Documentação do Trabalho Prático II da disciplina de Estruturas de Dados

Lucca Silva Medeiros - 2019054773

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

luccasilva@dcc.ufmg.br

1. Introdução

Esta documentação lida com o problema da exploração do Composto Z, material essencial para desenvolver as tecnologias capazes de restaurar as condições de vida em uma Terra pós-apocalíptica. Após o sucesso da fase exploração, tem-se início o processo de organização das bases de dados. Foram implementados 5 algoritmos de ordenação diferentes, cujas performances foram medidas e discutidas nesse relatório. Esta documentação tem como objetivo detalhar, analisar e discutir os métodos implementados, sendo eles o InsertionSort, MergeSort, QuickSort, NotRecursiveQuickSort e o ShellSort. Além disso, apresentar as instruções de compilação e execução do programa.

Esta documentação está separada em seções. Na segunda é apresentado a implementação do programa no geral e de cada algoritmo utilizado. Na terceira as instruções de compilação e execução do programa. Na quarta a discussão dos resultados e por último a conclusão do Trabalho.

2. Implementação

O código está organizado nas funções listadas a seguir:

- readFile: responsável por ler o arquivo de entrada e armazenar no vetor “dataPlanet” o nome e a distância dos planetas presentes na base de dados. Os dados foram organizados no struct Planet, que possui uma string para armazenar o nome e um inteiro para armazenar a distância.

- printPlanets: responsável por imprimir os 7 planetas mais distantes após ordenação dos dados.

- deletePlanets: responsável por apagar o vetor “dataPlanet” no final da execução.

- insetionSort: responsável por ordenar o vetor decrescentemente utilizando o algoritmo insertionSort.

- merge: responsável por executar o passo “merge” do algoritmo mergeSort.

- mergeSortMethod: responsável por ordenar o vetor decrescentemente utilizando o algoritmo mergeSort.

- mergeSort: função auxiliar para chamada da função mergeSortMethod, visando padronizar as chamadas dos algoritmos de ordenação.

- Partition: responsável por executar o passo da partição do algoritmo quickSort.

- Order: responsável por ordenar o vetor decrescentemente utilizando o algoritmo quickSort.

- quickSort: função auxiliar para chamada da função Order, visando padronizar as chamadas dos algoritmos de ordenação.

- notRecQuickSort: responsável por ordenar o vetor decrescentemente utilizando o algoritmo quickSort, porém com a melhoria da pilha auxiliar.

- shellSort: responsável por ordenar o vetor decrescentemente utilizando o algoritmo shellSort.

O código também conta com um objeto auxiliar:

- **Stack**: estrutura de dado do tipo Pilha, responsável por armazenar as partições do vetor geradas pela ordenação do quickSort. Essas partições são representadas pelo struct do tipo Item, com um inteiro para representar o extremo esquerdo e outro inteiro para representar o extremo direito da partição.

Todos o objeto Stack e o struct Planet foram declarados separadamente nos respectivos arquivos.h presentes na pasta include. Os métodos de cada objeto estão implementados separadamente nos respectivos arquivos.cc presentes na pasta src. Ao compilar os arquivos.o de cada objeto estarão disponíveis na pasta obj.

A função main está no arquivo Main.cc, presente na pasta src. Ela é responsável por ler os argumentos passados pelo usuário (arquivo de dados e quantidade de entradas) e realizar a ordenação decrescente dos planetas lidos. Após ordenação o programa imprime os 7 planetas mais distantes. Por *default* o programa realiza a ordenação utilizando o método quickSort.

Caso seja necessário trocar o algoritmo de ordenação basta comentar a chamada da função quickSort e retirar o comentário da função desejada, por exemplo:

- Configuração *default*, o programa realizará a ordenação pelo quickSort.

```
    //insertionSort(dataPlanet,SIZE);
    //mergeSort(dataPlanet,SIZE);
    quickSort(dataPlanet,SIZE);
    //notRecQuickSort(dataPlanet,SIZE);
    //shellSort(dataPlanet,SIZE);
```

- Agora o programa realizará a ordenação pelo mergeSort.

```
    //insertionSort(dataPlanet,SIZE);
    mergeSort(dataPlanet,SIZE);
    //quickSort(dataPlanet,SIZE);
    //notRecQuickSort(dataPlanet,SIZE);
    //shellSort(dataPlanet,SIZE);
```

A configuração utilizada para testar o programa está especificada abaixo:

– **Sistema Operacional**: Windows 10 Home

– **Linguagem de programação implementada:** C++

– **Compilador utilizado:** mingw / G++

– **Processador**: Inter(R) Core(TM) i7-8750H CPU @2.20GHz 2.21GHz

– **Quantidade de memória RAM:** 16,0 GB

2.2 Implementação dos algoritmos de ordenação

1- Insertion Sort

O insertion sort é um algoritmo de ordenação que consiste em ordenar os itens inserindo-os na posição correspondente do vetor, semelhante a um jogo de baralho.

O insertion sort no seu melhor caso, ou seja, quando todas as entradas já estão devidamente ordenadas, possui uma complexidade O(n), porém nos seus demais casos, sua complexidade é $O(n^2)$, tanto para número de comparações quanto para número de movimentações. Sua complexidade de espaço é O(1) e é um algoritmo de ordenação estável.

2- Merge Sort

O merge sort é um algoritmo de ordenação do tipo dividir para constar. Sua ideia consiste basicamente em dividir o vetor através de chamadas recursivas e durante o processo de merge, ou seja, junção das partes divididas, realizar a ordenação.

Sua complexidade tanto no pior caso, quanto no caso médio, quanto no melhor caso é O(n log n), sua complexidade de espaço é O(n) e é um algoritmo de ordenação estável.

3- Quick Sort

O quick sort é um algoritmo que adota a estratégia de divisão e conquista, e ele possui basicamente 3 passos:

- Escolha do pivô;

- Partição, ou seja, rearranjar os elementos do vetor tendo como referência o pivô;

- Ordenar recursivamente os elementos;

Sua complexidade no melhor caso e no caso médio é O(n log N) porém existe um caso específico, seu pior caso, onde a complexidade deste algoritmo será $O(n^2)$. Sua complexidade de espaço é O(n) e o algoritmo não é estável.

Em relação ao quick sort, por ser um algoritmo recursivo, o mesmo acaba gastando muita memória na pilha de recursão, portanto, a melhoria proposta foi a implementação do quick sort não recursivo. Essa implementação se deu através da criação de uma estrutura de dados pilha para controlar qual parte do vetor já foi ordenada e qual ainda falta ser.

Essa melhoria não altera a complexidade do algoritmo, porém faz com que ele ocupe mais memória no geral, mas economiza na pilha de recursão.

4- Shell Sort

O shell sort é similar ao insertion sort, porém ao invés de comparar o dado atual com o dado da frente a comparação é feita em saltos, diminuindo assim o tempo gasto para ordenar o vetor. O método para escolher os passos fica a critério do usuário, e esse passo impacta fortemente no desempenho do algoritimo.

O método de saltos escolhido nessa implementação foi o seguinte:

Para h sendo o salto;

E s sendo a posição do vetor de dados;

h(s) = 1, se s = 1 e h(s) = 3h(s - 1) + 1, se s > 1

A sequência gerada por esse método de salto é difícil de ser batida por mais de 20% em eficiência.

O shell sort possui uma complexidade de comparações de O(n log n), se h(s) = 1 e de $O(n(\ln n)^2)$, se h(s) = 3h(s - 1) + 1. Já sua complexidade de espaço é O(1) e é um algoritmo de ordenação instável.

3. Instruções de compilação e execução

1- Baixe o arquivo.zip e extraia a pasta tp2;

2- A pasta deverá conter os seguintes arquivos;

/ bin – run.out, dados_aleatorio.txt, dados_crescente.txt e dados_decrescente.txt

/ include – Planet.h e Stack.h

/ obj

/ src –Main.cc e Stack.cc

/ Makefile

3- Utilizando um terminal, abra o diretório da pasta tp2 e execute o arquivo [Makefile] utilizando o seguinte comando: < make >;

4- Com esse comando o Makefile irá compilar o programa e deverá criar com sucesso os objetos: ./obj/Main.o ./obj/Stack.o.

5- Utilizando um terminal, abra o diretório da pasta bin e execute o arquivo [run.out] utilizando o seguinte comando: < run.out argumento1.txt número >. O argumento 1 deverá ser o arquivo.txt referente ao arquivo dos dados que se deseja ordenar e o argumento 2 deverá ser número de planetas que serão ordenados.

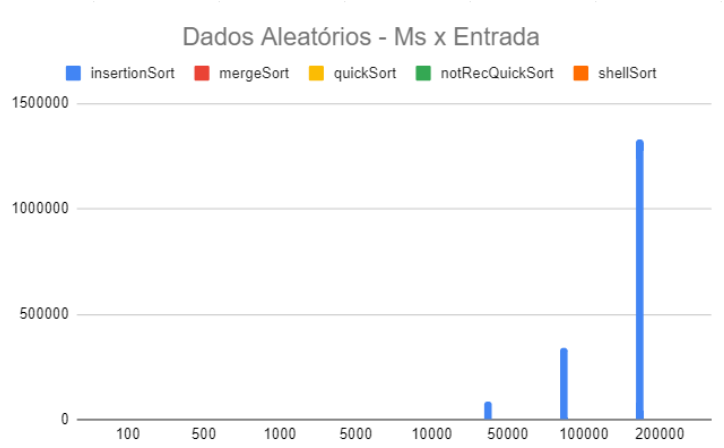
6- Com isso o programa irá executar e a saída esperada aparecerá no terminal.

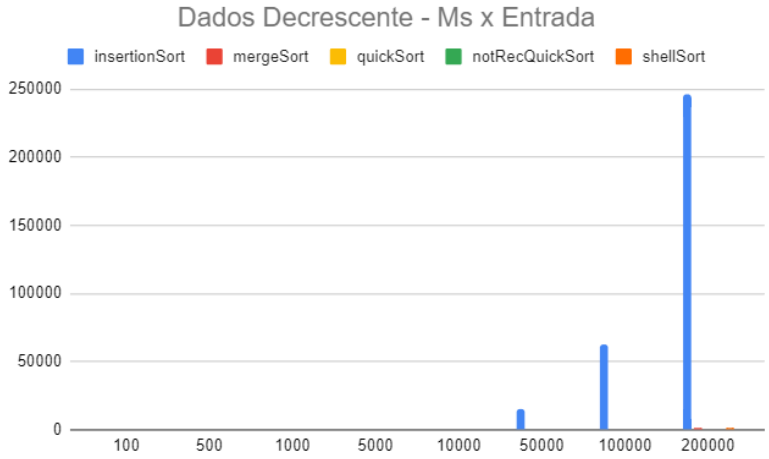
4. Análise dos resultados

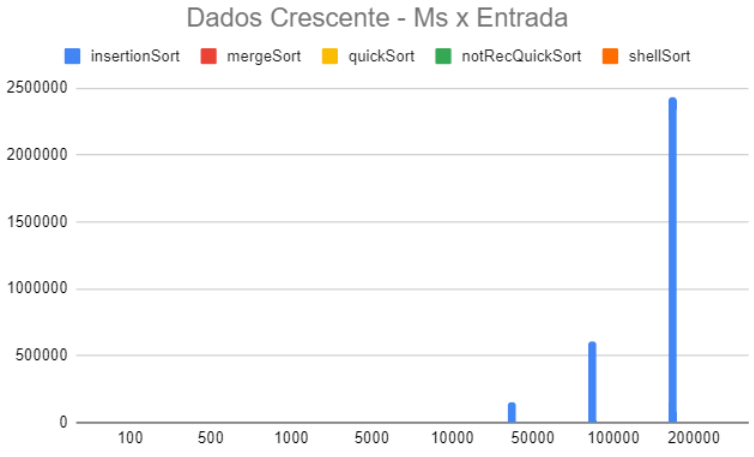
O tempo de execução de cada ordenação foi medido em milissegundos utilizando a biblioteca <time.h>. O resultado da performance final na ordenação dos dados aleatórios, dados crescentes e dados decrescentes de cada algoritmo está presente na tabela abaixo:

dados_aleatorio					
Entradas	insertionSort	mergeSort	quickSort	notRecQuickSort	shellSort
100	7	8	6	7	6
500	15	10	7	7	8
1000	41	13	10	10	11
5000	854	33	16	16	29
10000	3408	62	28	28	54
50000	85875	321	114	120	321
100000	338513	671	233	241	709
200000	1332802	1392	464	493	1564
dados_decrescente					
Entradas	insertionSort	mergeSort	quickSort	notRecQuickSort	shellSort
100	7	7	7	6	7
500	8	9	6	9	9
1000	14	12	8	7	8
5000	171	31	13	14	25
10000	637	61	21	23	46
50000	15486	319	78	84	269
100000	62164	658	147	161	599
200000	246632	1390	321	323	1407
dados_crescente					
Entradas	insertionSort	mergeSort	quickSort	notRecQuickSort	shellSort
100	16	7	5	7	6
500	41	8	8	7	8
1000	76	11	9	7	9
5000	1526	33	14	13	23
10000	6148	63	19	20	48
50000	151004	316	69	75	275
100000	606148	656	135	146	578
200000	2433372	1392	276	297	1366

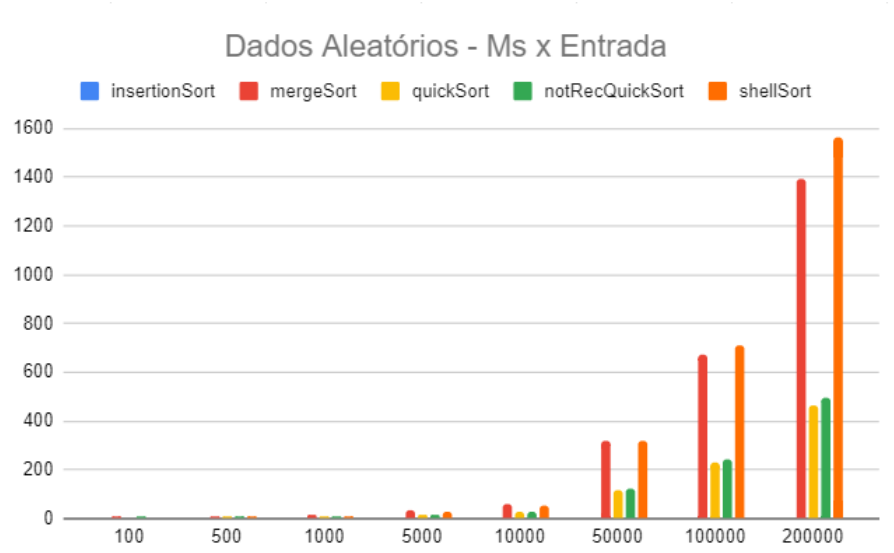
Para melhor visualização e para facilitar a comparação entre os algoritmos foram plotados os seguintes gráficos:

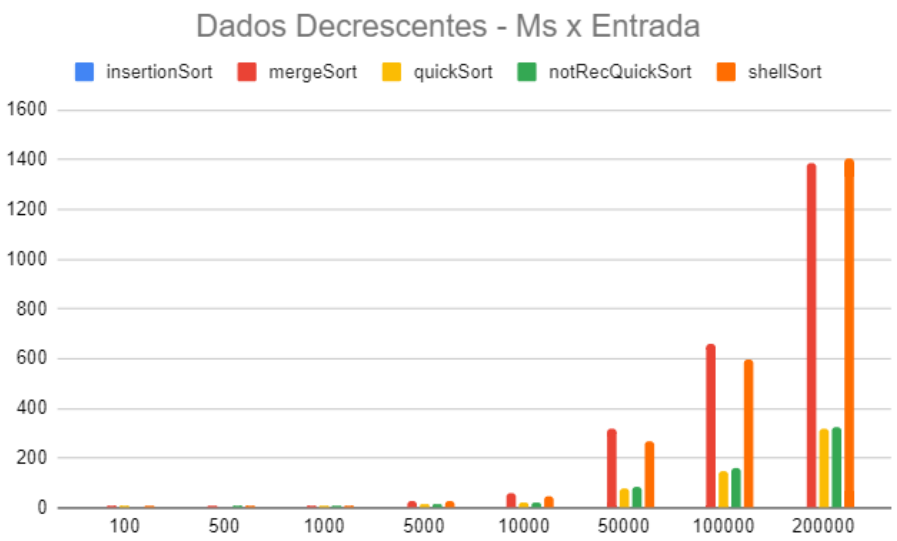


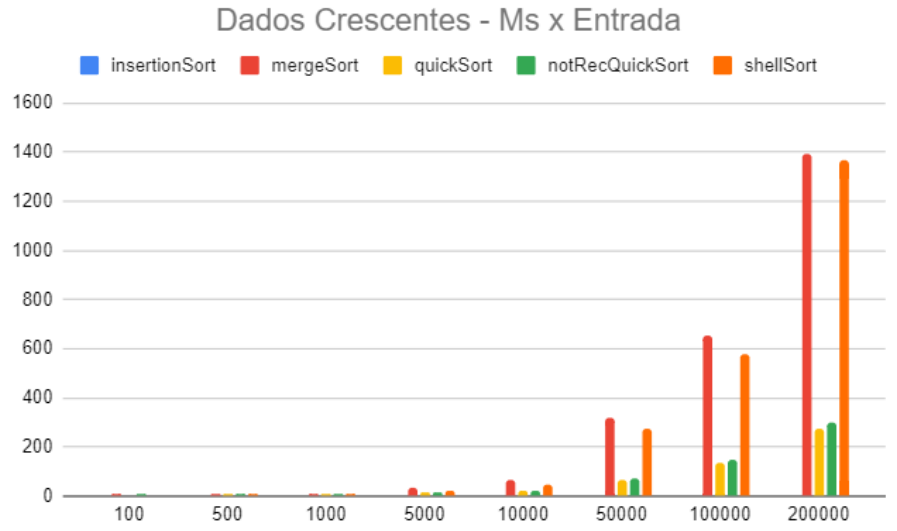




É possível observar de início que a performance do insertionSort é muito inferior à performance dos outros algoritmos. Seu tempo de execução foi tão superior que fica impossível visualizar os outros dados. Portanto foram plotados outros três gráficos, comparando agora apenas os algoritmos restantes:


Dados Aleatórios - Ms x Entrada
insertionSort
mergeSort
quickSort
notRecQuickSort
shellSort
1600
1400
1200
1000
800
600
400
200
0
100
500
1000
5000
10000
50000
100000
200000


Dados Decrescentes - Ms x Entrada
insertionSort
mergeSort
quickSort
notRecQuickSort
shellSort
1600
1400
1200
1000
800
600
400
200
0
100
500
1000
5000
10000
50000
100000
200000

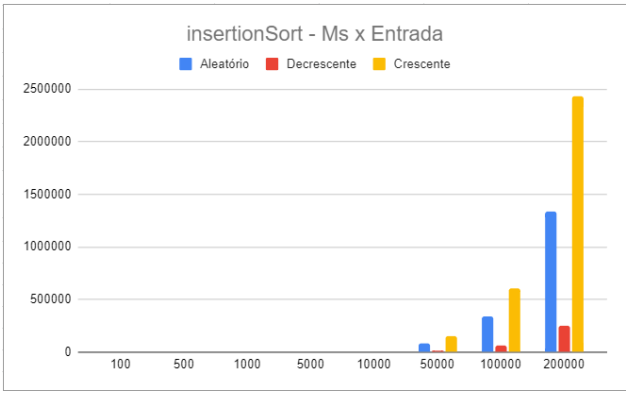

Dados Crescentes - Ms x Entrada
insertionSort
mergeSort
quickSort
notRecQuickSort
shellSort
1600
1400
1200
1000
800
600
400
200
0
100
500
1000
5000
10000
50000
100000
200000

É possível observar através desses gráficos que o quicksort foi o método mais eficiente independentemente da forma em que os dados de entrada estavam ordenados. É possível observar também que a melhoria implementada não impacta drasticamente na performance do algoritmo, uma vez que apresentou resultado muito semelhante.

Já sobre o shellsort e o mergesort, ambos apresentaram resultados bastante semelhantes, porém com algumas observações. O shellsort se comporta melhor quando os dados de entrada estão ordenados de forma crescente, e o mergesort se comporta melhor quando os dados de entrada estão ordenados de forma aleatória.

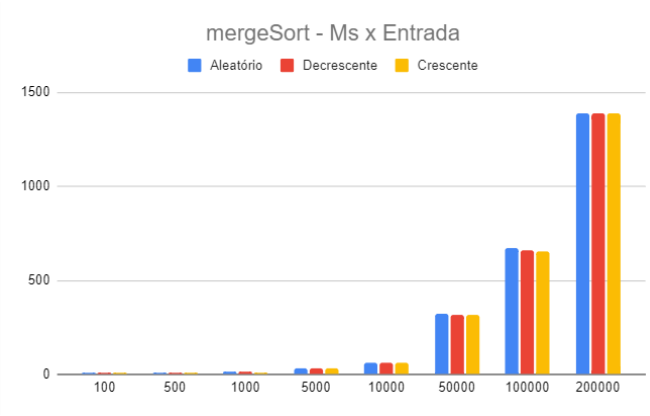
Analisando a performance individual de cada algoritmo temos:

insertionSort			
Entradas	Aleatório	Decrescente	Crescente
100	7	7	16
500	15	8	41
1000	41	14	76
5000	854	171	1526
10000	3408	637	6148
50000	85875	15486	151004
100000	338513	62164	606148
200000	1332802	246632	2433372



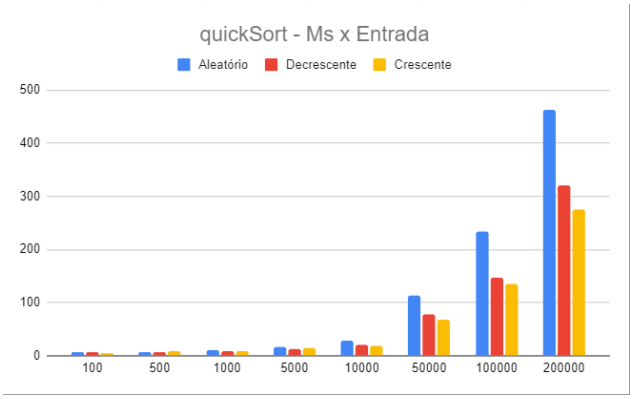
A performance do insertionsort é drasticamente influenciada pela forma de ordenação dos dados de entrada. Após os testes fica nítida a diferença entre seu melhor caso (quando os dados já estão ordenados decrescentemente) e o seu pior caso (quando os dados estão ordenados de forma crescente).

mergeSort			
Entradas	Aleatório	Decrescente	Crescente
100	8	7	7
500	10	9	8
1000	13	12	11
5000	33	31	33
10000	62	61	63
50000	321	319	316
100000	671	658	656
200000	1392	1390	1392

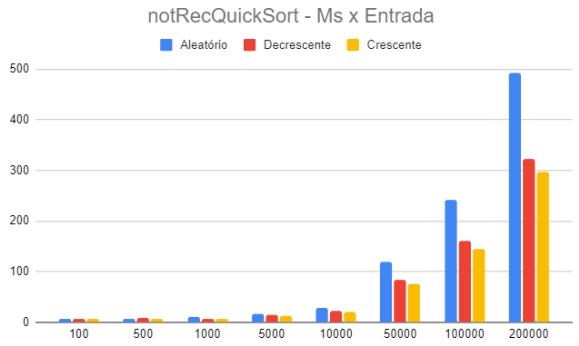


A performance do mergesort não é drasticamente influenciada pela forma de ordenação dos dados de entrada.

quickSort			
Entradas	Aleatório	Decrescente	Crescente
100	6	7	5
500	7	6	8
1000	10	8	9
5000	16	13	14
10000	28	21	19
50000	114	78	69
100000	233	147	135
200000	464	321	276

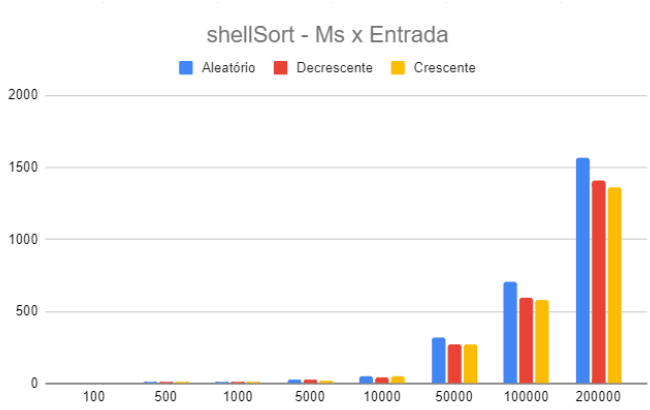


notRecQuickSort			
Entradas	Aleatório	Decrescente	Crescente
100	7	6	7
500	7	9	7
1000	10	7	7
5000	16	14	13
10000	28	23	20
50000	120	84	75
100000	241	161	146
200000	493	323	297



A performance do quicksort e do quicksort não recursivo é bastante semelhante. Ambas não são drasticamente influenciadas pela forma de ordenação dos dados de entrada. Porém quando os dados estão ordenados de forma aleatória a performance é inferior aos demais casos.

shellSort			
Entradas	Aleatório	Decrescente	Crescente
100	6	7	6
500	8	9	8
1000	11	8	9
5000	29	25	23
10000	54	46	48
50000	321	269	275
100000	709	599	578
200000	1564	1407	1366



A performance do shellsort não é drasticamente influenciada pela forma de ordenação dos dados de entrada.

5. Conclusão

Este trabalho lidou com o problema da organização de dados da missão de extração do Composto Z, o qual a abordagem utilizada para resolução foi a implementação de um programa em C++ capaz de ordenar as bases de dados utilizando 5 algoritmos de ordenação diferentes. Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionadas à disciplina Estrutura de Dados: Algoritmos de Ordenação, e ver na prática sua implementação e execução para a solução de um problema palpável. Além de relembrar conceitos relacionados à disciplina de Programação e Desenvolvimento de Software II como programação orientada à Objetos, boas práticas de programação e modularização de código.

6. Referências

- Chaimowicz, L. and Prates, R. (2021). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.
- https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/shellsort.pdf