UFMG UNIVERSIDADE FEDERAL DE MINAS GERAIS	**Universidade Federal de Minas Gerais** **Turma:** Ciência da Computação **Prof.:** Gisele e Wagner **Nome:** Rubia Alice Moreira de Souza **Matrícula:** 2022043507

TP2 - Comparação Algoritmos Ordenação

Belo Horizonte

2022

Sumário

1. Introdução

Este trabalho foi implementado com dois fluxos de execução principais, que podem ser selecionados pela flag “-r” ao executar o programa. O primeiro, é responsável por testar os diferentes algoritmos de Quick Sort e seus aprimoramentos. Já o segundo, realiza os testes do melhor Quick Sort, do Merge Sort e do Heap Sort. Nosso objetivo principal com o trabalho é analisar as diferentes vantagens e desvantagens de cada implementação e identificar qual o melhor contexto para utilizá-las.

2. Método

2.1. Classes

Inicialmente, temos as classes responsáveis pela ordenação. Todas essas classes possuem os seguintes métodos estáticos padronizados:

- ordenarCrescente(vetor : Registro, tamanho : unsigned int) : void
- ordenarDecrescente(vetor : Registro, tamanho : unsigned int) : void

Assim, nós temos uma classe para cada algoritmo de ordenação que será avaliado neste trabalho. São eles:

Parte 1: Avaliação Quick Sort

- **QuickSort:** Implementação padrão do Quick Sort
- **QuickSortMediana:** Adaptação do Quick Sort para selecionar a mediana entre pivôs aleatórios
- **QuickSortSelecao:** Adaptação do Quick Sort para iniciar o método de ordenação por Seleção a partir de um certo tamanho de vetor
- **QuickSortIterativo:** Adaptação não recursiva do Quick Sort
- **QuickSortIterativoInteligente:** Adaptação não recursiva do Quick Sort que prioriza a menor pilha para ser ordenada primeiro

Parte 2: Quick Sort, Merge Sort e Heap Sort

- **QuickSort:** Melhor algoritmo dentre os presentes na parte 1
- **MergeSort**
- **HeapSort**

Apenas as classes QuickSortSelecao e QuickSortMediana possuem um terceiro parâmetro que está relacionado ao seu funcionamento. No primeiro, ele representa o valor

mínimo para iniciar a ordenação por seleção e, no segundo, quantos pivôs serão selecionados aleatoriamente.

Além disso, temos um arquivo de “Utils”, que contém funções para avaliar a ordenação dos vetores e gerar elementos aleatórios:

Utils
gerarStringAleatoria(tamanho : unsigned int) : std::string gerarNumeroInteiroAleatorioEmUmaFaixa(valorInicial : int, valorFinal : int) : int gerarNumeroInteiroAleatorio() : int gerarNumeroFloatAleatorioEmUmaFaixa(valorInicial : float, valorFinal : float) : float gerarNumeroFloatAleatorio() : float gerarRegistroAleatorio() : Registro* verificarOrdenacaoCrescente(vetor : Registro[], tamanho : unsigned int) : void verificarOrdenacaoDecrescente(vetor : Registro[], tamanho : unsigned int) : void

Também temos a classe de “Registro”, conforme definida no enunciado. Essa classe tem diversas sobrecargas dos operadores de comparação a fim de facilitar o uso nos algoritmos de ordenação.

Registro	
Atributos	**Métodos**
chave : int textos : char[15][200] numeros : float[10]	operator<=(outroRegistro : Registro) : bool operator<(outroRegistro : Registro) : bool operator>=(outroRegistro : Registro) : bool operator>(outroRegistro : Registro) : bool print() : void

Por fim, temos uma estrutura de “Pilha”, que foi implementada a fim de simular a stack para o Quick Sort não recursivo. Basicamente, ela é responsável por armazenar quais serão as próximas faixas/partes do vetor que serão ordenadas:

FaixaOrdencao	
Atributos	**Métodos**
inicio : int fim : int	

Celula

Atributos	Métodos
proxima : Celula* anterior : Celula* valor : FaixaOrdencao*	

Pilha	
Atributos	**Métodos**
topo : Celula* base : Celula* tamanho : unsigned int	empilhar(faixa : FaixaOrdencao) : void desempilhar() : FaixaOrdencao* estaVazia() : bool getTamanho() : unsigned int limpar() : void print() : void

Por fim, temos a classe “DadosDesempenho”, que é responsável por armazenar as estatísticas de desempenho dos algoritmos de ordenação:

DadosDesempenho	
Atributos	**Métodos**
quantidadeComparacoes : unsigned int quantidadeAtribuicoes : unsigned int	print() : void

2.2. Configurações de Ambiente

O código foi executado nas seguintes configurações de ambiente:

- **Sistemas operacionais:** Windows, Linux e na WSL do Windows utilizando a versão Ubuntu
- **Linguagem utilizada:** C++11
- **Compilador utilizado:** g++ da GNU Compiler Collection
- **Processador utilizado:** Intel(R) Core(™) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Núcleo(s), 4 Processador(es) Lógico(s)
- **Quantidade de memória RAM disponível:** 8Gb

3. Análise de Complexidade

Inicialmente, temos a análise do Quick Sort padrão. O seu melhor caso ocorre quando o vetor é sempre dividido ao meio a cada iteração, que realiza O(n) operações em O(log(n)) chamadas. Assim, o custo do melhor caso seria O(nlog(n)), devido ao paradigma dividir para conquistar. Já o pior caso, ocorre quando o pivô selecionado é sempre um dos

extremos da ordenação (o menor ou maior elemento), fazendo com que haja O(n) chamadas, ao invés de O(log(n)). Portanto, o custo final passa a ser quadrático $O(n^2)$.

Já com relação ao espaço, há sempre um custo do vetor a ser ordenado, que tem tamanho “n”. Contudo, além disso, também há um custo extra de espaço gerado pelas chamadas recursivas, que pode ser de O(log(n)) no melhor caso ou O(n) no pior caso.

Quick Sort		
Caso	**Tempo**	**Espaço**
Melhor	O(nlog(n))	O(n) + O(log(n)) = O(n)
Médio	O(nlog(n))	
Pior	$O(n^2)$	O(n) + O(n) = O(n)

Para o Quick Sort com Mediana, a sua complexidade de tempo varia segundo a implementação utilizada. Para métodos como Median of Medians, o pior caso pode reduzir para O(n). No entanto, a implementação especificada para o trabalho apenas reduz a probabilidade do pior caso ocorrer. Portanto, o custo no pior caso continua sendo $O(n^2)$. Além disso, há um custo de processamento e espaço um pouco maior, devido à necessidade de encontrar a mediana e alocar os possíveis pivôs em um array, respectivamente. No entanto, ambos não são suficientes para alterar o custo assintótico.

Quick Sort Mediana		
Caso	**Tempo**	**Espaço**
Melhor	O(nlog(n))	O(n)
Médio	O(nlog(n))	
Pior	$O(n^2)$ ou O(n) (dependendo da implementação)	

Para a forma híbrida do Quick Sort, o custo de tempo passa a depender do ponto em que o algoritmo irá alterar o modo de ordenação. Nesse caso, passamos a ter duas variáveis, o valor de “n”, que representa o tamanho do vetor a ser ordenado, e “k”, que indica o tamanho inicial em que a ordenação irá alterar seu método.

Quick Sort Seleção		
Caso	**Tempo**	**Espaço**
Melhor	O(nk + nlog(n/k))	O(n)
Médio		

Pior		

Já para o Quick Sort Iterativo, não há mudanças em relação ao seu custo assintótico de tempo. A principal mudança gerada nesse algoritmo é remover a necessidade de utilizar a Pilha de Execução para organizar quais partições do vetor serão ordenadas. Ao invés dela, é utilizada uma estrutura de pilha padrão, que é responsável por empilhar e desempilhar as faixas do vetor que serão ordenadas durante a execução. O custo de espaço ainda depende, principalmente, do array de tamanho “n” a ser ordenado, mas o espaço extra passa a depender do tamanho da estrutura da pilha utilizada (considerando um tamanho k), ao invés das chamadas recursivas. Assim, o custo final predominante é apenas o tamanho do array a ser ordenado.

Quick Sort Iterativo		
Caso	**Tempo**	**Espaço**
Melhor	O(nlog(n))	O(n) + O(k) = O(n)
Médio	O(nlog(n))	
Pior	$O(n^2)$	

Já o Quick Sort Iterativo Inteligente, consegue reduzir o tamanho máximo da pilha que contém as faixas de ordenação para log(k), ao ordenar as faixas de menor tamanho primeiro. Assim, ela apresenta uma otimização em relação a utilização de memória.

Quick Sort Iterativo Inteligente		
Caso	**Tempo**	**Espaço**
Melhor	O(nlog(n))	O(n) + O(log(k)) = O(n)
Médio	O(nlog(n))	
Pior	$O(n^2)$	

O Merge Sort tem um comportamento equivalente ao Quick Sort, uma vez que ambos são baseados no princípio de dividir para conquistar. Assim como o Quick Sort, o Merge Sort tem um custo de O(nlog(n)) para o melhor e para o caso médio. No entanto, o Merge Sort consegue garantir um pior caso de O(nlog(n)), uma vez que o algoritmo sempre realiza a divisão ao meio. A maior desvantagem do Merge Sort em relação ao Quick Sort é o fato de ele não ser in-place. Ou seja, é necessário alocar uma quantidade de memória extra para poder realizar o Merge dos subvetores ordenados. Assim, além do custo O(n) do array a ser ordenado, há também o custo de O(n) devido aos vetores auxiliares alocados.

Merge Sort		
Caso	**Tempo**	**Espaço**
Melhor	O(nlog(n))	O(n) + O(n) = O(n)
Médio	O(nlog(n))	
Pior	O(nlog(n))	

Já o HeapSort é uma outra alternativa que é capaz de evitar o pior caso do Quick Sort. Assim como no Merge Sort, o Heap Sort tem em todos os casos uma complexidade de tempo de O(nlog(n)). Isso pois, o algoritmo tem duas partes principais. Uma que consiste em organizar o array em um heap e outra que consiste em ajustar o heap ao longo da ordenação, sendo que ambas possuem como custos O(nlog(n)) e O(log(n)), respectivamente. Assim, o custo assintótico predominante é O(nlog(n)). Além disso, uma vantagem em relação ao Merge Sort, é que este algoritmo é in-place, não utilizando nenhum espaço extra além do array que será ordenado.

Heap Sort		
Caso	**Tempo**	**Espaço**
Melhor	O(nlog(n))	O(n)
Médio	O(nlog(n))	
Pior	O(nlog(n))	

4. Estratégias de Robustez

As estratégias implementadas têm o objetivo de garantir a entrada correta de dados pelas flags do programa. Todas são exceções que podem ser lançadas na main do programa nos seguintes casos:

- ArquivoEntradaNaoFornecidoException: É lançada caso o arquivo de entrada não seja informado pela flag “-i”
- ArquivoSaidaNaoFornecidoException: É lançada caso o local do arquivo de saída não seja informado pela flag “-o”
- ArgumentoInvalidoExcecao: É lançada caso nenhuma método de ordenação tenha sido escolhido pela flag “-v”
- ArquivoEntradaNaoAbertoException: É lançada caso não seja possível abrir o arquivo de entrada
- ArquivoSaidaNaoAbertoException: É lançada caso não seja possível abrir ou criar o arquivo de saída
- ErroAoEncerrarMemLogException: É lançada caso o memlog não seja encerrado corretamente

- ArgumentoInvalidoExcecao: É lançada ao selecionar a versão do Quick Sort com Mediana, mas não informar uma valor para a quantidade de pivôs que serão selecionados aleatoriamente
- ArgumentoInvalidoExcecao: É lançada ao selecionar a versão do Quick Sort com Seleção, mas não informar um valor mínimo para iniciar a ordenação por seleção

5. Análise Experimental

5.1. Desempenho Computacional

Inicialmente, vamos comparar os algoritmos que utilizam parâmetros com diferentes valores. Esse é o caso do Quick Sort com Mediana e com Seleção.

5.1.1. Parte 1: Impacto de Variações do Quick Sort

5.1.1.1. Quick Sort com Mediana

Considerando os resultados da avaliação experimental do Quick Sort com as diferentes medianas, obtivemos os seguintes resultados.

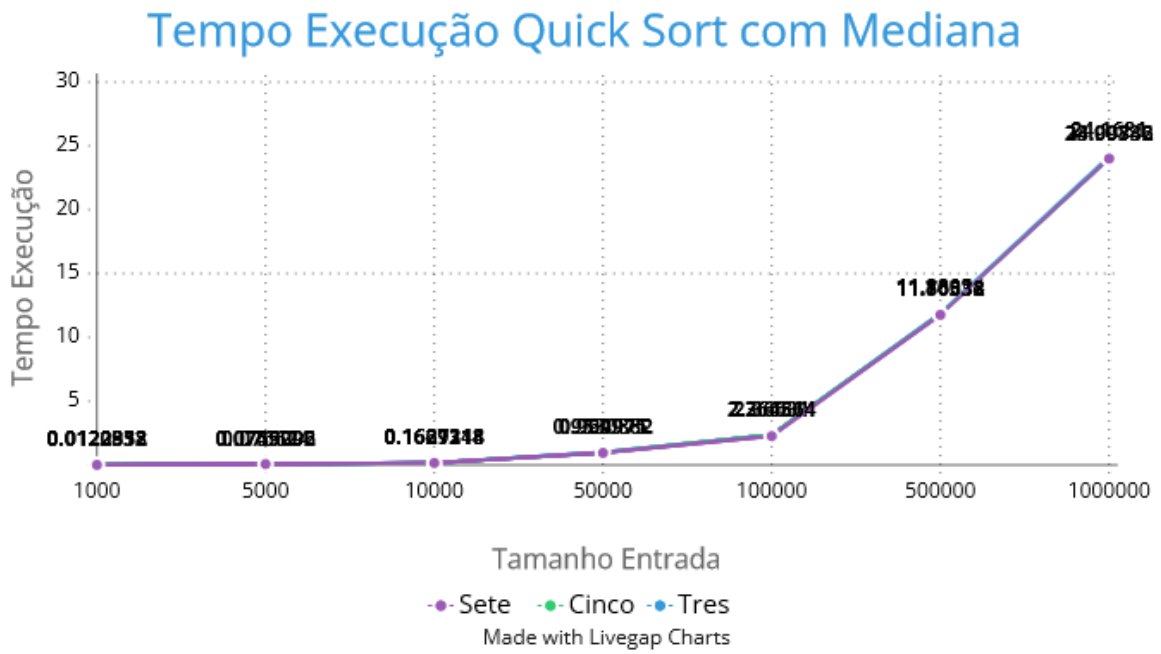
Quick Sort Mediana de Três							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Comparações	13997	82848,6	182643,4	1054488,2	2307984,6	12856932,6	27208479,6
Atribuições	8637,8	52169	111055,8	638333,2	1338427,8	7520951,4	15757644,4
Tempo	0,0120352	0,0749206	0,1661344	0,999975	2,36631	11,88054	24,1681

Quick Sort Mediana de Cinco							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Comparações	14034,6	86208,6	185026,4	1070965,6	2321769,8	13249031,2	27690514
Atribuições	8877	52377,4	111878,4	640633,4	1347463,4	7524557,6	15794254,6
Tempo	0,0122818	0,076624	0,1627118	0,964971	2,366514	11,80332	23,99346

Quick Sort Mediana de Sete							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000

Comparações	14066,6	84352	182802,2	1066177,2	2333930,2	13051320,6	27594183,2
Atribuições	8897,6	52595	112172,8	642389,6	1348930,6	7558562,2	15816757,6
Tempo	0,0122932	0,0755092	0,1629218	0,9530882	2,264864	11,76538	24,00732

É possível notar que alterar o valor da mediana não teve um efeito muito perceptível sobre o tempo de execução, uma vez que todos os casos se mantiveram, em média, próximos de 24 segundos. Isso torna-se ainda mais claro em uma representação gráfica:



Nele, nem é possível diferenciar claramente as linhas entre os diferentes algoritmos. Assim, é possível notar que a variação da mediana não gera um efeito sobre o tempo de execução, o que faz sentido, pois essa métrica tem o objetivo de evitar o pior caso do Quick Sort e não gerar uma melhora direta no desempenho.

Portanto, não há um algoritmo que prevaleça sobre os outros, uma vez que o tempo de execução foi bem próximo.

Observação: Como o propósito desses algoritmos é apenas evitar a probabilidade do pior caso ocorrer, soluções mais simples podem ser melhores na prática, como a seleção de um pivô aleatório ou do meio, pois demandam menos processamento e são mais simples de oferecer manutenção de código.

5.1.1.2. Quick Sort com Seleção

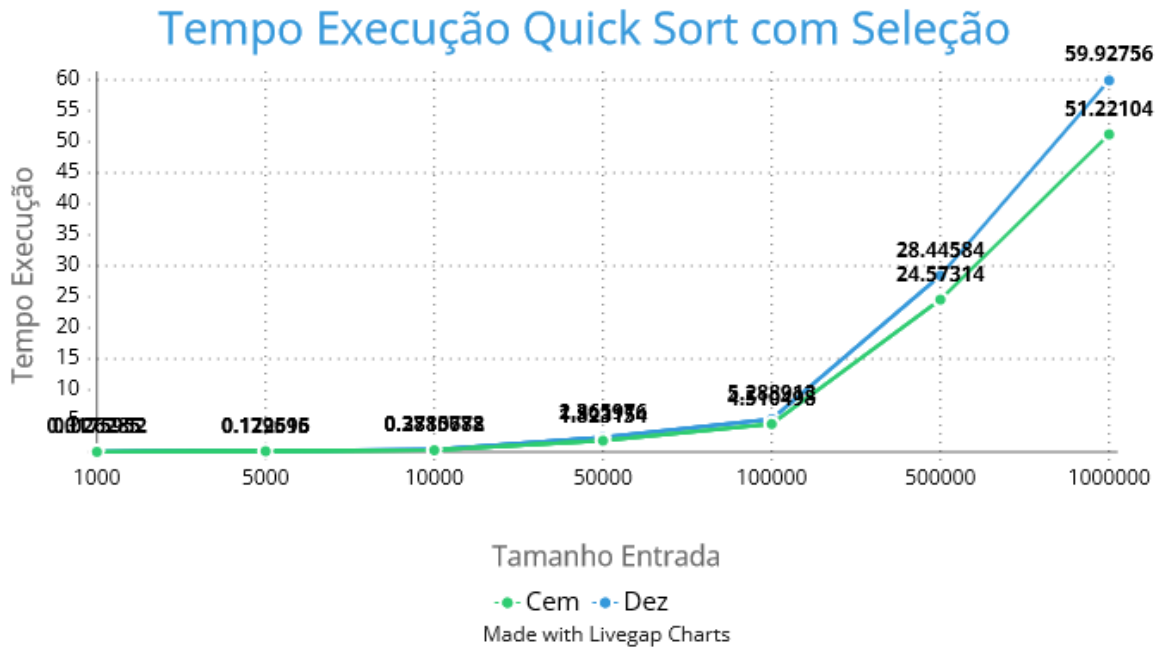
Em relação ao Quick Sort com Seleção, obtivemos os seguintes resultados:

Quick Sort Seleção de Dez

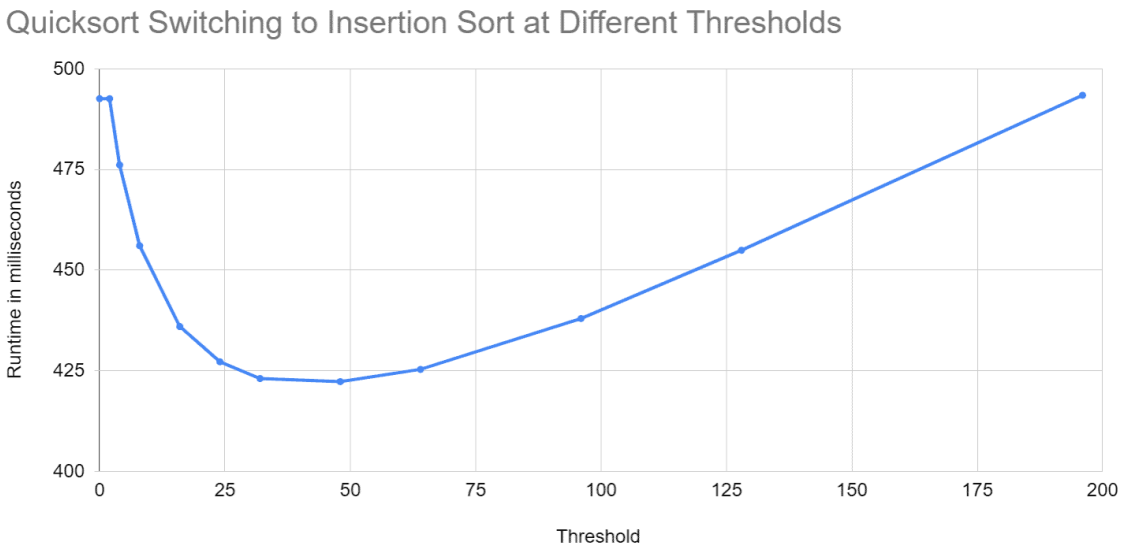
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Comparações	9269,6	62552,8	139424,6	867765	1841970,4	10843948,4	23200605,2
Atribuições	15089,6	100237,8	216531,8	1302537,8	2848836,8	16789817,4	35525017,8
Tempo	0,026285	0,172695	0,3715782	2,365976	5,288912	28,44584	59,92756

Quick Sort Seleção de Cem							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Comparações	0,026285	0,172695	0,3715782	2,365976	5,288912	28,44584	59,92756
Atribuições	8413,4	67329,4	153281,2	979411	2208846,2	13563590,4	29054586,2
Tempo	0,0175952	0,129516	0,2880678	1,823154	4,510498	24,57314	51,22104

Com base neles, é possível notar que aumentar o valor de “m” levou a uma redução do tempo de execução:



Em comparação com outras implementações, como a proposta por Sven Woltmann, utilizando o algoritmo de inserção, o tempo de execução tende a ser melhor quando os valores de “m” estão entre a faixa de 25 e 50:



Com base nisso, os resultados obtidos são um pouco diferentes dos esperados. Algumas possíveis justificativas para essa diferença seriam o ambiente computacional em que os testes foram executados e a implementação dos algoritmos, uma vez que o proposto por Sven Woltmann utiliza o algoritmo de inserção e o implementado no trabalho foi o de seleção.

Portanto, considerando os resultados obtidos, podemos verificar que o Quick Sort com Seleção com o valor de 100 foi mais eficiente.

5.1.1.3. Quick Sort Iterativo

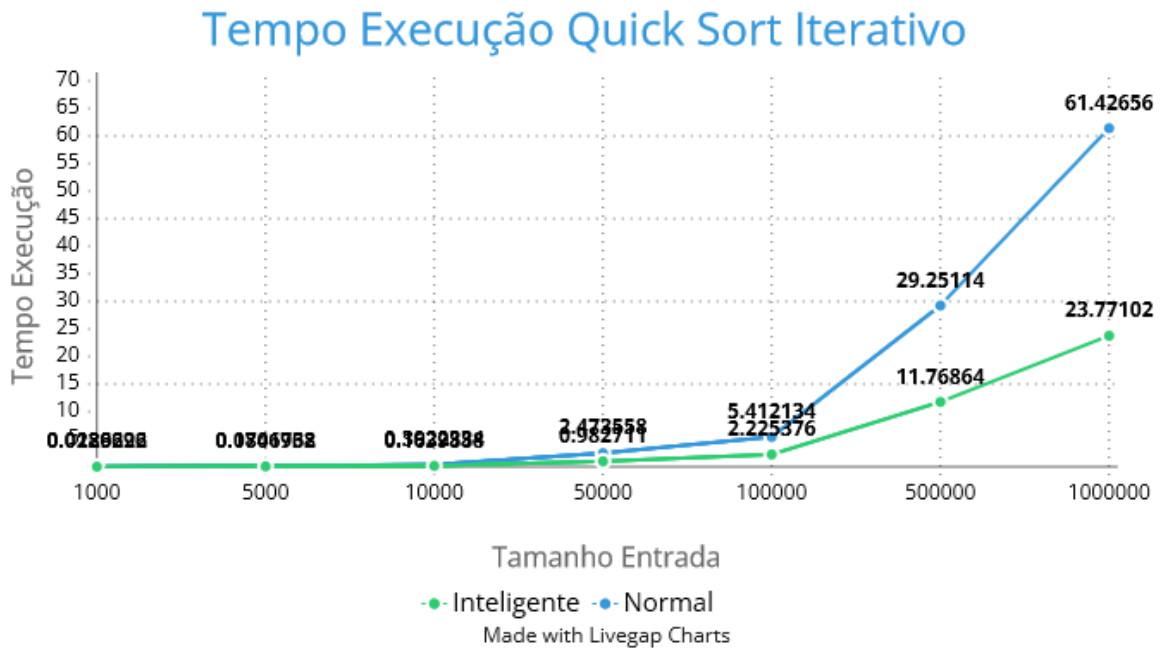
Por fim, podemos comparar os algoritmos do Quick Sort não recursivos.

Quick Sort Iterativo							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Comparações	10894,4	70233	154854,4	944829,4	1996005,6	11612697	24740216,4
Atribuições	19432,2	121681,8	259734,4	1518566,2	3279332	18943422,6	39835436,8
Tempo	0,0289692	0,1806968	0,3922334	2,473558	5,412134	29,25114	61,42656

Quick Sort Itertativo Inteligente							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Comparações	13281,6	82618,6	178008	1043490,2	2259471	12895600,4	27362520,4
Atribuições	9598,66	59176,96	127513,3	735010,72	1569406,7	8885203,44	15522738

Tempo	0,0126226	0,0741732	0,1639888	0,982711	2,225376	11,76864	23,77102

Novamente, temos um resultado interessante, pois o Quick Sort Iterativo Inteligente, ao ordenar as partições de tamanhos menores, supostamente, apresentaria um desempenho parecido com o versão iterativa normal, pois essa técnica tem o objetivo principal de evitar pilhas muito grandes. Contudo, o Quick Sort Iterativo Inteligente apresentou um desempenho muito superior em relação a versão iterativa normal:



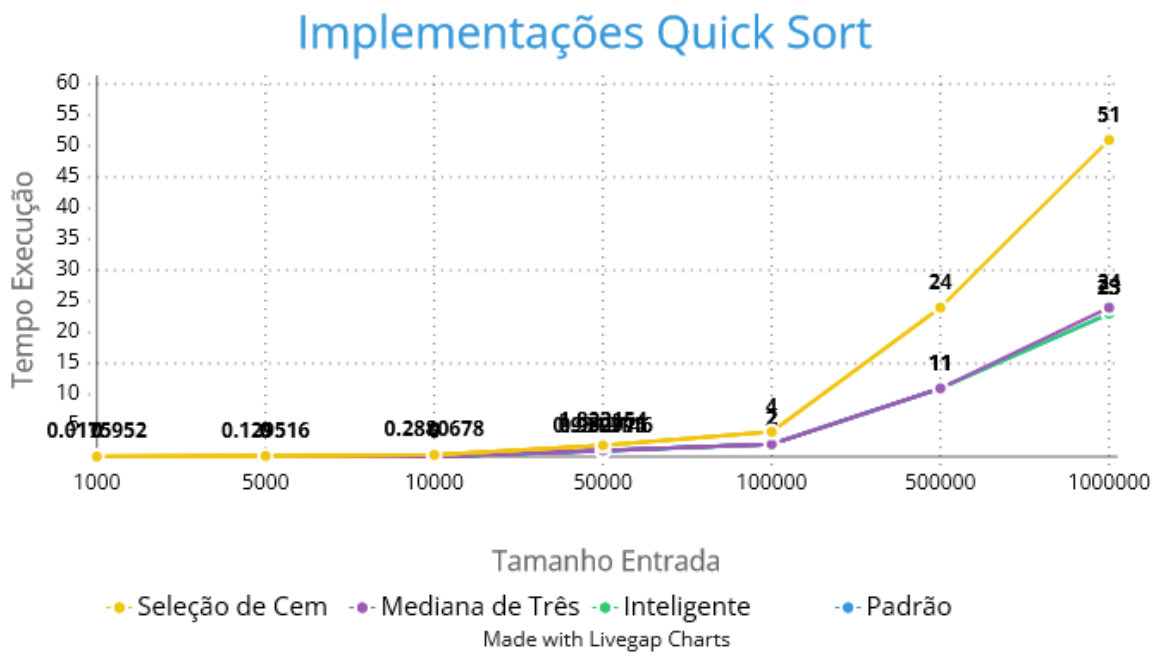
Assim, entre as duas implementações, temos que o Quick Sort Iterativo Inteligente prevalece em relação a sua implementação normal.

5.1.1.4. Diferentes Implementações Quick Sort

Uma vez que já sabemos qual a melhor versão dentre cada tipo de implementação do Quick Sort utilizada no trabalho, podemos começar a comparar o desempenho entre elas. A versão padrão do Quick Sort obteve os seguintes resultados:

Quick Sort							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Comparações	13281,6	82618,6	178008	1043490,2	2259471	12895600,4	27362520,4
Atribuições	8638	51380,4	109745	628644,4	1324270	7438379,2	15522738
Tempo	0,0117684	0,0741104	0,158806	0,9319146	2,282398	11,55856	23,68064

Se compararmos o tempo de execução de cada versão do Quick Sort, ou seja, Quick Sort com Mediana de Três, Quick Sort com Seleção de 100, Quick Sort Iterativo Inteligente e o Quick Sort Padrão, temos o seguinte gráfico:



Implementações Quick Sort							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Padrão	0,0117684	0,0741104	0,158806	0,9319146	2,282398	11,55856	23,68064
Inteligente	0,0126226	0,0741732	0,1639888	0,982711	2,225376	11,76864	23,77102
Mediana de Três	0,0120352	0,0749206	0,1661344	0,999975	2,36631	11,88054	24,1681
Seleção de Cem	0,0175952	0,129516	0,2880678	1,823154	4,510498	24,57314	51,22104

Nele é possível notar que o Quick Sort com Seleção de Cem foi o algoritmo que apresentou um crescimento maior do tempo de execução à medida que as entradas aumentam. Contudo, os outros algoritmos permaneceram bem próximos. Assim, podemos descartar o Quick Sort com Seleção de Cem como o mais eficiente para os resultados apresentados.

Entre os três que sobraram e tiveram um desempenho próximo, é possível notar que cada um tem um contexto em que seria ideal de ser utilizado e, por causa disso, ele seria uma escolha melhor em relação aos outros a depender da situação.

Caso o algoritmo seja inserido em um contexto em que há pouca disponibilidade de memória e é ideal ter controle sobre ela, o Quick Sort Iterativo Inteligente prevalece sobre os outros, uma vez que ele não realiza longas chamadas recursivas e que a pilha pode ser controlada conforme a necessidade.

Caso o algoritmo esteja em um sistema que irá lidar, mais frequentemente, com o pior caso do Quick Sort, o ideal seria utilizar o Quick Sort com Mediana de Três, uma vez que esse método consegue reduzir consideravelmente a probabilidade do pior caso ocorrer.

Por fim, se o algoritmo será inserido em um contexto em que há a necessidade de uma implementação simples e que deve ter fácil manutenibilidade de código, a implementação padrão do algoritmo do Quick Sort pode ser mais interessante que as outras devido a sua simplicidade e eficiência.

Portanto, entre os três não há um resultado claro, pois a escolha do algoritmo ideal irá depender do contexto da aplicação. Considerando o contexto do trabalho que busca por uma performance ideal e constante a fim de poder comparar o Quick Sort com os próximos algoritmos de ordenação, podemos escolher o Quick Sort com Mediana, pois ele evitará resultados distorcidos que podem ocorrer no pior caso. No entanto, é importante mencionar que qualquer um dos outros dois poderiam ser igualmente escolhidos para realizar os testes da próxima parte do trabalho.

5.2. Parte 2: Quick Sort vs Merge Sort vs Heap Sort

Agora que já selecionamos um algoritmo do Quick Sort ideal para os testes do trabalho, nós podemos compará-lo com outros dois algoritmos: Merge Sort e Heap Sort. Ao realizar testes com 5 sementes diferentes para cada um dos tamanhos, obtivemos os seguintes resultados médios:

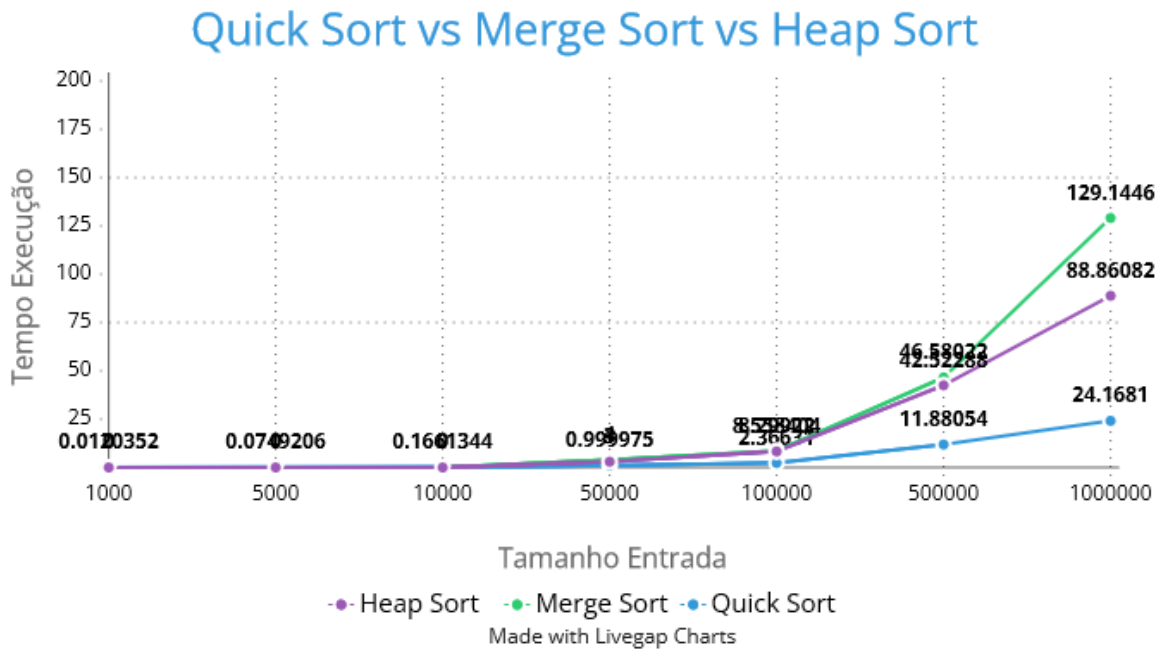
Quick Sort Mediana de Três							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Comparações	13997	82848,6	182643,4	1054488,2	2307984,6	12856932,6	27208479,6
Atribuições	8637,8	52169	111055,8	638333,2	1338427,8	7520951,4	15757644,4
Tempo	0,0120352	0,0749206	0,1661344	0,999975	2,36631	11,88054	24,1681

Merge Sort							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Comparações	8709,8	55247,6	120462	718219,4	1536249,4	8837435,4	18674496,4
Atribuições	19952	123616	267232	1568928	3337856	18951424	39902848
Tempo	0,0599134	0,467899	0,791584	4,036328	8,558404	46,58022	129,1446

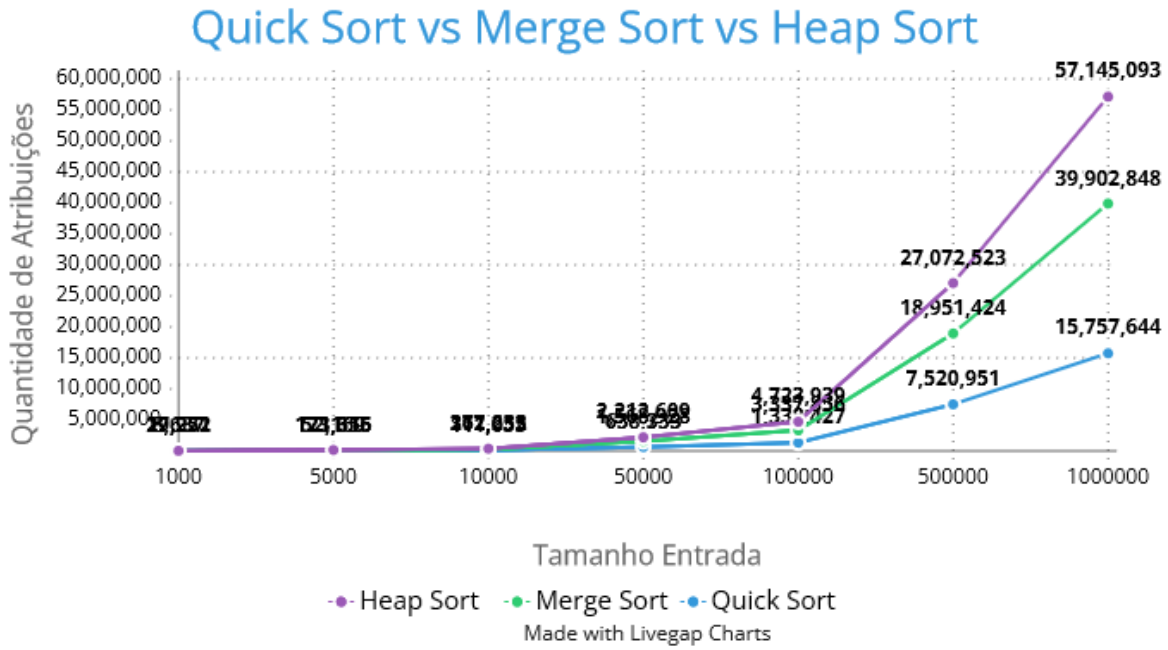
Heap Sort							
Tamanho	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000

Comparações	19181,2	119236,8	258436,8	1525073,2	3249292,8	18548348,8	39096729,2
Atribuições	27271,8	171355,2	372655,2	2212609,8	4723939,2	27072523,2	57145093,8
Tempo	0,0419704	0,2867028	0,6057374	3,953052	8,29922	42,52288	88,86082

Comparando o tempo de execução de cada um dos três algoritmos, é possível notar como o Quick Sort consegue escalar melhor em relação aos outros para entradas maiores:



Assim, entre os três algoritmos, para um contexto geral, o Quick Sort com Mediana de Três seria o mais interessante. Essa grande diferença entre os algoritmos de ordenação pode ser explicada pela quantidade de atribuições. O Quick Sort realiza trocas com muito menos frequência que os outros dois algoritmos, trocando apenas quando necessário. Já o Heap Sort e o Merge Sort realizam trocas e atribuições frequentemente, pois o heap precisa ser reajustado várias vezes e o Merge precisa atribuir os valores em ordem ao novo vetor:



Portanto, uma vez que o Quick Sort consegue limitar as trocas e atribuições para serem feitas apenas quando necessário, o seu tempo de execução consegue ser bem menor em comparação aos outros, obtendo um desempenho melhor entre os três.

Observação: Contudo, existem situações em que os outros dois algoritmos podem ser mais interessantes. O Merge Sort não apresenta o melhor desempenho para ordenação de arrays em memória primária. No entanto, ele consegue ser um algoritmo mais eficiente para realizar ordenações de grandes dados em memória secundária ou de listas encadeadas. Já o Heap Sort é um algoritmo capaz de impedir totalmente o pior caso do Quick Sort, uma vez que o seu custo nessa situação é O(nlog(n)), além de não exigir memória extra como o Merge Sort.

6. Conclusões

Neste trabalho, nós comparamos diferentes tipos de algoritmos de ordenação. Segundo os resultados obtidos no ambiente computacional testado, foi possível notar que os aprimoramentos realizados no Quick Sort conseguem arrumar diferentes pontos negativos do algoritmo sem apresentar um aumento grande no custo computacional de tempo. Dentre eles, o Quick Sort com Mediana de Três e o Quick Sort Iterativo Inteligente foram os aprimoramentos que tiveram os melhores desempenhos. Além disso, ao comparar o Quick Sort com Mediana de Três com o Merge Sort e o Heap Sort, foi possível notar que Quick Sort tem a melhor performance entre os três à medida que o tamanho das entradas aumenta. Também, foi possível aprender por meio deste trabalho que um algoritmo pode ser mais adequado que outro a depender do contexto em que será utilizado. Assim, é importante avaliar o contexto da aplicação como um todo a fim de escolher o algoritmo que melhor atende os requisitos dela.

7. Bibliografia

Todos os dados utilizados para criação das tabelas e dos gráficos podem ser acessados neste link: [ED] Média Desempenho Algoritmos Ordenação.

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

C Programing. Sorting Algorithm Comparison. Disponível em: <https://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html>. Acesso em: 07/11/2022.

Geeks for Geeks. Iterative Quick Sort. Disponível em: <https://www.geeksforgeeks.org/iterative-quick-sort/>. Acesso em: 07/11/2022.

Geeks for Geeks. Can QuickSort be implemented in O(nLogn) worst case time complexity?. Disponível em: <https://www.geeksforgeeks.org/can-quicksort-implemented-onlogn-worst-case-time-complexity/> Acesso em: 07/11/2022.

Geeks for Geeks. K'th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time). Disponível em: <https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/>. Acesso em: 07/11/2022.

Geeks for Geeks. QuickSort Tail Call Optimization (Reducing worst case space to Log n). Disponível em: <https://www.geeksforgeeks.org/quicksort-tail-call-optimization-reducing-worst-case-space-log-n/>. Acesso em: 07/11/2022.

Happycoders. Comparing All Quicksort Optimizations. Disponível em: <https://www.happycoders.eu/algorithms/quicksort/#Comparing_All_Quicksort_Optimizations>. Acesso em: 07/11/2022.

IME USP. Ordenação: Quicksort. Disponível em: <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/quick.html> Acesso em: 07/11/2022.

Opengenus. Time and Space complexity of Quick Sort. Disponível em: <https://iq.opengenus.org/time-and-space-complexity-of-quick-sort/>. Acesso em: 07/11/2022.

ShardeCourse. HW3 Solution: Disponível em: <https://www.sharecourse.net/sharecourse/upload/person/3085/files/ch7%20solution.pdf>. Acesso em: 07/11/2022.

Wikipedia. Median of medians. Disponível em: <https://en.wikipedia.org/wiki/Median_of_medians>. Acesso em: 07/11/2022.

Wikipedia. Quick Sort Space complexity. Disponível em: <https://en.wikipedia.org/wiki/Quicksort#Space_complexity>. Acesso em: 07/11/2022

Apêndice 1. Instruções para Compilação

Para executar o código do trabalho e disponibilizado no github, você deve realizar os seguintes passos em qualquer terminal de preferência:

1. Acesse a pasta raiz (UFMG-ED-Algoritmos-Ordenacao) do projeto, onde está presente o Makefile.
2. Execute o comando make.
3. Acesse a pasta “bin” do projeto.
4. Inicie o executável “run” considerando as seguintes flags:
    - **4.1. [Obrigatória]** -i “caminho/nome_arquivo_entrada.txt”
        - **4.1.1.** Informe o caminho e nome do arquivo de entrada que contém a quantidade de casos de teste e o tamanho de cada um em linhas separadas.
    - **4.2. [Obrigatória]** -o “caminho/nome_saida.txt”
        - **4.2.1.** Informe o caminho e nome do arquivo resultante que irá conter as estatísticas de desempenho da execução atual.
    - **4.3. [Opcional]** -p [“caminho/nome_arquivo_log.out”]
        - **4.3.1.** Ao informar está flag, você pode definir o caminho e nome do arquivo “.OUT” gerado no fim da execução.
        - **4.3.2.** Por padrão será criado um arquivo chamado “log.out” no mesmo local do executável.
    - **4.4. [Opcional]** -r
        - **4.4.1.** Está flag define qual das duas partes do trabalho será executada. Os possíveis valores para ela são:
            - 4.4.1.1. **1:** Executa os algoritmos de ordenação da primeira parte do trabalho.
            - 4.4.1.2. **2:** Executa os algoritmos de ordenação da segunda parte do trabalho.
        - **4.4.2.** Se a flag não for informada, o programa irá assumir por padrão que deverá executar a primeira parte do trabalho.
    - **4.5. [Obrigatória]** -v
        - **4.5.1.** Está flag define qual algoritmo de ordenação será utilizado. Os valores variam de acordo com a flag “-r” informada.
            - 4.5.1.1. Se a flag “-r” não foi informada ou for 1, os valores abaixo executam os respectivos algoritmos de ordenação:
                - 4.5.1.1.1. **1:** Quick Sort

4.5.1.1.2. **2:** Quick Sort com Mediana

4.5.1.1.3. **3:** Quick Sort com Seleção

4.5.1.1.4. **4:** Quick Sort Iterativo

4.5.1.1.5. **5:** Quick Sort Iterativo Inteligente

4.5.1.2. Se a flag “-r” foi informada com o valor 2, os valores abaixo executam os respectivos algoritmos de ordenação

4.5.1.2.1. **1:** Quick Sort com Mediana de Três

4.5.1.2.2. **2:** Merge Sort

4.5.1.2.3. **3:** Heap Sort

4.5.1.3. Qualquer valor informado que esteja acima ou abaixo dos listados levará a um erro.

4.6. [Obrigatória] -s

4.6.1. Esta flag define qual a semente para geração de números aleatórios. Caso não seja informada, será exibido o seguinte erro:

4.6.1.1. “[ERRO] A semente para geracao aleatoria de numeros nao foi informada. Informe um valor por meio da flag: -s. Encerrando execucao.”.

4.7. [Semi-Obrigatório] -k

4.7.1. Esta flag **DEVE** ser informada caso você execute o algoritmo de ordenação do **Quick Sort com Mediana**.

4.7.2. O valor de “-k” representa a quantidade de pivôs que serão selecionados aleatoriamente para encontrar a mediana.

4.7.3. Caso a flag não seja informada para o valor de “-v 2”, o seguinte erro será exibido:

4.7.3.1. “[ERRO] A versao do quick sort com mediana (2) necessita de um valor de quantos elementos selecionar para a mediana. Informe um valor por meio da flag: -k. Encerrando execucao.”.

4.8. [Semi-Obrigatório] -m

4.8.1. Esta flag **DEVE** ser informada caso você execute o algoritmo de ordenação do **Quick Sort com Seleção**.

4.8.2. O valor de “-m” representa o tamanho inicial em que o algoritmo de seleção será chamado para ordenar o vetor restante.

4.8.3. Caso a flag não seja informada para o valor de “-v 3”, o seguinte erro será exibido:

4.8.3.1. “[ERRO] A versao do quick sort com selecao (3) necessita de uma valor para identificar quando iniciar a ordenacao por selecao. Informe um valor por meio da flag: -m. Encerrando execucao.”.

5. Após isso, o programa irá gerar um arquivo de texto com os resultados de desempenho de execução do algoritmo de ordenação escolhido no caminho informado pela flag “-o”.

- Exemplo de comando para executar Quick Sort Padrão:

```
./run -v 1 -s 2000 -i ./entrada.txt -o ./saida.txt;
```

- Exemplo de comando para executar Quick Sort com Mediana:

```
./run -v 2 -s 2000 -k 3 -i ./entrada.txt -o ./saida.txt;
```

- Exemplo de comando para executar Quick Sort com Seleção:

```
./run -v 3 -s 2000 -m 100 -i ./entrada.txt -o ./saida.txt;
```

- Exemplo de comando para executar Quick Sort Iterativo:

```
./run -v 4 -s 2000 -i ./entrada.txt -o ./saida.txt;
```

- Exemplo de comando para executar Quick Sort Iterativo Inteligente:

```
./run -v 5 -s 2000 -i ./entrada.txt -o ./saida.txt;
```

- Exemplo de comando para executar Merge Sort:

```
./run -r 2 -v 2 -s 2000 -i ./entrada.txt -o ./saida.txt;
```

- Exemplo de comando para executar Heap Sort:

```
./run -r 2 -v 3 -s 2000 -i ./entrada.txt -o ./saida.txt;
```