Universidade Federal de Viçosa
Campus Florestal
CCF 211 - Algoritmos e Estruturas de Dados I

Trabalho Prático 3

Trabalho Prático 03 – AEDS 1

Gabriel Miranda - 3857
Felipe Dias - 3888
Mariana Souza - 3898

Sumário

1 Introdução

O presente trabalho tem como objetivo avaliar e fazer uma comparação mais concreta do desempenho de diversos métodos de ordenação, implementados na linguagem C, contabilizando para cada método, o número de comparações, movimentações realizadas e o tempo gasto para realizar a ordenação dos algoritmos, onde, no primeiro cenário os itens a serem ordenados são de valores inteiros e no segundo cenário são os itens do conjunto de registros, cada registro com um inteiro e dez cadeias de caracteres.

Para o desenvolvimento dos algoritmos, foram utilizados o compilador GCC, as IDEs Visual Studio Code e CodeBlocks, e a plataforma GitHub para cooperação entres os integrantes do grupo.

2 Desenvolvimento

Para alcançar o objetivo do trabalho prático proposto, foi necessário avaliar e comparar o desempenho dos métodos de ordenação, foram implementados os algoritmos de ordenação, Bubblesort, Selectionsort, Insertionsort, Shellsort e Quicksort. Posteriormente os integrantes do grupo decidiram pela implementação do algoritmo Heapsort, Mergesort e Radixsort, porém, para os testes de execução decidimos não fazer a análise do algoritmo de ordenação Radixsort, visto que, a professora da disciplina decidiu em sala que o algoritmo Heapsort não seria um algoritmo obrigatório e sim um optativo para o grupo.

Para uma melhor organização e compreensão do código, implementamos a modularização onde, temos os arquivos denominados “Arraysort.c”, para a ordenação dos vetores, conjuntos de valores inteiros, passados por parâmetro nas funções de ordenação, e os arquivos chamados “Recordsort.c”, contendo os registros, sendo um inteiro e dez cadeias de caracteres, cada uma com 200 caracteres e 4 valores reais.

2.1 Cenário 1

O cenário 1 tem como objetivo realizar testes para ordenação dos itens do conjunto de valores inteiros, esses valores são passados como parâmetros nas funções dos algoritmos de ordenação, onde a função “`InicializaArray`” tem como objetivo inicializar um vetor aleatório por meio da função rand com tamanho N alocado dinamicamente, como podemos visualizar no trecho de código abaixo:

```
int *InicializaArray(int n){
   int *Array = (int *)malloc(n * sizeof(int));
   for (int i = 0; i < n; i++){
      Array[i] = rand() % n;
   }
   return Array;
}
```

No menu estão disponíveis as opções para inicializar o vetor, selecionar o algoritmo de ordenação desejado para ordenar o vetor que foi gerado aleatoriamente.

```
printf("|-------------------------------------------------|\n");
printf("|    [0] - Inicializar Vetor => Array             |\n");
printf("|    [1] - Selecionar o algoritmo de ordenacao    |\n");
printf("|    [2] - Retornar aos Cenarios                  |\n");
printf("|-------------------------------------------------|\n");
```

Figura 1. Menu cenário 1 (Array).

2.2 Cenário 2

Para o cenário 2, foram desenvolvidos TAD's para os registros, onde cada inteiro contém uma chave da ordenação, apresentada no trecho de código abaixo, dez cadeias de caracteres, representados pelo TAD `Titem`. O registro é inicializado com valores aleatórios de tamanho N que são passados como parâmetro para as funções de ordenação.

```
typedef int Tipochave;
typedef struct {
    char caracters[200];
    float real[4];
}Titem;
typedef struct {
    Tipochave Chave;
    Titem itens[10];
}Tregister;
typedef struct {
    Tipochave Verificado;
    Tipochave Verificador;
}Tverifica_radix;
```

No menu para o cenário 2, contém as opções de inicializar um registro com valores aleatórios, selecionar o algoritmo de ordenação desejado e imprimir os registros onde será mostrado na tela os caracteres aleatórios e os valores reais que foram gerados a cada iteração.

```
printf("|-----------------------------------------------|\n");
printf("|    [0] - Inicializar vetor => Registro        |\n");
printf("|    [1] - Selecionar o algoritmo de ordenacao  |\n");
printf("|    [2] - Imprimir Registros                   |\n");
printf("|    [3] - Retornar aos Cenarios                |\n");
printf("|-----------------------------------------------|\n");
```

Figura 2. Menu cenário 2 (Registros).

Em ambos os cenários, para cada método de ordenação foi definido um contador com a variável denominada “cont_if ”, passada como parâmetro para as funções de ordenação para contabilizar o número de comparações, e a variável “cont_move”, contabilizando o número de movimentações (a cada troca de item, contabiliza-se como uma movimentação) realizadas pelo vetor. O tempo de relógio foi contabilizado pela biblioteca disponível na linguagem C, “`#include <time.h>”`, onde o `“clock_t”` analisa o tempo de execução para cada ordenação.

2.3 Cenário 3 - Teste Rápido

O teste rápido foi desenvolvido a fim de fazer uma análise melhor das médias relacionadas aos algoritmos de ordenação, as de tempo de execução e quantidades de comparações e movimentações, para que pudéssemos compará-las melhor. Ele foi feito da seguinte forma, quanto aos cálculos de iterações, pedimos ao usuário que, no menu, na opção cenário 3, digite o tamanho do vetor e o teste. Tal teste será quantas vezes todos os algoritmos vão rodar, para cada iteração teremos contadores que contarão quantas movimentações, comparações e os tempos de execução de cada algoritmo, ou seja, para cada n de teste temos:

```
 for (int i = 0; i < teste; i++)
{
   Array = InicializaArray(n);
   begin = clock();
   Selectionsort_A(Array, n, &cont_if, &cont_move); //o algoritmo da vez
   end = clock();
   double time_spent = (double)(end - begin) / ((CLOCKS_PER_SEC));
   som_if = som_if + cont_if;
   som_move = som_move + cont_move;
   som_time = som_time + time_spent;
   ccont_if = 0;
   cont_move = 0;
   time_spent = 0;
}// Esse trecho é repetido para todos os algoritmos de ordenação n vezes
```

Logo após isso, mandamos os contadores para uma função que vai calcular e preencher em um arquivo a média de tempo, movimentações e comparações de todos os algoritmos de ordenação. Porém, vale ressaltar que, por conta da nossa memória computacional limitada, o código encerra sua execução sem o preencher o arquivo para valores de tamanho do vetor e de teste muito altos. Para contornar essa situação, para estipularmos médias de vetores maiores, utilizamos estimativas com saídas dadas pelos cenários 1 e 2 para esses vetores.

3 Execução Vetores

Para a análise da execução foi compilado cada um dos métodos de ordenação, para os vetores de tamanho 1.000, 5.000, 10.000, 50.000, 100.000, 500.000, 1.000.000 e realizado uma média do número de comparações, movimentações e tempo gasto para o cenário 1. Foram inseridos gráficos a fim de facilitar a visualização das comparações e as tabelas abaixo de cada gráfico apresentam os valores das movimentações, comparações e tempo de relógio realizados, com médias para tamanhos de vetores diferentes. Especificações do computador que foi utilizado para compilação dos algoritmos de ordenação:

Notebook: Acer Aspire 5 A515-54-59X2;
Processador: Intel Core i5-10210U (10ª geração) – 6 MB de cache
4 núcleos e 8 *threads* – de 1.60 GHz até 4.20 GHz;
Memória RAM: 8 GB - DDR4 2400 MHz;
SSD: 512GB;
Sistema Operacional: Windows - 64 bits.

Outrossim, são os gráficos apresentados abaixo que comparam o tempo médio para um determinado vetor aleatória, as comparações, e as movimentações, decidimos deixar as análises de uma entrada em cada gráfico para ficar melhor a visualização dos melhores algoritmos de ordenação e dos piores, isto também ajuda na comparação, todos os gráficos estão em escala logarítmica.

3.1 Vetor mil entradas

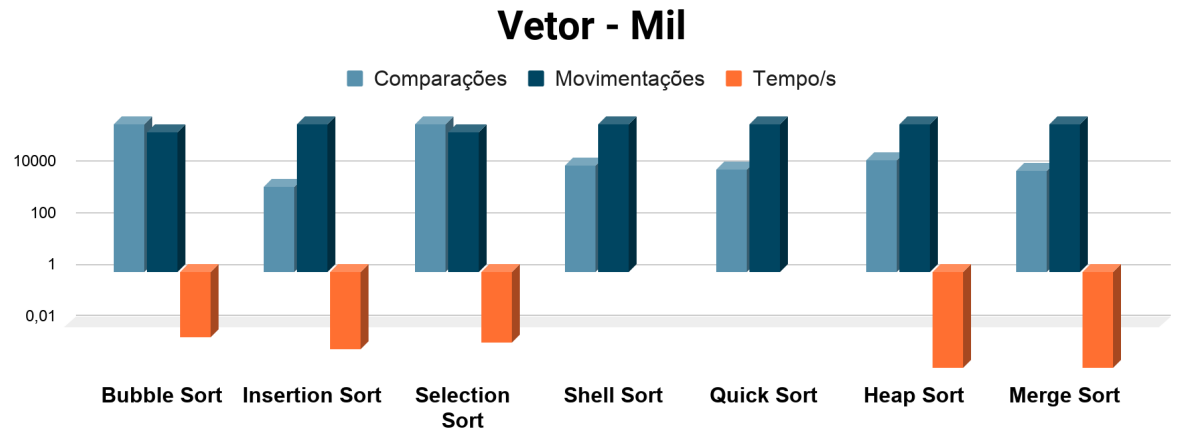

Tabela para o vetor de mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	499500	249975,4	0,0032
Insertion Sort	1993,6	501914,2	0,001
Selection Sort	499500	250967,2	0,002
Shell Sort	14130,2	510587,4	0
Quick Sort	8953,8	513292,4	0
Heap Sort	20912,2	522359,2	0,0002
Merge Sort	8687,6	526656,2	0,0002

Para 1000 entradas no vetor, os algoritmos tiveram um bom desempenho no geral, tirando o Shell e o Quick que em relação às suas médias de tempo foram bem melhores para essa entrada. Logo para um entrada pequena, se vê valores bem distantes entre os algoritmos, mas nada muito discrepante.

3.2 Vetor cinco mil entradas

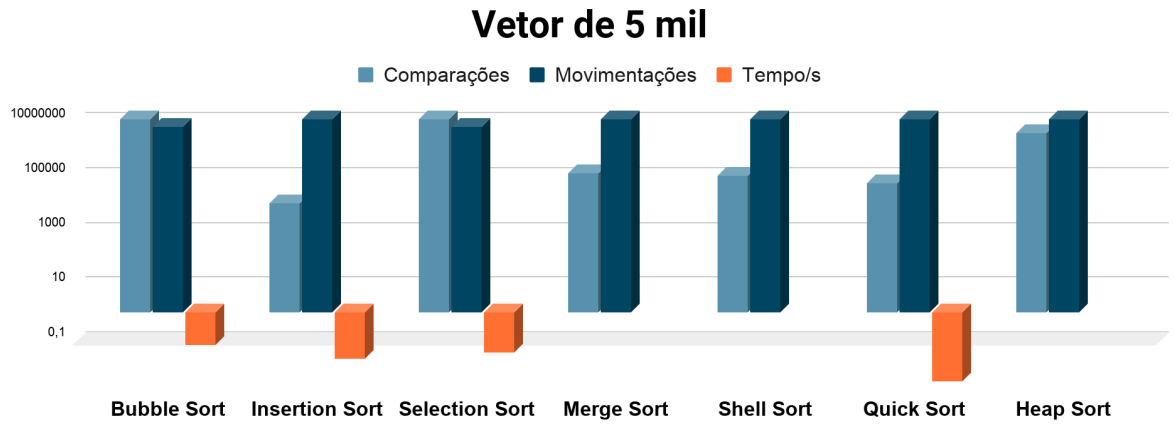

Tabela para o vetor de 5 mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	12497500	6234911,6	0,0656
Insertion Sort	9990,8	12469319,4	0,022
Selection Sort	12497500	6239903,8	0,0374
Merge Sort	133656	12610297,2	0
Shell Sort	102807	12537042,4	0
Quick Sort	57658,8	12553159,4	0,003
Heap Sort	3755799	12610297,2	0

Para o vetor de 5 mil entradas, temos uma análise quase idêntica ao de mil entradas, só o que muda mesmo é o tempo, e os melhores algoritmos já começam se mostrado superiores em relação ao seu tempo.

3.3 Vetor dez mil entradas

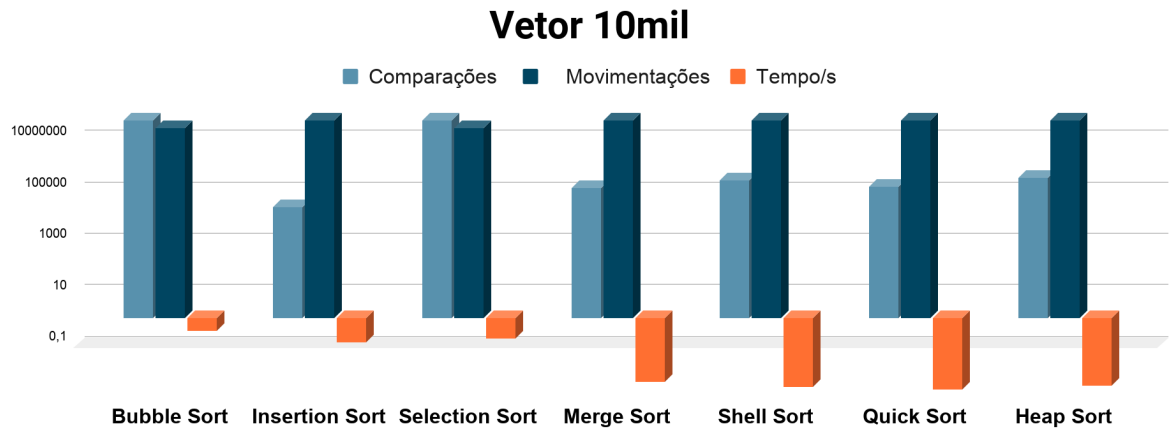

Tabela para o vetor de 10 mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	49995000	24948271,2	0,3066
Insertion Sort	19990	49998349,8	0,109
Selection Sort	49995000	24958261,4	0,1654
Merge Sort	120447,2	50381281,6	0,0032
Shell Sort	240321,4	50163428,2	0,002
Quick Sort	122782	50197981,8	0,0016
Heap Sort	292339	50322158,8	0,0024

Para 10 mil entradas os algoritmos que tiveram mais gastos computacionais foram os Bolha e Seleção, na comparação e movimentação, também tivemos o inserção que teve um alto número de movimentações, mas o que impressiona nesse algoritmo para esse teste é o seu baixo número de comparações, entretanto esse algoritmos mostram ser muito ineficientes e lento em comparação aos outros.

Destarte, temos o Shell, merge, Quick e Heap, que tiveram um número não tão expressivo de comparações, e obtiveram ótimo tempos médios para essa entrada, mas já em relação a suas comparações foram todos maiores que 50 milhões um número muito expressivo, mas mesmo assim em relação tempo foram ótimos.

3.4 Vetor cinquenta mil entradas

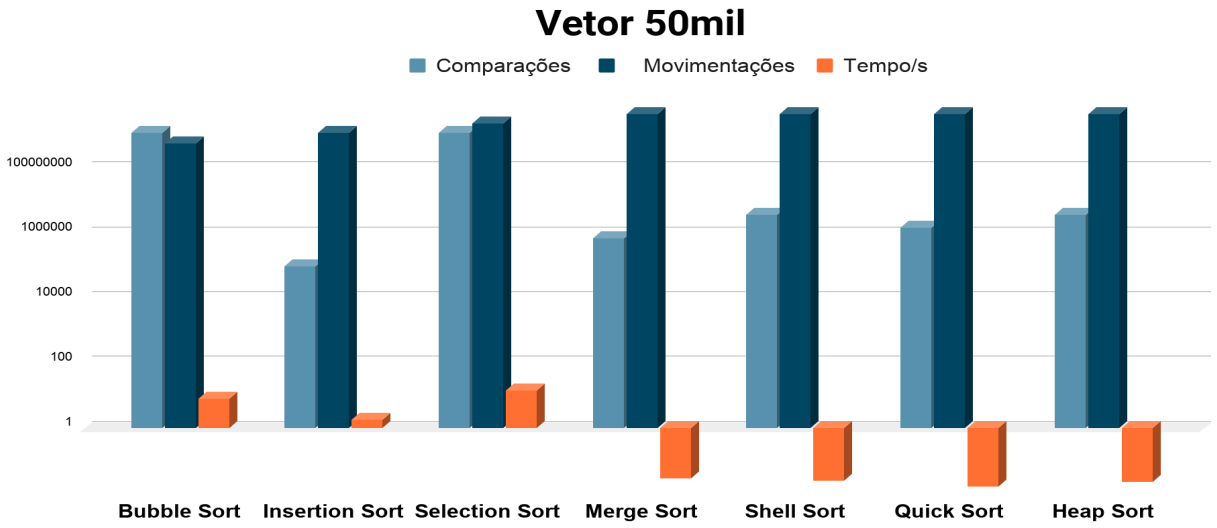

Tabela para o vetor de 50 mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	1249975000	624486108	7.8796
Insertion Sort	99989.4	1248907507	1.9028
Selection Sort	1249975000	2503473630	14.7278
Merge Sort	718144.6	5010639788	0.0282
Shell Sort	3839332.2	5007871139	0.0244
Quick Sort	1516989.4	5008304925	0.0154
Heap Sort	3755799	5009879918	0.0232

Para 50 mil entradas, tivemos outro resultado ruim para o bolha, inserção e seleção, pois além de novamente terem tido piores resultados, para ordenar 50 mil valores inteiros demoram um tempo pequeno, mas expressivo quando se fala

de computação.

Em contrapartida, o Shell, Merge, Quick e Heap tiveram um número muito grande em suas comparações, mas em tempo foram ótimos tendo resultado bem menores que 1s, digo isso, pois todos os outros algoritmos de ordenação fora esses quatro levaram mais que 1s para ordenar o vetor de tamanho 50 mil.

3.5 Vetor cem mil entradas

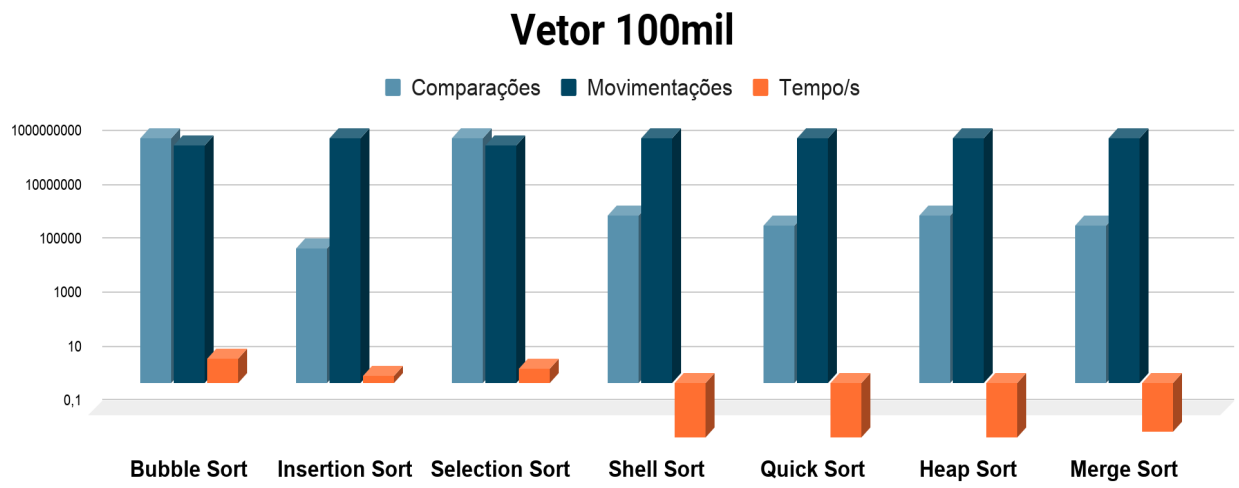

Tabela para o vetor de 100 mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	1249975000	624486108	7,8796
Insertion Sort	99989,4	1248907507	1,9028
Selection Sort	1249975000	624536097,8	3,5022
Shell Sort	1714654,4	1250166442	0,0094
Quick Sort	727023,4	1250368058	0,0092
Heap Sort	1752827,8	1251105490	0,0094
Merge Sort	718144,6	1251460555	0,0156

Já para 100 mil entradas tivemos uma quantidade muito expressiva de tempo, movimentações e comparações para todos os algoritmos, mas em relação a tempo e melhor processamento os Shell, merge, Quick e Heap foram novamente os melhores

3.6 Vetor quinhentas mil entradas

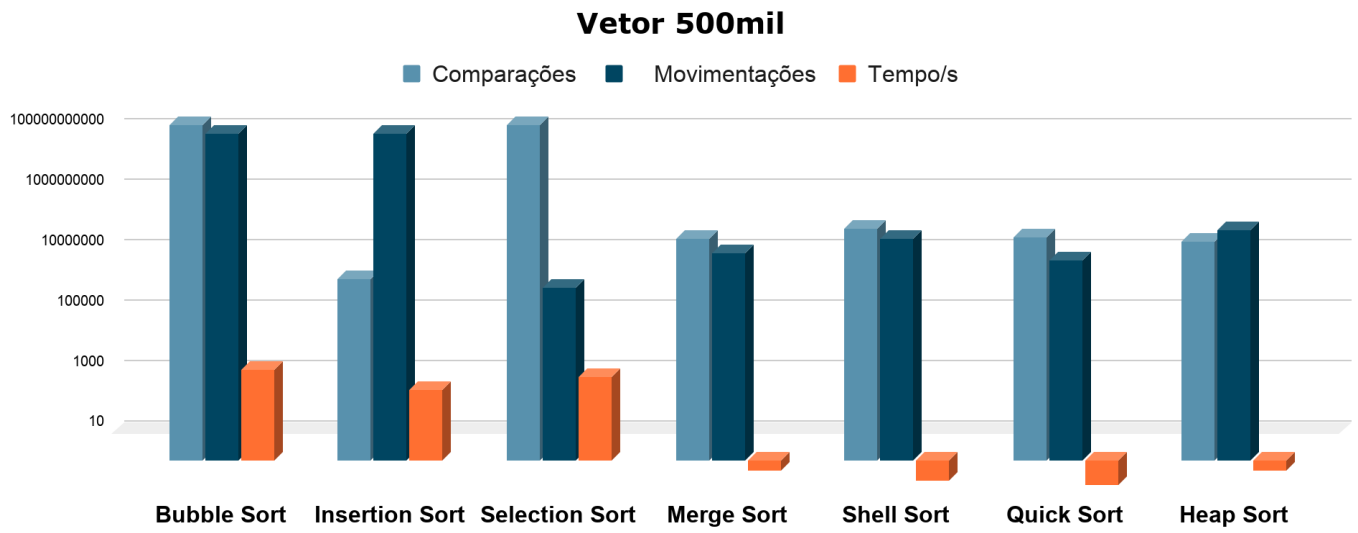

Tabela para o vetor de 500 mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	124999750000	62584919950	929,498
Insertion Sort	999993	62584919950	205,504
Selection Sort	124999750000	499973	583,553
Merge Sort	20384112	6769000	0,4305
Shell Sort	45046899	21520910	0,2128
Quick Sort	22466449	3915613	0,1548
Heap Sort	17595878	39062629	0,432

Para vetores de tamanho 500 mil já sabemos que pelas análises anteriores quais são os melhores e piores algoritmos, e nesse caso temos valores bem discrepantes em relação aos seus tempos, média de comparações e média de movimentação. Dessa forma, o pódio de melhor tempo para esses tamnho vai para o Shell, Quick e Heap, que não há dúvidas que são os melhores algoritmos ordenação analisados aqui.

3.7 Vetor um milhão de entradas

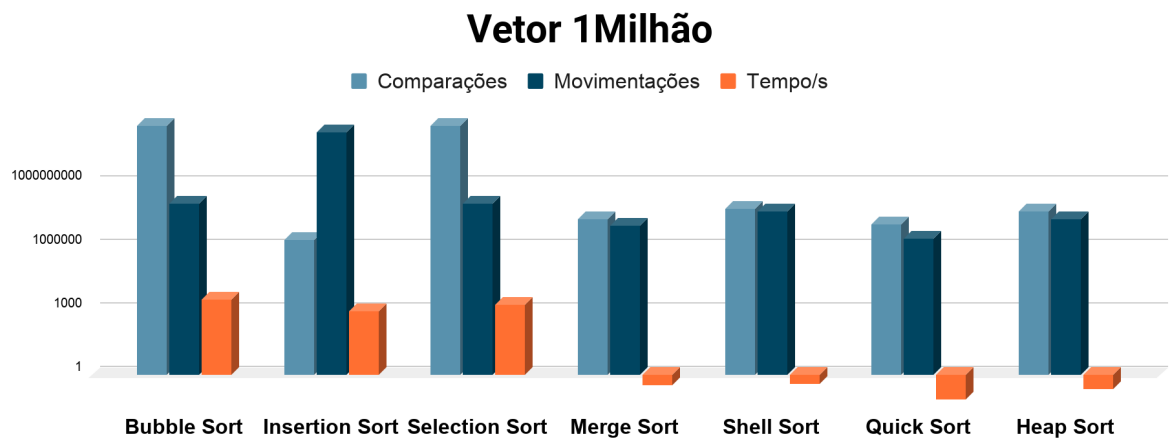

Tabela para o vetor de 1 milhão de entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	499999500	99995300	3203,112
Insertion Sort	1999989	249775606016	892,915
Selection Sort	499999500	99995200	1803,641
Merge Sort	18673484	9278825	0,328
Shell Sort	60605439	48801174	0,375
Quick Sort	12142307	2290358	0,063
Heap Sort	44673181	18803859	0,202

No vetor de tamanho um milhão podemos perceber mais a diferença em relação aos tempos em segundos apresentados, onde algoritmos de ordenação simples, demoram muito tempo para serem ordenados, o bubble sort considerado o pior algoritmo de ordenação para esse vetor de tamanho um milhão apresentou a maior quantidade de comparações realizadas e também o maior tempo levando 53 minutos para ser ordenado.

4 Execução Registros

Na análise dos algoritmos de ordenação para o registro, foram utilizados gráficos e tabelas para representação da comparação do tempo em segundos, movimentações e comparações para registros de tamanho: 1.000, 5.000, 10.000, 50.000, 100.000, 500.000 e 1.000.000, para o cenário 2. Foi utilizado para compilação o mesmo computador da comparação para o cenário 1, especificado anteriormente na documentação.

4.1 Registro mil entradas

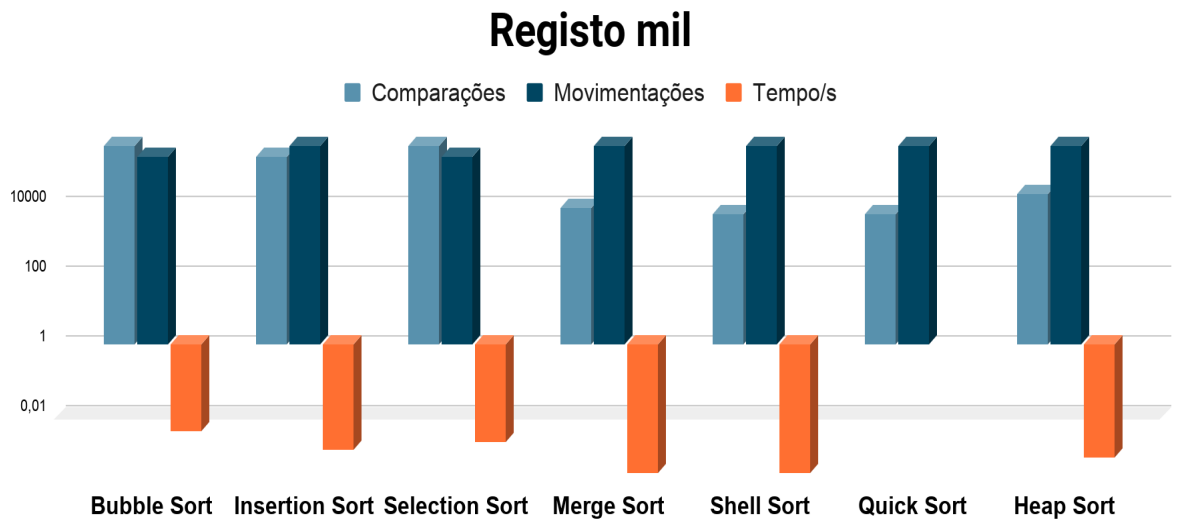

Tabela para o registro de mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	499500	254709	0.00240
Insertion Sort	251581,4	506291	0.00100
Selection Sort	499500	255702	0.00180
Merge Sort	8691,4	531075,6	0
Shell Sort	5457	514947,6	0
Quick Sort	5799,6	517611,8	0
Heap Sort	20968,2	526697,2	0

Para 1000 entradas no Registro, os algoritmos tiveram um bom desempenho no geral, tirando o Shell e o Quick que em relação às suas médias de tempo foram bem melhores para essa entrada. Logo para uma entrada pequena, se vê valores bem distantes entre os algoritmos.

4.2 Registro cinco mil entradas

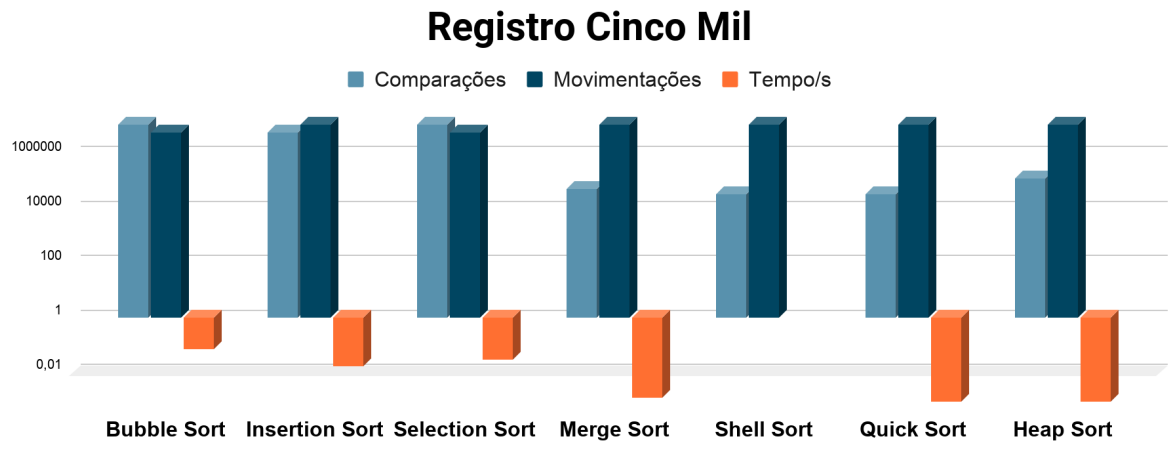

Tabela para o registro de cinco mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	12497500	6324039,2	0,0708
Insertion Sort	6240623	12564664	0,0176
Selection Sort	12497500	6329029,8	0,0304
Merge Sort	55183	12734676,4	0,0012
Shell Sort	35084	12633296,6	0
Quick Sort	34588,4	12649421,2	0,0008
Heap Sort	133664,4	12706543,4	0,0008

No tamanho do registro com cinco mil entradas, temos que o tempo gasto fica sempre abaixo de um segundo onde todos os algoritmos de ordenação apresentam uma boa performance, visto que,o valor de entrada e considera um valor pequeno para ordenação, e pode-se observar também que a quantidade de movimentações e comparações são praticamente as mesmas para o bubble, insertion e selection, já para os demais algoritmos a quantidade de comparações cai significativamente.

4.3 Registro dez mil entradas

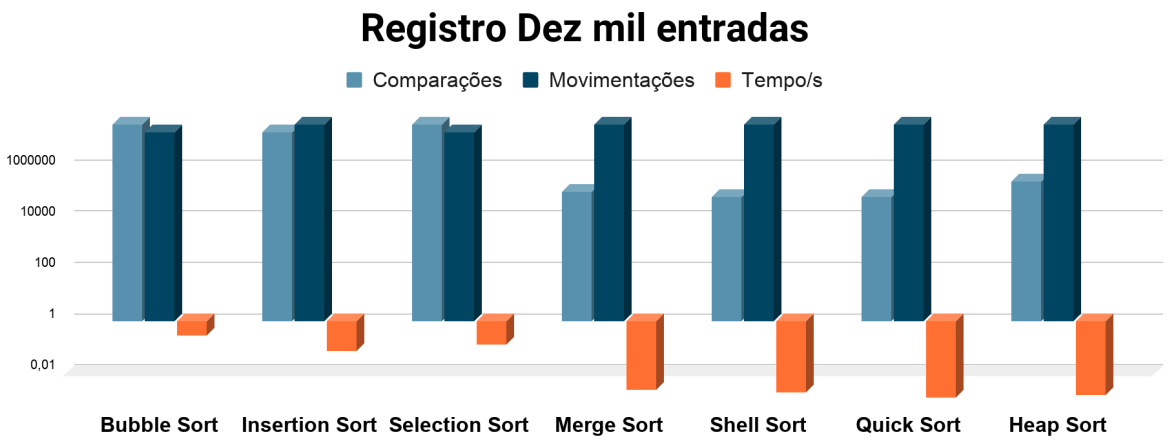

Tabela para o registro de dez mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	49995000	24939490	0,285
Insertion Sort	24947778,2	49887268,2	0,07
Selection Sort	49995000	24949482	0,1222
Merge Sort	120449	50270677,2	0,002
Shell Sort	75243	50050669,8	0,0016
Quick Sort	73630,4	50085152	0,001
Heap Sort	292339,2	50209375	0,0014

Para o registro de dez mil entradas temos que todos os tempos irão ficar abaixo de um segundo e todos os algoritmos apresentam um bom desempenho no geral, visto que, para entradas muito pequenas como o tamanho n de dez mil essas comparações e movimentações não diferenciam tanto de algoritmos mais simples para os mais complexos, então temos o bubble com um bom tempo de execução e o quick que são uns dos melhores métodos de ordenação também apresentou uma performance muito boa.

4.4 Registro cinquenta mil entradas

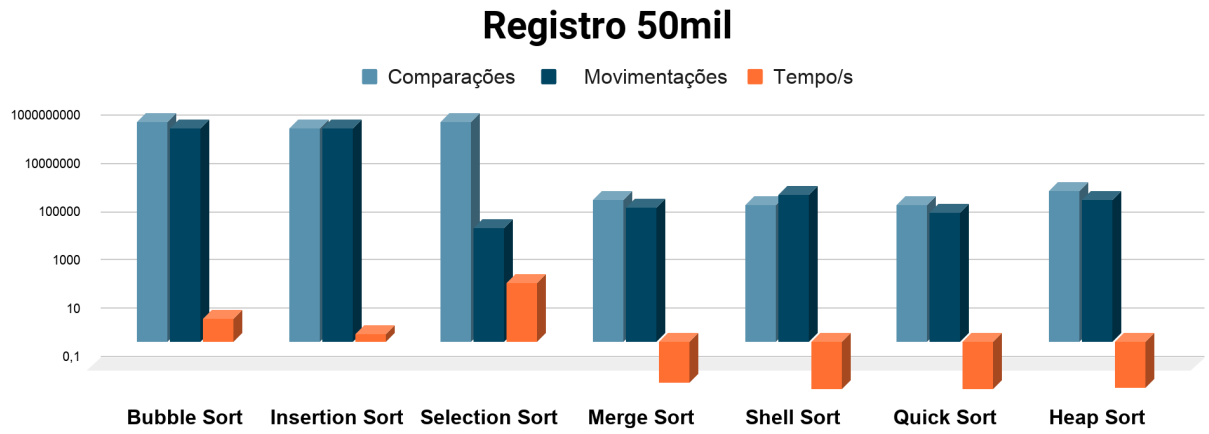

Tabela para o registro de 50 mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	1249975000	625476024	8,347
Insertion Sort	625848123	625798134	1,96
Selection Sort	1249975000	49991	246,535
Merge Sort	718318	363441	0,02
Shell Sort	455719	1228655	0,01
Quick Sort	421634	200187	0,01
Heap Sort	1753481	737583	0,012

Para os registros de tamanho cinquenta mil, ao analisar a tabela onde mostra as comparações, movimentações e o tempo gasto em segundos, podemos concluir que os algoritmos de ordenação mais simples, que são o bubble, Insertion e selection, gastam um tempo bem significativo em relação aos demais, onde o selection pode gastar 4 minutos para realizar a ordenação de um registro neste tamanho.

4.5 Registro cem mil entradas

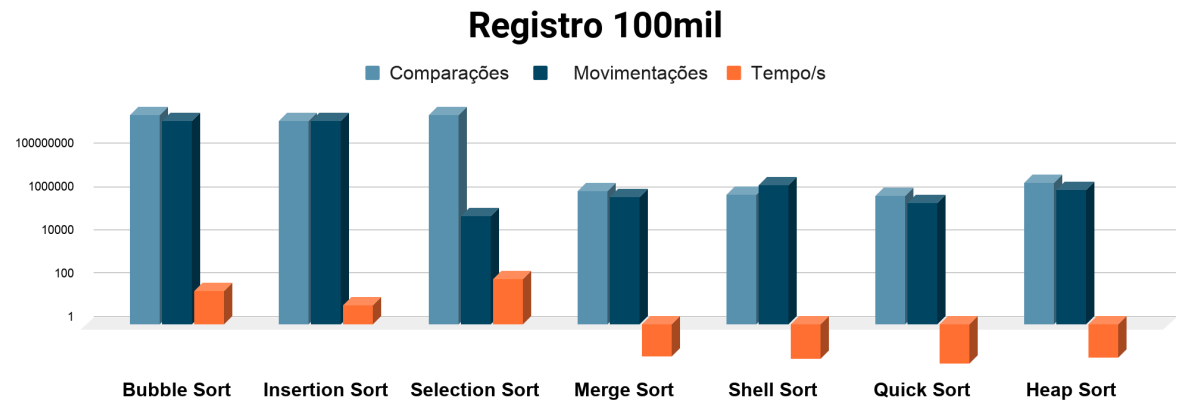

Tabela para o registro de 100 mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	4999950000	2506044511	33,895
Insertion Sort	2497249280	2497149291	7,87
Selection Sort	4999950000	99987	126,798
Merge Sort	1536246	776121	0,032
Shell Sort	967146	2744420	0,025
Quick Sort	911760	439011	0,015
Heap Sort	3755831	1574969	0,031

Registro de cem mil entradas, analisamos também mais uma vez a participação significativa das comparações dos algoritmos bubble and selection e seus tempos elevados, e para o menor tempo e quantidade de movimentações temos um algoritmo sofisticado o quick sort.

4.6 Registro quinhentas mil entradas

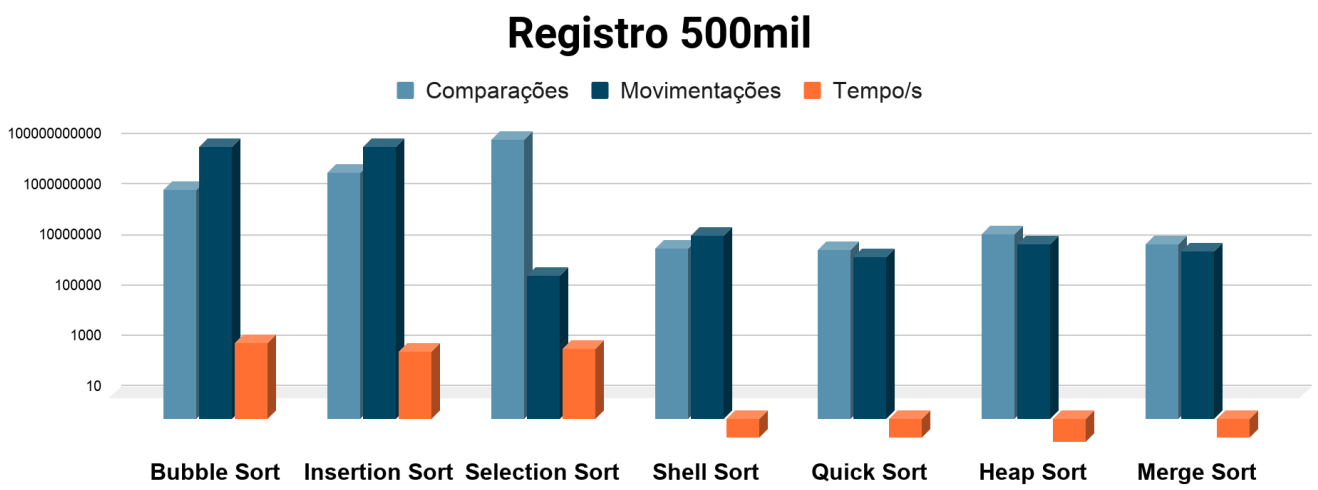

Tabela para o registro de quinhentas mil entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	1249997500	62584919950	1141,316
Insertion Sort	6247448445	62471948449	478,959
Selection Sort	124999750000	499973	613,499
Shell Sort	5601426	20297065	0,19
Quick Sort	5397300	2661321	0,197
Heap Sort	21993878	9300223	0,124
Merge Sort	8837164	4447264	0,188

Analisando o gráfico e a tabela apresentada nota-se que quanto maior a entrada mais perceptível fica a diferença em relação ao tempo e as quantidades de comparações e movimentações realizadas por cada algoritmo, onde shell, quick, heap e merge sempre tem um melhor desempenho em relação aos demais em todas as análises realizadas nessa documentação.

4.7 Registro um milhão de entradas

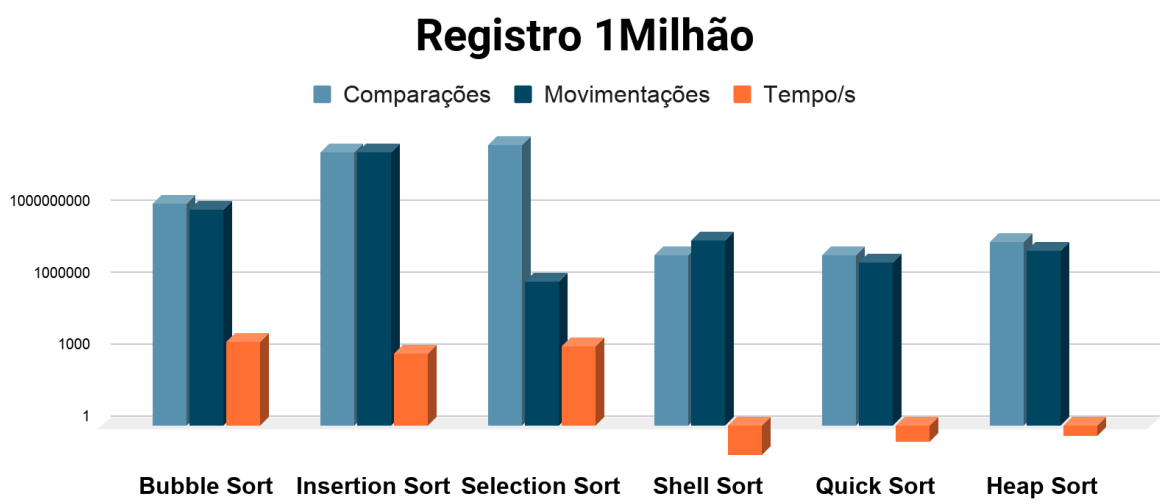

Tabela para o registro de um milhão de entradas:

	Comparações	Movimentações	Tempo/s
Bubble Sort	1783293664	912196378	2909,338
Insertion Sort	250124019591	250123019598	984,511
Selection Sort	499999500000	999952	1916,184
Shell Sort	11804265	48484352	0,053
Quick Sort	11514404	5709779	0,199
Heap Sort	45846629	19047720	0,367

Para entradas de 1 milhão, os algoritmos sofisticados fizeram ótimas ordenações, entretanto tivemos alguns impasses em relação a memória, já que para o Mergesort só conseguimos fazer ele rodar para um tamanho de entrada menor que 500 mil, e a média de comparações dele para essa entrada foi de 8837164, já a sua média de movimentações foi de 4447264, e sua média de tempo foi de 0.15000, um tempo muito baixo por sinal.

Outrossim, os outros algoritmos conseguiram rodar para 1 milhão de vezes, o bolha, inserção, e seleção, tiveram péssimos tempos para ordenar os registros de 1 milhão. Já Shell, Quick e o Heap, foram melhores em tempo, não só nisso como também, em suas movimentações e suas comparações.

5 Conclusão

Neste trabalho prático foram analisados 8 algoritmos de ordenação, durante esse processo verificamos a quantidade de tempo, comparações e movimentações que foram feitas em cada um desses códigos. Dessa forma, com essa análise feita podemos confirmar quais algoritmos são mais eficientes para serem usados para determinados casos.

No quesito desempenho, tivemos os algoritmos de Bolha, inserção e seleção, esses foram os piores, pois para todos os casos de vetores gerados aleatoriamente, isto também para registros, os algoritmos se mostram ineficientes, pois o número de trocas e de comparações é muito grande, esse algoritmos tem complexidade assintótica O(n^2) , ou seja, não são bons algoritmos para serem usados, isto foi dito em sala de aula e comprovado com o trabalho prático, pois os gráficos mostram uma quantidade excessiva de tempo para terminar a ordenação. Logo, é perceptível se tratando de eficiência que esses três não podem ser usados para ordenar algo em algum sistema pela sua ineficiência.

Outrossim, são os algoritmos que tiveram ótimos resultados para determinados tamanho aleatórios de vetores, os códigos foram Shellsort, Quicksort, Heapsort e Radixsort, esses algoritmos tiveram desempenho bons durante a análise, principalmente pelo seus tempos que foram ótimos.

Em suma, neste trabalho prático o grupo aprendeu quais são os melhores algoritmos de ordenação para serem implementados e usados, pois como hoje em dia os programadores trabalham com grande quantidade de dados precisamos de algoritmos sofisticados para resolverem esses problemas de ordenação para um n muito grande.