UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Documentação Trabalho Prático 2
Estruturas de Dados - DCC205 - TF

Extração - Z
Parte 2

Luiz Gustavo Almeida de Oliveira
Matrícula: 2019083781
luizorst@hotmail.com

7 de março de 2021

Sumário

1 Introdução

1.1 Contextualização

Com o sucesso da missão para extração do Composto Z, aquele que poderá ser usado para realizar um processo de terraformação e retornar a Terra para um estado de ampla biodiversidade, os cientistas do Instituto de Ciências Exoplanetárias agora tem um novo desafio, trazer todo o recurso coletado de volta.

Os robôs com recursos estão espalhados por diversos sistemas planetários, mesmo fora da Via Láctea, então foi decidido que as bases priorizadas serão aquelas mais distantes, que requerem mais saltos hiperespaciais para serem alcançadas. Foram propostos diversas formas para organizar as informações na ordem correta de prioridade.

A partir do debate com os diversos times de engenharia, foram implementados os seguintes métodos:

- **Insertion Sort**
  Um método vantajoso para arquivos quasi ordenados e uma boa forma quando se precisa adicionar poucos itens a um arquivo já ordenado, e é estável, porém tem alto custo de movimentações de elementos no vetor.
- **Heap Sort**
  Um elegante método e com comportamento consistente, independentemente da entrada. Por outro lado, tem uma sequência de operações internas bastante complexa quando comparada com outros métodos e não é um método estável
- **Quick Sort** e **Quick Sort Modificado**
  Aqueles que obtiveram mais votos do time de engenharia, amplemente usado em outras aplicações. Esses métodos são bastante eficientes, necessitando de uma pequena pilha de memória auxiliar. Por outro lado, o método não é estável e tem um pior caso quadrático, o que é preocupante. A sua versão modificada busca resolver um pouco dessas desvantagens
- **Shell Sort**
  Esse método é uma extensão do *Insertion Sort* que compara elementos que estão distantes ao invés de adjacentes, ele melhora sua complexidade fazendo uso da vantagem que é usar o *Insertion Sort* em vetores parcialmente ordenados, além de usar apenas uma pequena memória adicional constante, independente do tamanho da entrada.

1.2 Metodologia de Comparação

Durante a implementação e para a realização de testes, uma máquina com as seguintes configurações de software e hardware foi utilizada:

Sistema Operacional	Manjaro 20.2.1 Nibia
Kernel	x86-64 Linux 5.4.89-1-MANJARO
CPU	Intel Core i5-7200U @ 4x 3.1GHz
RAM	7838 MiB
GPU	GeForce 940MX
Compilador	g++ 10.2.0
Linguagem	C++
C++ Standard	14

Cada um dos algoritmos foi executado 3 vezes por entrada e então a média do tempo nessas execuções foi tomadada para estudo.

O header ***chrono***[1], da biblioteca *date and time*, foi utilizado para medir o tempo de execução dos diversos algoritmos da seguinte forma:

```
#include <chrono>

void f() {...}

int main()
{
    auto t1 = std::chrono::high_resolution_clock::now();
    f();
    auto t2 = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double, std::milli> duracao_total_ms = t2 - t1;
}
```

2 Instruções

2.1 Compilação e Execução

Pré-requisitos:

- Compilador moderno de C++(e.g: g++ 10.x.x)
- Programa *GNU Make* instalado (altamente recomendado)

Primeiro, deve-se extrair os arquivos em um diretório onde deseja que os mesmos fiquem armazenados. As instruções para gerar o binário com ou sem o programa *GNU Make* estão nas próximas subseções, mas considerando que ele foi gerado com sucesso, ele deve ser utilizado da seguinte maneira, assumindo que você está na mesma pasta que o executável e que o arquivo contendo os dados corretos está no mesmo diretório também:

```
./meu_executavel [Caminho arq. com dados] [Linhas a serem lidas]
```

O executável gerado pelo Makefile tem o nome padrão run.out, então, exemplificando novamente, assumindo que o arquivo dados_crescente.txt está no mesmo diretório que o executável, basta executar, sendo que o primeiro argumento é o nome do arquivo contendo o mapa e o segundo é o número de linhas que devem ser consideradas para extração de dados no arquivo:

```
./run.out dados_crescente.txt 100
```

2.1.1 Gerando o executável sem o programa *GNU Make*

Caso o programa *GNU Make* não esteja instalado, será necessário entrar na pasta *Projeto* e compilar manualmente os arquivos, fazendo a inclusão dos arquivos na pasta *include* no comando, por exemplo:

```
// pwd output: ./mydir/Projeto
g++ -I ./include/*.hpp src/*.cc
```

2.1.2 Gerando o executável usando o programa *GNU Make*

O arquivo comprimido contém, também, um *Makefile*, que facilita a construção do executável. Com o programa *GNU Make* instalado, considerando que o arquivo comprimido original foi extraído em uma pasta qualquer, é necessário apenas entrar na pasta *Projeto* e digitar o comando *make*. Depois disso, será gerado um binário na pasta *bin*, que deve ser usado para executar o programa.

2.2 Escolha do Método de Ordenação

O método padrão de ordenação é o Heap Sort, ou seja, se você, após extrair o arquivo comprimido, construir um binário, o método que será utilizado para ordenação será o Heap Sort. Para alterar do Heap Sort para os outros métodos de ordenação basta comentar a seção que se refere ao mesmo no arquivo *Main.cc* e descomentar a seção referente aos outros métodos.

Escolhas:

- Heap Sort

  Escolha padrão. A sua seção no main está nas linhas 48-70, claramente separada das outras por um comentário no código. Para fazer uso de outros métodos é necessário comentar toda essa seção Heapsort.
- Quick, Insertion e Shell Sort

  Para usar qualquer um desses métodos, basta comentar toda a seção referente ao Heap Sort e descomentar toda a seção que está nas linhas 26 até 44. Depois disso, é necessário escolher o método que será utilizado nas linhas 36, 37, 38 e 39, cada uma se referindo claramente à cada um dos métodos.

3 Análise do código

Nessa seção serão explicadas as decisões tomadas ao longo do desenvolvimento, será feita também uma análise sobre o comportamento dos diferentes algoritmos testados e serão apresentados gráficos e tabelas referentes ao tempo de cada um para diferentes entradas, variando no tamanho e ordenação.

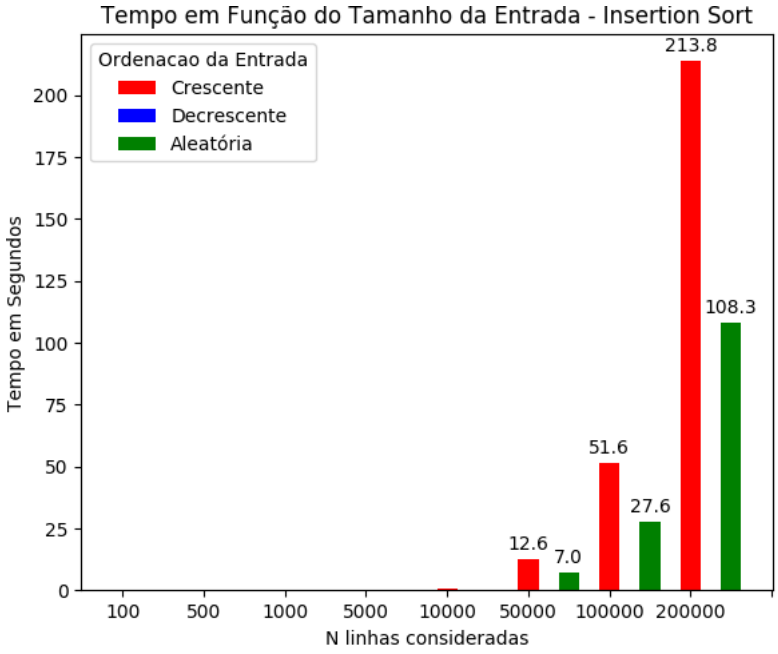
Considerando que as bases com maior prioridade são aquelas que requerem mais saltos, a entrada será ordenada de maneira decrescente, ou seja, o melhor caso dos algoritmos, em geral, se dará quando a entrada estiver ordenada de maneira decrescente e o pior quando estiver ordenada de maneira crescente, sendo necessário inverter toda a entrada. de alguma forma.

3.1 Algoritmos Vistos em Sala

3.1.1 Insertion Sort

O primeiro método que foi sugerido quando um dos times de engenharia soube que as entradas poderiam estar quase ordenadas, pois ele mantém um custo linear nesse caso. Além disso, é um método estável. Por outro lado, é importante considerar que o método tem um alto custo tratando do número de movimentações e seu pior caso pode deteriorar sua complexidade assintótica temporal para $O(n^2)$.

A sua implementação é simples e pouco susceptível à erros, o custo assintótico espacial é constante. Devido à sua deterioração para os piores casos, o gráfico referente ao desempenho do *Insertion Sort* para as diferentes entradas será apresentado aqui.



Conforme era esperado, a entrada ordenada de maneira crescente, pior caso, obteve um tempo de execução dezenas de vezes pior que quando a entrada era ordenada de

maneira decrescente, o melhor caso, será possível observar esse último na comparação geral com os outros métodos.

Esse tempo maior se dá principalmente pelo número de comparações e movimentações que, no pior caso, tem complexidade assintótica quadrática enquanto no melhor caso é linear. Quando a entrada estava ordenada de maneira aleatória, o resultado foi intermediário entre o melhor e pior caso, mas quando comparado à outros algoritmos nota-se um custo de tempo consideravelmente maior, tornando esse método pouco adequado para uso na aplicação.

Custo	Melhor Caso	Pior Caso
Temporal	$O(n)$	$O(n^2)$
Espacial	$O(1)$	$O(1)$

3.1.2 Heap Sort

Esse foi o método sugerido pelos mais experientes engenheiros, não só pela sua elegância, mas também porque não acham muito interessante se expor aos riscos da complexidade temporal de outros métodos, já que esse se comporta sempre $O(n \log n)$, independentemente da entrada, e tem complexidade espacial constante. Por outro lado, o *Heap Sort* não é um algoritmo de ordenação estável e seu anel interno tem uma complexidade consideravelmente maior que outros algoritmos, como o *Quick Sort*, não sendo a alternativa mais rápida em alguns casos por essa razão.

A sua implementação requer bastante atenção, sendo que, assim como nos outros métodos, para ordenar de maneira decrescente, foram necessárias modificações no algoritmo de ordenação mostrado pelos professores[2]. O algoritmo modificado consiste em:

1. Criar um *min heap*, onde a raiz é sempre o menor elemento, com o vetor de entrada
2. Trocar a raiz com o elemento na última posição que ainda não foi trocada no vetor
3. Criar um *min heap* com o vetor original menos as posições já substituídas
4. Repetir os passos 2 e 3 até que o vetor esteja garantidamente ordenado

Como os menores elementos vão sendo colocados no fim do vetor, depois de iterar ao longo do vetor e recriar o *min heap* n vezes, podemos garantir que a entrada está ordenada de maneira decrescente, como era desejado.

Custo	Melhor Caso	Pior Caso
Temporal	$O(n \cdot \log n)$	$O(n \cdot \log n)$
Espacial	$O(1)$	$O(1)$

3.1.3 Quick Sort

Um dos métodos mais amplamente utilizado pelos engenheiros, é um algoritmo que se enquadra na classe dos que usam a estratégia "*Dividir para Conquistar*" para ordenar, dividindo a entrada em subpartes, ordenando-as e então combinando-as.

Tratando da sua implementação, é necessário tomar cuidado com o delicado processo de partição do vetor, e é nesse processo também que podem ser feitas otimziações visando melhorar o seu custo assintótico temporal e evitar o pior caso, que acontece quando temos um arquivo ordenado e o pivô escolhido é repetidamente um dos extremos(maior ou menor), pois o procedimento de ordenação conseguirá eliminar apenas um item por chamada recursiva nesse caso.

Os principais pontos à favor da escolha desse método são sua alta eficiência na ordenação, complexidade assintótica temporal média de $O(n \log n)$ e número de comparações também proporcional a $n \log n$ aproximadamente. Além de precisar apenas de uma pilha como memória auxiliar.

Por outro lado, é importante considerar esse é um método que não é estável e que o seu pior caso tem complexidade assintótica temporal $O(n^2)$. Outrossim, a implementação do mesmo requer demasiada atenção, já que pequenos erros podem levar à resultados inesperados para certas entradas.

Custo	Melhor Caso	Pior Caso
Temporal	$O(n \cdot \log n)$	$O(n^2)$
Espacial	$O(\log n)$	$O(\log n)$

3.2 Algoritmos Novos ou Modificados

3.2.1 Quick Sort Otimizado

Conhecendo algumas deficiências do popular método de ordenação, um *Quick Sort* modificado foi proposto e buscou otimizar os seguintes pontos:

- Custo das chamadas recursivas
- Número de comparações e pior caso

Tratando do primeiro ponto listado, a solução mais óbvia foi implementar o algoritmo de maneira não recursiva, evitando assim o custo das várias chamadas recursivas. Assim como na sua implementação recursiva, requer bastante atenção.

Sobre o segundo ponto, sabendo que o pior caso do *Quick Sort* acontece quando escolhemos o primeiro ou último elemento em vetores quasi ordenados, o melhor pivô possível é a mediana do seguimento sendo ordenado, pois permitirá que o algoritmo divida o segmento em partes aproximadamente iguais. A vantagem de usar esse algoritmo da mediana de 3 elementos é que ele garante que duas partições estão o com o tamanho mais próximo de iguais possível, minimizando o custo final da ordenação.

A implementação da melhoria para escolha da mediana de 3 elementos foi feita dessa forma porque, caso a mediana fosse ser calculada com base em todos os elementos, esse

passo poderia dominar o custo assintótico temporal. Tendo isso em vista, o algoritmo funciona da seguinte forma:

1. Escolher três elementos de maneira aleatória no segmento sendo ordenado
2. Calcular a mediana entre esses três elementos
3. Usar essa mediana, calculada com base em elementos aleatórios, como pivô

Embora exista um aumento no custo computacional para realizar o cálculo da mediana $O(\log n)$ vezes, fazer uso dessa técnica ajuda a reduzir o risco do pior caso e, além disso, o número de comparações esperadas para o caso médio diminui de $\approx 1,4 \cdot n \cdot \log n$ para $\approx 1,2 \cdot n \cdot \log n$[3].

Custo	Melhor Caso	Pior Caso
Temporal	$O(n \cdot \log n)$	$O(n^2)$
Espacial	$O(\log n)$	$O(\log n)$

3.2.2 Shell Sort

Esse algoritmo foi escolhido por ser uma extensão do algoritmo de ordenação por inserção e contornar alguns de seus problemas, além de ter uma implementação bastante simples, com poucas linhas de código.

O problema do algoritmo de inserção tratado é[4]:

- Troca itens adjacentes para determinar o ponto de inserção
- São efetuadas $n - 1$ comparações e movimentações quando o menor item está na posição mais a direita no vetor

Para contornar o problema citado, o *Shell Sort* permite trocas de elementos que estão distantes um do outro. O algoritmo compara itens separados por uma série de distâncias h e os ordena, todo h-ésimo item leva a uma sequência ordenada e essa sequência é dita estar *h-ordenada*. O valor de h diminui e quando $h = 1$ o algoritmo funciona como o algoritmo de inserção, que tem excelente desempenho em vetores ordenados e quasi ordenados.

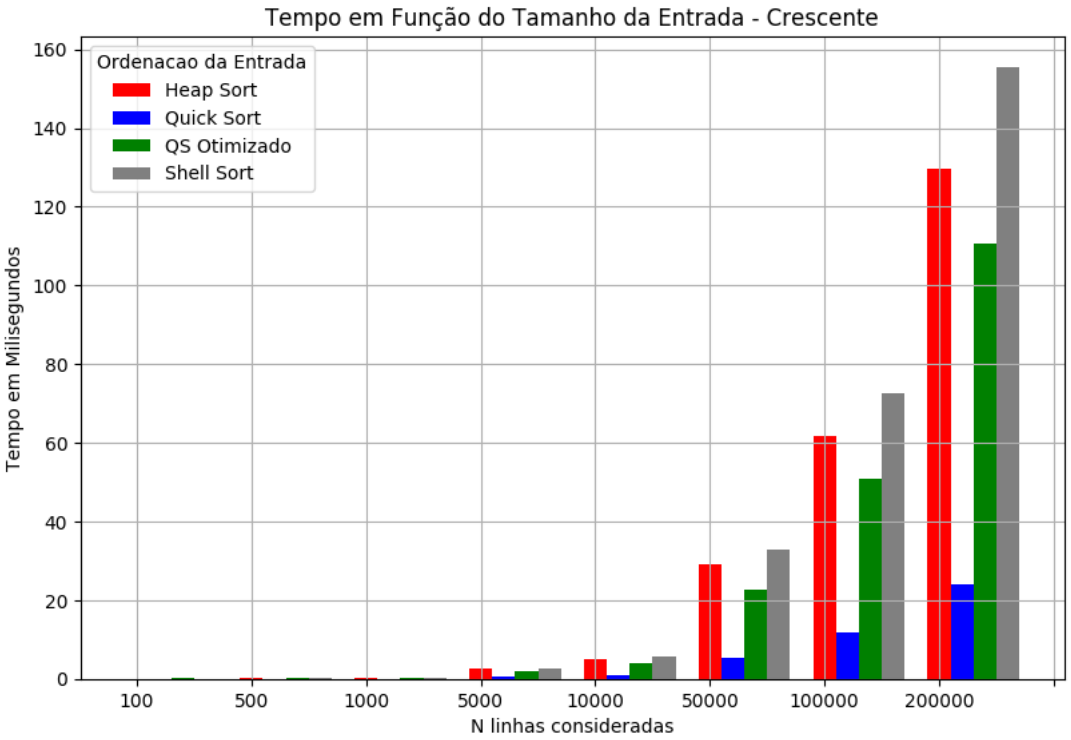
Todavia, embora mostre significativa melhora sobre o método de inserção, o *Shell Sort* é um método não estável e seu tempo de execução é bastante sensível à entrada, como será possível ver na seção de comparação entre os métodos.

Custo	Melhor Caso	Pior Caso
Temporal	$O(n \cdot \log n)$	$O(n \cdot (\log n)^2)$
Espacial	$O(1)$	$O(1)$

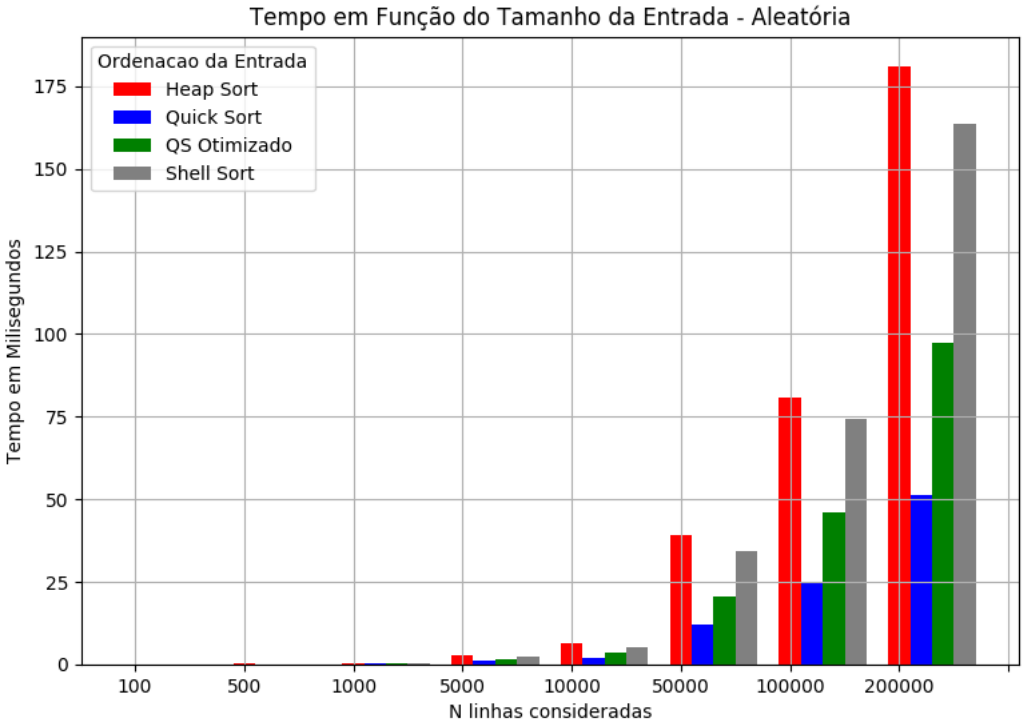
3.3 Comparação

Sabendo de todas as considerações acerca de cada um dos métodos, o teste empírico e a comparação entre os mesmos são procedimentos necessários para estabelecer um juízo bem embasado sobre qual deles se adequará melhor à aplicação.

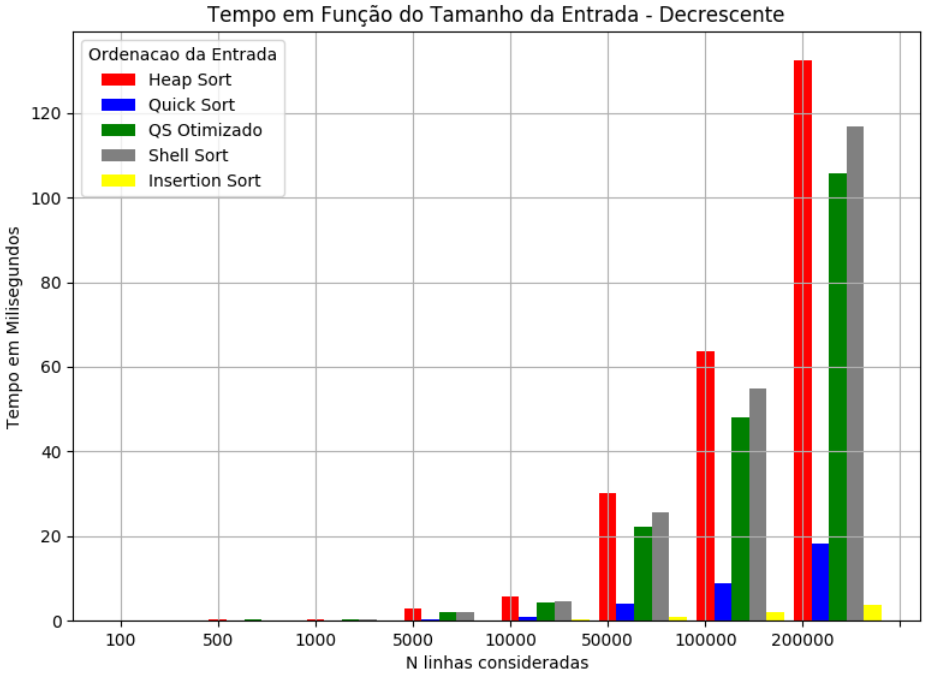
Novamente, é importante considerar que como os planetas que requerem mais saltos hiperspaciais são a prioridade inicial, o melhor caso se dará quando o arquivo estiver ordenado de maneira decrescente e vice versa. Serão apresentados à seguir o tempo de cada um dos algoritmos para tamanhos de entranda $n \in \{100, 500, 1000, 5000, 10000, 50000, 100000, 200000\}$.



Diante do gráfico com o tempo gasto por cada um dos algoritmos no pior caso, é possível observar que o *Heap Sort* teve um crescimento *quasi* linear, assim como o *Quick Sort* e o *Quick Sort Otimizado*, o que sugere que o tamanho do arquivo de entrada ainda não foi suficiente para destacar a natureza quadrática do pior caso com o *Quick Sort*. O melhor tempo, com uma grande margem, foi desse mesmo algoritmo, enquanto o pior foi o do *Shell Sort*.



A análise do gráfico com o tempo gasto considerando entradas ordenadas de maneira aleatória sugere que, na verdade, a ordenação aleatória pode ter criado diversos subvetores ordenados, pois o tempo gasto foi comparável com o pior caso. Nota-se, de qualquer modo, que embora todos os algoritmos tenham aumentado seu tempo gasto para ordenar, o *Quick Sort* teve seu tempo praticamente dobrado em relação ao pior caso e o *Quick Sort Otimizado* teve um tempo menor para $n = 200000$, demonstrando a resiliência do algoritmo para evitar oscilações muito grandes no tempo de execução. É valido mencionar também que o *Heap Sort* apresentou a menor variação, validando sua característica de ser um método com comportamento bastante estável quanto ao tempo gasto para ordenação.



Agora, considerando o melhor caso, é importante inserir também os dados referentes ao *Insertion Sort*, pois ele apresenta um desempenho excelente quando a entrada está ordenada, lembrando também que uma das possíveis melhorias para serem adicionadas ao *Quick Sort* a sua utilização para pequenos subvetores. O método de ordenação por inserção se mostrou cerca de três vezes mais rápido que o segundo colocodo, o *Quick Sort*.

O *Heap Sort*, mais uma vez, apresentou uma variação muito pequena em relação aos outros casos. O mesmo pode-se dizer do *Quick Sort Otimizado*, que se mostrou uma implementação com tempo de execução bastante estável em detrimento do *Quick Sort* e do *Shell Sort*.

		T(ms)	T(ms)	T(ms)	T(ms)	T(ms)	T(ms)	T(ms)	T(ms)
Crescente	**N de Elementos**	**100**	**500**	**1000**	**5000**	**10000**	**50000**	**100000**	**200000**
	Insertion Sort	0.188273	1.38376	5.75769	126.238	498.598	12593.2	51568	213812
	Heap Sort	0.031104	0.196466	0.391794	2.57024	5.2106	29.2756	61.6936	129.7038
	Quick Sort	0.06387	0.052994	0.111662	0.519014	1.059726	5.35916	11.76906	24.0816
	Quick Sort Opt	0.234808	0.224954	0.435074	1.99289	4.04944	22.7738	50.8294	110.6232
	Shell Sort	0.11951	0.203524	0.414658	2.54966	5.63534	32.8198	72.6848	155.4542
		T(ms)	T(ms)	T(ms)	T(ms)	T(ms)	T(ms)	T(ms)	T(ms)
Decrescente	**N de Elementos**	**100**	**500**	**1000**	**5000**	**10000**	**50000**	**100000**	**200000**
	Insertion Sort	0.00546	0.011746	0.020674	0.089732	0.178546	0.925858	1.88407	3.75651
	Heap Sort	0.03402	0.098306	0.211622	1.37833	2.91382	15.0288	31.7789	66.265
	Quick Sort	0.013178	0.021788	0.041724	0.199159	0.395223	2.02823	4.4741	9.15899
	Quick Sort Opt	0.046596	0.106826	0.188282	0.932573	2.20175	11.0447	23.9959	52.9057
	Shell Sort	0.020254	0.073522	0.142917	0.984724	2.30669	12.8059	27.4113	58.4057
		T(ms)	T(ms)	T(ms)	T(ms)	T(ms)	T(ms)	T(ms)	T(ms)
Aleatória	**N de Elementos**	**100**	**500**	**1000**	**5000**	**10000**	**50000**	**100000**	**200000**
	Insertion Sort	0.038928	0.685767	3.46478	73.927	292.214	7041.87	27642.5	108325
	Heap Sort	0.038561	0.230284	0.494513	3.00514	6.38121	39.2721	80.9178	181.019
	Quick Sort	0.01502	0.096148	0.195295	1.01146	2.12003	11.9156	24.6517	51.1312
	Quick Sort Opt	0.036744	0.178842	0.341509	1.77083	3.66662	20.6795	45.8731	97.4092
	Shell Sort	0.029607	0.146835	0.360094	2.23231	5.08222	34.346	74.2509	163.771

Figura 1: Medidas de tempo para diferentes ordenações medidas em milisegundos

4 Conclusão

Diante de tudo que foi exposto, é possível concluir que a escolha do método de ordenação mais apropriado para a ordenação das bases por nível de prioridade depende da relevância de fatores como estabilidade do algoritmo, espaço disponível e a tolerância a variâção ou não no tempo do algoritmo.

Através das análises estáticas e dinâmicas, os métodos que mais se destacaram, considerando os fatores mencionados acima e que não é possível conhecer previamente se a entrada estará ordenada ou não, foram:

- Heap Sort

  Uma boa opção caso a tolerância à variação no tempo de execução do algoritmo e o custo espacial sejam os fatores mais relevantes.

- Quick Sort Otimizado

  O Quick Sort otimizado introduziu mudanças para evitar o pior caso do Quick Sort e também reduzir o gasto extra de memória com as diversas chamadas recursivas, embora acabe usando uma fila para realizar um trabalho análogo às chamadas recursivas do compilador.
  Esse método teve um tempo de execução que variou pouco com a entrada e foi mais rápido que o *Heap Sort* em todos os casos, em parte pelo seu anel interno de execução, que é consideravelmente mais simples que o do *Heap Sort*. Caso o custo espacial não seja um impedimento, é uma opção interessante e que não apresenta variação no tempo de execução tão notável quanto às outras opções.

Outros métodos, como o *Insertion Sort*, que se mostrou extremamente eficiente quando a entrada está ordenada, se mostraram o melhor para certas situações, mas com comportamentos que variam muito conforme a entrada. Sabendo que não há como prever a forma que a entrada chegará, e que provavelmente ela estará desordenada, é preciso considerar principalmente o caso para entrada organizada de maneira aleatória e o pior caso de cada um dos métodos.

É valido mencionar também que o *Quick Sort Otimizado* ainda poderia ser melhorado, por exemplo, com a chamada para o *Insertion Sort* para pequenos subvetores, pois nesse caso o método de inserção é mais eficiente que o relativamente complexo processo de particionamento feito pelo *Quick Sort Otimizado*.

Por fim, nesse trabalho foi possível verificar a validade de diversos conceitos relacionados aos diversos algoritmos de ordenação e suas especificidades. Em especial, a versão otimizada do *Quick Sort* chamou bastante atenção, assim como o *Heap Sort*, e ambos os algoritmos devem ser considerados pelo time de engenharia junto com os fatores de maior relevância para fazer a melhor escolha para a implementação final no projeto.

Referências

[1] "Standard library header <chrono>," https://en.cppreference.com/w/cpp/header/chrono, 2021, [Online; accessed 28-Feb-2021].

[2] L. Chaimowicz and R. Prates, "Slides virtuais da disciplina de Estruturas de Dados," disponibilizado via moodle. Departamento de Ciencia da Computacao. Universidade Federal de Minas Gerais. Belo Horizonte.

[3] "Quicksort optimizations," https://yourbasic.org/golang/quicksort-optimizations/#pivot-element, 2021, [Online; accessed 28-Feb-2021].

[4] N. Ziviani, *Projeto de algoritmos: com implementacoes em Pascal e C.* Pioneira Thomson Learning, 2004, cap. 4 Ordenação.