc = 0; ac_insert_comp = 0;
ac_bubble_troc = 0; ac_bubble_comp = 0;
ac_shell_troc  = 0; ac_shell_comp  = 0;
ac_heap_troc   = 0; ac_heap_comp   = 0;
ac_quick_troc  = 0; ac_quick_comp  = 0;
ac_merge_troc  = 0; ac_merge_comp  = 0;
}
}


void insertion_sort(int *original, int n) {
```

```
int i, j, atual;
for (i = 1; i < n; i++)
{
atual = original[i];
j = i - 1;
comp_insert++;
while ((j >= 0) && (atual < original[j])) {
original[j + 1] = original[j];
j--;
comp_insert++;
troca_insert++;
        }
original[j+1] = atual;
}
}

void selection_sort(int *nrs, int tam) {
 int i, j, min, aux;

 for(i = 0; i < tam - 1; i++)
 {
min = i;
for(j= i + 1; j < tam ; j++)
{
comp_select++;
    if ( nrs[j] < nrs[min] ) {
        min = j;
        troca_select++;
        }
    }
aux = nrs[i];
nrs[i] = nrs[min];
nrs[min] = aux;
   }
}

void bubble_sort(int *nrs, int tam) {
int i, j, aux;
for(i = 0; i < tam - 1 ; i++)
for(j= 0; j < tam - 1 ; j++) {
comp_bubble++;
        if ( nrs[j] > nrs[j+1] ) {
        troca_bubble++;
            aux = nrs[j];
            nrs[j] = nrs[j+1];
            nrs[j+1] = aux;
        }
}
}

void heap_sort(int *a, int n) {
   int i = n / 2, pai, filho, t;
   while(true)
   {
      if (i > 0) {
          i--;
          t = a[i];
```

```
        } else {
            n--;
            if (n == 0)
  {
            return;
  }
            t = a[n];
            a[n] = a[0];
            troca_heap++;
        }
        pai = i;
        filho = i * 2 + 1;
        while (filho < n)
  {
  if ((filho + 1 < n)  &&  (a[filho + 1] > a[filho]))
  {
  filho++;
  comp_heap++;
  }
  comp_heap++;
            if (a[filho] > t) {
 a[pai] = a[filho];
               pai = filho;
               filho = pai * 2 + 1;
               troca_heap++;
            } else {
                break;
            }
        }
  a[pai] = t;
   }
}

void shell_sort(int *vvet, int size) {
    int i , j , value;
    int gap = 1;
while(gap < size)
{
        gap = 3*gap+1;
    }
while ( gap > 1)
{
        gap /= 3;
        for(i = gap; i < size; i++)
{
comp_shell++;
            value = vvet[i];
            j = i;

            while (j >= gap && value < vvet[j - gap])
{
                vvet[j] = vvet[j - gap];
                j = j - gap;
                troca_shell++;
                comp_shell++;
            }
            vvet[j] = value;
```

```
        }
    }
}

void quick_sort(int *nrs, int esquerda, int direita) {
    int i, j, meio, aux;
    i = esquerda;
    j = direita;
    meio = nrs[(esquerda + direita) / 2];

    while(i <= j)
{
        while(nrs[i] < meio && i < direita)  {
            i++;
            comp_quick++;
        }
        while(nrs[j] > meio && j > esquerda) {
            j--;
        comp_quick++;
        }
        if(i <= j)
{
            aux = nrs[i];
            nrs[i] = nrs[j];
            nrs[j] = aux;
            i++;
            j--;
            troca_quick++;
        }
    }

    if(j > esquerda)
    {
        quick_sort(nrs, esquerda, j);
        comp_quick++;
    }
    if(i < direita)
    {
        quick_sort(nrs,  i, direita);
        comp_quick++;
    }
}

void merge(int *vetor, int comeco, int meio, int fim) {
    int com1 = comeco, com2 = meio+1, comAux = 0, tam = fim-comeco+1;
    int *vetAux;
    vetAux = (int*)malloc(tam * sizeof(int));

    while(com1 <= meio && com2 <= fim)
{
comp_merge++;
        if(vetor[com1] < vetor[com2]) {
            vetAux[comAux] = vetor[com1];
            com1++;
        } else {
            vetAux[comAux] = vetor[com2];
            com2++;
```

```
        }
        comAux++;
    }

    while(com1 <= meio) {
        vetAux[comAux] = vetor[com1];
        comAux++;
        com1++;
        troca_merge++;
    }
    while(com2 <= fim) {
        vetAux[comAux] = vetor[com2];
        comAux++;
        com2++;
        troca_merge++;
    }
    for(comAux = comeco; comAux <= fim; comAux++){
        vetor[comAux] = vetAux[comAux-comeco];
    }
    free(vetAux);
}

void merge_sort(int *vetor, int comeco, int fim) {
    if (comeco < fim) {
        int meio = (fim+comeco)/2;
        merge_sort(vetor, comeco, meio);
        merge_sort(vetor, meio+1, fim);
        merge(vetor, comeco, meio, fim);
    }
}
```