FCT – Faculdade de Ciências e Tecnologia
DMC – Departamento de Matemática e Computação
Bacharelado em Ciência da Computação

LUCAS GONÇALVES FERREIRA
MATHEUS MAGALHÃES DA ROCHA

Projeto e Análise de Algoritmos

Trabalho Algoritmos de Ordenação

Presidente Prudente 2021

Sumário:

1. Introdução

O presente trabalho de Análise de Algoritmos é um relatório da disciplina Projeto e Análise de Algoritmos, listando o tempo de execução de diferentes algoritmos de ordenação(após a implementação dos mesmos em uma linguagem escolhida pela equipe) e comparando com a análise assintótica dos mesmos.

Resumidamente, algoritmos que ordenam um conjunto, geralmente representado por um vetor ou lista de dados, são chamados de algoritmos de ordenação. Tais algoritmos podem ordenar completamente ou parcialmente os elementos presentes no vetor em questão, de forma ascendente ou descendente. A eficiência, vantagens e uso de cada algoritmo depende do projeto e é claro do tempo,complexidade e espaço em memória ocupado pelo algoritmo.

Alguns dos aspectos e características da implementação, bem como a velocidade e complexidade dos algoritmos em questão serão observados ao longo do trabalho, de forma a explicitar o mais detalhadamente possível a relação entre os dados obtidos na experimentação e a análise assintótica dos mesmos.

2. Especificações Técnicas e Contextualização do Trabalho

Esta seção conceitua as especificações do computador usado nos testes e algumas informações a respeito do processo de experimentação e do trabalho como um todo.

De forma geral, para cada algoritmo de ordenação proposto, foram realizados 5 testes onde foram captadas as informações do tempo de execução para as três entradas diferentes(um vetor com números gerados de forma aleatória, um vetor com número em ordem crescente e um último em ordem decrescente). Os tamanhos dos vetores usados no experimento foram: 1000, 5000, 10000, 15000, 20000, 25000 e 50000.

Os mesmos valores das entradas foram usados para todos os algoritmos, por exemplo, o vetor aleatório de 1000 elementos era o mesmo para todos, de forma a manter a experimentação o mais justa possível.

O fato de terem sido feitos múltiplos testes foi para evitar certos erros derivados do sistema, e manter os resultados mais fiéis possíveis. Alguns algoritmos também executavam bem rápido, então tinha-se a necessidade de testar várias vezes, pois tal fato dificultava a captura do intervalo de tempo.

Por fim, durante a apresentação dos resultados, cada algoritmo teve seus resultados apresentados por 5 tabelas diferentes informando o tempo de execução em ms(milissegundos) em cada um dos testes. Ao final também foi apresentado uma tabela com os valores médios dos 5 testes e um gráfico comparativo(baseado nesta última tabela) com o tempo de execução dos vetores crescente, decrescente e aleatório.

No capítulo 3 são apresentados os resultados obtidos por meio dos testes e é feita uma comparação com a análise assintótica. No capítulo 4, os diferentes resultados obtidos pelos algoritmos são discutidos de forma mais geral.

- **Especificações da Máquina de Teste:**

A máquina usada durante as experimentações foi um notebook Acer, cuja informações/especificações estão listadas na tabela abaixo:

Modelo	Aspire A315-53-32U4.
Processador	i3 7th gen 7020u.
Memória RAM	4gb de ram.
HD	HD 1 TB para Notebook Western Digital - WD10SPZX.
Linguagem de Programação utilizada	Java.
IDE utilizada	Netbeans(versão 12.0).
Sistema Operacional	Windows 10(versão mais atualizada) 64-bits.

3. Experimento e Análise dos Algoritmos

3.1. Algoritmos de Ordenação Baseados em Troca

Nesta seção serão abordados os algoritmos de ordenação baseados em troca: Bubble Sort(versão original e melhorada) e Quick Sort(com pivô no início e pivô no centro).

3.1.1. Bubble Sort(versão original)

Começando por um dos algoritmos de ordenação mais famosos e considerado por muitos como um dos mais intuitivos, temos a versão original do Bubble Sort.
A ideia básica deste algoritmo é percorrer o vetor várias vezes, e a cada iteração, comparar cada elemento com seu sucessor, trocando-os de lugar caso estejam na ordem incorreta.

Abaixo estão listados os resultados dos 5 testes realizados com o algoritmo do Bubble Sort, para diferentes entradas e diferentes tamanhos de entrada. Obs: os resultados estão em ms(milissegundos).

Bubble Sort - Teste 1							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	13	18	24	58	98	136	551
Decrescente	7	36	102	236	411	670	2778
Aleatório	1	38	171	419	776	1240	5256

Bubble Sort - Teste 2							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	35	25	57	97	154	635
Decrescente	1	25	102	230	411	640	2695
Aleatório	1	35	171	416	774	1243	5961

Bubble Sort - Teste 3							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	7	28	57	97	166	620
Decrescente	1	25	102	230	418	635	2567
Aleatório	2	36	178	418	790	1445	5120

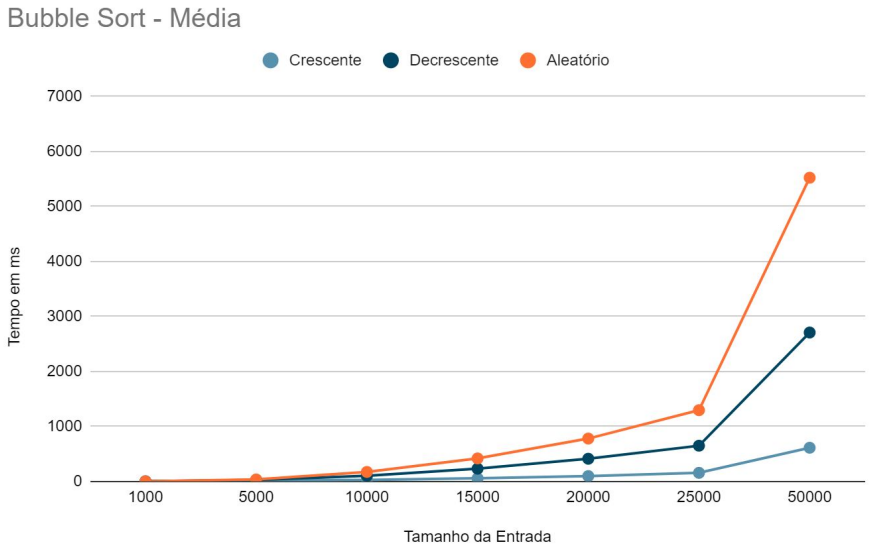
Bubble Sort - Teste 4							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	7	30	55	99	164	635
Decrescente	1	25	103	232	411	655	2924
Aleatório	1	35	173	421	779	1273	6075

Bubble Sort - Teste 5							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	6	25	55	97	162	619
Decrescente	1	25	102	232	411	646	2567
Aleatório	1	36	170	418	777	1273	5185

Com os resultados dos 5 testes, foi possível a criação de uma tabela com os valores médios de cada vetor e tamanho de entrada respectivo:

Bubble Sort - Média							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	2.6	14.6	26.4	56.4	97.6	156.4	612
Decrescente	2.2	27.2	102.2	232	412.4	649.2	2706.2
Aleatório	1.2	36	172.6	418.4	779.2	1294.8	5519.4

Também podemos ver a diferença do tempo de execução entre as entradas no gráfico comparativo abaixo:



De forma geral, o algoritmo de ordenação Bubble Sort por apresentar dois laços de repetição encadeados em seu código, acaba tendo uma complexidade da ordem $O(n^2)$ independente se o vetor está ordenado de forma crescente, decrescente ou mesmo se os dados estão dispostos de forma aleatória. portanto, a complexidade em seu pior caso,caso médio e melhor caso é sempre $O(n^2)$.
Mesmo a complexidade sendo de ordem $O(n^2)$ para qualquer entrada, ainda foi possível perceber uma diferença entre os tempos das entradas crescente, decrescente e aleatória. De acordo com a hipótese pensada pelo grupo, a entrada crescente apresenta tempo de execução menor, pelo fato de não ocorrerem operações de atribuição(o swap dentro do if), já que os elementos estão todos ordenados de forma crescente, o que faz o algoritmo não executar os passos dentro do if(a comparação vai ser falsa e o algoritmo vai para o próximo número do vetor).
Por outro lado, foi uma surpresa para o grupo a entrada decrescente não ter o maior tempo de execução nos testes,já que nela ocorrerá comparações e swap em praticamente em todas as iterações, enquanto na aleatória não será em todas.
Obs: de acordo com o professor Danilo Eler, uma hipótese para tentar justificar essa questão, pode ser alguma técnica ou fator relacionado com a memória cache na CPU, já que enquanto no vetor de entrada decrescente a comparação é sempre com o mesmo valor, no aleatório ocorre uma mudança no número que é comparado.

3.1.2. Bubble Sort(melhorado)

Esta é uma versão melhorada do algoritmo anterior, com o objetivo de evitar iterações desnecessárias. Em um simples resumo, basicamente este código apresenta uma verificação adicional para se saber se o vetor já está ordenado de forma crescente ou não.

Abaixo estão listados os resultados dos 5 testes realizados com o algoritmo do Bubble Sort(melhorado), para diferentes entradas e diferentes tamanhos de entrada.
Obs: os resultados estão em ms(milissegundos).

Bubble Sort(melhorado) - Teste 1							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	1
Decrescente	15	25	104	233	415	326	1285
Aleatório	3	72	173	428	801	1403	6608

Bubble Sort(melhorado) - Teste 2							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	0
Decrescente	0	32	103	235	414	649	2727
Aleatório	1	38	179	427	794	1301	6226

Bubble Sort(melhorado) - Teste 3							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	0
Decrescente	1	26	104	234	416	633	2590
Aleatório	1	38	174	426	795	1380	5257

Bubble Sort(melhorado) - Teste 4							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	0
Decrescente	0	25	104	235	416	669	2667
Aleatório	1	34	178	430	800	1336	5534

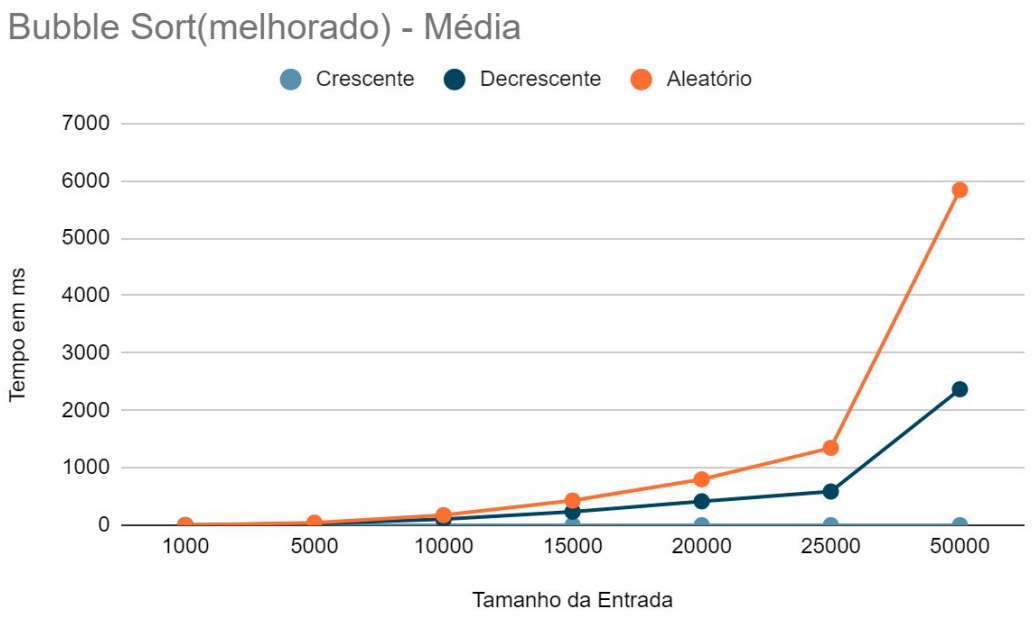
Bubble Sort(melhorado) - Teste 5							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	0
Decrescente	0	26	103	234	416	652	2568

Aleatório	1	35	173	432	800	1310	5579

Com os resultados dos 5 testes, foi possível a criação de uma tabela com os valores médios de cada vetor e tamanho de entrada respectivo:

Bubble Sort(melhorado) - Média							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	0.2
Decrescente	3.2	26.8	103.6	234.2	415.4	585.8	2367.4
Aleatório	1.4	43.4	175.4	428.6	798	1346	5840.8

Também podemos ver a diferença do tempo de execução entre as entradas no gráfico comparativo abaixo:



Diferente de sua versão mais simples, nesse algoritmo melhorado podemos ver algumas mudanças.
Basicamente, graças a verificação adicionada, o tempo de execução do algoritmo para vetores crescentes caiu muito(o que é visível no gráfico acima), enquanto que os tempos para os vetores decrescente e aleatório permaneceram os mesmos.
Portanto, diferente do algoritmo anterior, o Bubble Sort melhorado apresenta pior caso e caso médio $\Theta(n^2)$, e melhor caso $\Theta(n)$, ou seja, o melhor caso é o vetor ser crescente(já estar todo ordenado), o que vai fazer o algoritmo percorrê-lo apenas uma única vez.

3.1.3. Quick Sort(pivô no início)

Outro também famoso algoritmo de ordenação é o quick sort(trocas de elementos distantes são mais efetivas), considerado um melhoramento do Bubble Sort.
O algoritmo se baseia na ideia de dividir para conquistar, onde um vetor é dividido em dois vetores menores que serão ordenados independentemente e combinados para produzir o resultado final. Alguns passos importantes nesse processo são: escolher um elemento do vetor como pivô, reordenar os elementos para que os à esquerda do pivôs sejam menores que ele, e os à direita sejam maiores que ele,e por fim, remanejar cada vetor menor/subvetor(realizar os mesmos passos no vetores menores) até o processo estar completo.
Obs: Veremos mais à frente como a escolha do vetor influenciará nos tempos registrados.

Abaixo estão listados os resultados dos 5 testes realizados com o algoritmo do Quick Sort, para diferentes entradas e diferentes tamanhos de entrada. Obs: os resultados estão em ms(milissegundos).

Quick Sort(pivô início) - Teste 1							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	12	29	94	153	274	333	1567
Decrescente	4	29	80	256	449	462	1306
Aleatório	0	2	1	1	2	10	14

Quick Sort(pivô início) - Teste 2							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	24	67	165	281	562	2453
Decrescente	1	21	113	278	450	485	3265
Aleatório	0	0	1	1	2	2	6

Quick Sort(pivô início) - Teste 3							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	23	67	156	274	564	2256
Decrescente	0	22	112	260	539	781	2811
Aleatório	0	0	1	1	2	2	6

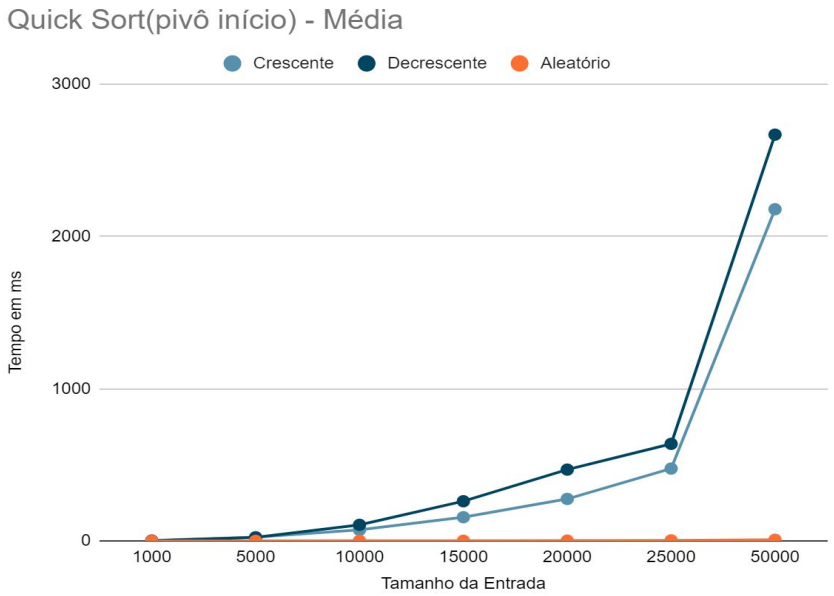
Quick Sort(pivô início) - Teste 4							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	22	68	153	273	461	2326
Decrescente	1	20	113	256	450	742	3021
Aleatório	0	0	1	1	2	3	6

Quick Sort(pivô início) - Teste 5							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	23	67	153	278	457	2292
Decrescente	1	25	111	254	455	719	2941
Aleatório	0	0	1	1	2	2	6

Com os resultados dos 5 testes, foi possível a criação de uma tabela com os valores médios de cada vetor e tamanho de entrada respectivo:

Quick Sort(pivô início) - Média							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	3.2	24.2	72.6	156	276	475.4	2178.8
Decrescente	1.4	23.4	105.8	260.8	468.6	637.8	2668.8
Aleatório	0	0.4	1	1	2	3.8	7.6

Também podemos ver a diferença do tempo de execução entre as entradas no gráfico comparativo abaixo:



Uma informação antes de continuar é que o Quick Sort foi implementado em sua versão iterativa(usando uma pilha auxiliar), e não na versão recursiva a qual a maior parte das pessoas estão familiarizadas. Isso se deu em razão da enorme quantidade de elementos acabar estourando a stack/memória pelo grande número de recursões realizadas.
Voltando ao assunto, o algoritmo Quick Sort com pivô no início(primeira posição do vetor), apresenta relativamente um bom desempenho, embora pudesse ser melhor, usando algum outro pivô além do elemento na primeira posição(veremos os resultados do pivô central mais adiante).
Ambos melhor caso e caso médio apresentam mesma complexidade Θ(n log n), que ocorre quando o particionamento gera 2 subconjuntos de tamanho n/2(ou perto disso). Podemos ver o tempo de execução do vetor aleatório, por exemplo.
O pior caso ocorre quando a lista acaba sendo dividida de forma desbalanceada, ou seja, duas sub listas uma com tamanho 0 e outra com tamanho n - 1(ou perto disso). A complexidade nesse caso seria Θ(n^2), e um exemplo disso, pelo menos nesse caso com pivô no ínicio, seria quando a lista já está ordenada, ou inversamente ordenada(como podemos ver nos resultados dos testes).

3.1.4. Quick Sort(pivô no centro)

Continuando o assunto a respeito de Quick Sort, agora temos o mesmo algoritmo anterior porém com uma pequena diferença, o pivô não é mais o elemento inicial, ele agora é o elemento central da lista. A partir disso, veremos como nossos resultados sofrerão uma grande alteração.

Abaixo estão listados os resultados dos 5 testes realizados com o algoritmo do Quick Sort(pivô central), para diferentes entradas e diferentes tamanhos de entrada.
Obs: os resultados estão em ms(milissegundos).

Quick Sort(pivô centro) - Teste 1							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	2	0	0	0	7	20
Decrescente	0	1	0	0	0	2	5
Aleatório	0	1	1	1	2	3	7

Quick Sort(pivô centro) - Teste 2							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	1	4
Decrescente	0	0	0	0	0	1	3
Aleatório	0	0	1	1	2	3	5

Quick Sort(pivô centro) - Teste 3							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	2	2
Decrescente	0	0	0	0	0	2	1
Aleatório	0	0	1	1	2	5	5

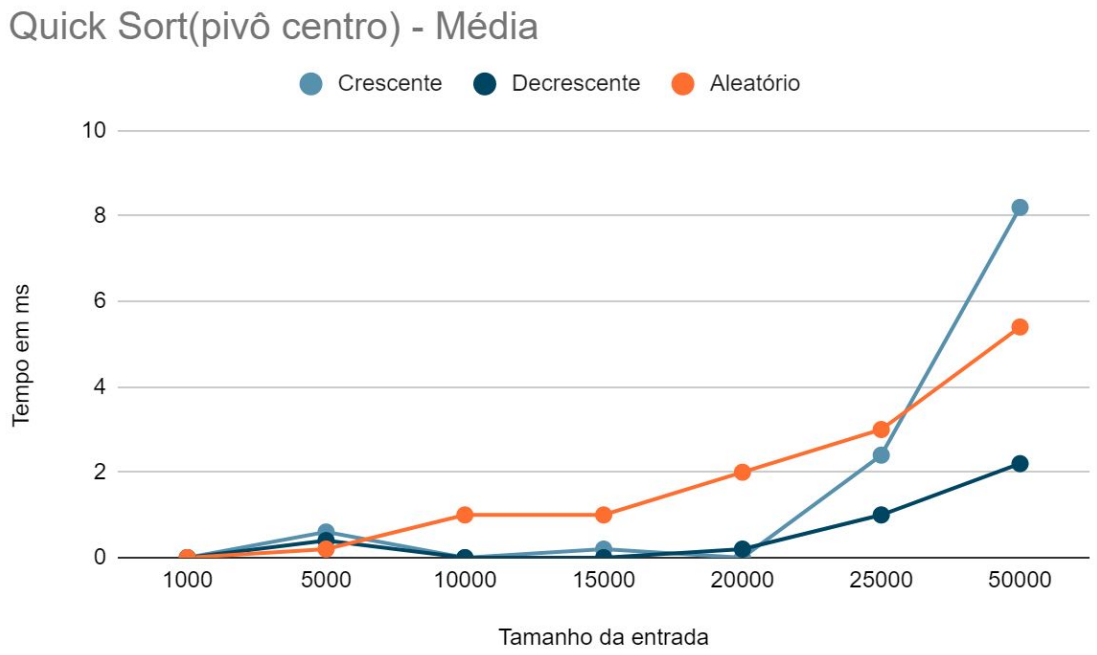
Quick Sort(pivô centro) - Teste 4							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	1	0	0	0	1	2
Decrescente	0	1	0	0	0	0	1
Aleatório	0	0	1	1	2	2	5

Quick Sort(pivô centro) - Teste 5							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	1	0	1	13
Decrescente	0	0	0	0	1	0	1
Aleatório	0	0	1	1	2	2	5

Com os resultados dos 5 testes, foi possível a criação de uma tabela com os valores médios de cada vetor e tamanho de entrada respectivo:

Quick Sort(pivô centro) - Média							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0.6	0	0.2	0	2.4	8.2
Decrescente	0	0.4	0	0	0.2	1	2.2
Aleatório	0	0.2	1	1	2	3	5.4

Também podemos ver a diferença do tempo de execução entre as entradas no gráfico comparativo abaixo:



Embora a complexidade se mantenha a mesma, ou seja, melhor e médio caso $\Theta(n \log n)$, enquanto pior caso $\Theta(n^2)$, fica evidente que o tempo de execução do Quick Sort com pivô central foi bem menor do que sua outra versão com pivô no início. Enquanto naquele caso, ambos vetores crescente e decrescente acabavam fazendo o algoritmo cair no pior caso(o tempo de execução do algoritmo nesses vetores era bem longo), aqui isso não acontece, e todos os três vetores apresentam um bom tempo de execução.
O objetivo de utilizar diferentes pivôs(no início,fim,central,aleatória) é fugir dos piores casos a todo custo, embora nem sempre isso acabe sendo possível.

Mesmo que o algoritmo acima apresente um bom tempo de execução não se pode esquecer que ainda existe chance de cair no pior caso dependendo de como estiver organizado o vetor, afinal, o algoritmo quick sort em si é considerado instável/imprevisível justamente por isso.
Obs: alguns até recomendam escolher um pivô aleatório para evitar ao máximo a chance de se cair em um pior caso, mas até assim ainda não é totalmente garantido.

3.2. Algoritmos de Ordenação Baseados em Inserção

Nesta seção serão abordados os algoritmos de ordenação baseados em inserção: Insertion Sort e Shell Sort.

3.2.1. Insertion Sort

Entrando agora nos algoritmos baseados em Inserção, falaremos a respeito do Insertion Sort, um dos mais básicos algoritmos desse tipo.
Em um simples resumo, o Insertion Sort ou Inserção Simples, tem como ideia básica inserir um dado elemento em sua posição correta em um conjunto já ordenado, em outras palavras, ordenar um conjunto inserindo seus elementos em um subconjunto já ordenado.

Abaixo estão listados os resultados dos 5 testes realizados com o algoritmo do Insertion Sort, para diferentes entradas e diferentes tamanhos de entrada. Obs: os resultados estão em ms(milissegundos).

Insertion Sort - Teste 1							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	1	9
Decrescente	9	16	27	58	105	174	702
Aleatório	2	9	13	27	50	86	345

Insertion Sort - Teste 2							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	0
Decrescente	0	5	27	58	106	162	683
Aleatório	2	2	12	28	50	83	336

Insertion Sort - Teste 3							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	0
Decrescente	0	6	27	64	106	169	674
Aleatório	0	2	11	31	51	94	328

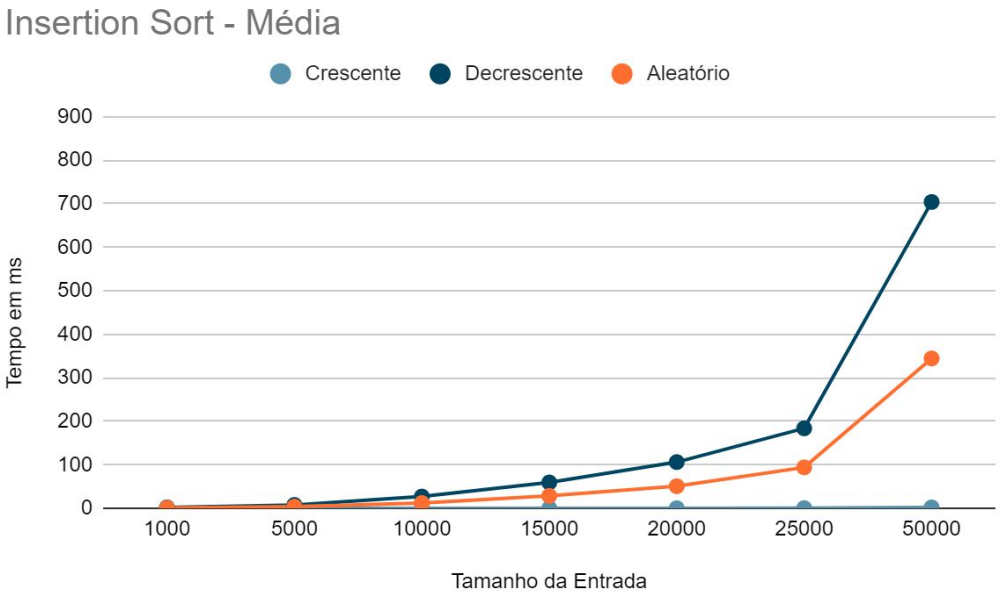
Insertion Sort - Teste 4							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	0
Decrescente	0	5	28	58	105	248	729
Aleatório	0	3	12	28	51	122	365

Insertion Sort - Teste 5							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	0
Decrescente	0	6	25	58	108	165	731
Aleatório	0	2	12	28	51	84	347

Com os resultados dos 5 testes, foi possível a criação de uma tabela com os valores médios de cada vetor e tamanho de entrada respectivo:

Insertion Sort - Média							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0.2	1.8
Decrescente	1.8	7.6	26.8	59.2	106	183.6	703.8
Aleatório	0.8	3.6	12	28.4	50.6	93.8	344.2

Também podemos ver a diferença do tempo de execução entre as entradas no gráfico comparativo abaixo:



Um fato que podemos perceber ao fazer uma rápida comparação com os outros algoritmos é que o Insertion Sort faz menos comparações do que o Bubble Sort, já que a parte ordenada não é comparada novamente a cada iteração.
Além disso, embora simples, ambos os códigos são bem famosos por sua intuitividade e funcionam como porta de entrada para muitos outros algoritmos.
De forma geral, o Insertion Sort apresenta uma complexidade $\Theta(n)$ no melhor caso, ou seja, quando o vetor está ordenado, e uma complexidade $\Theta(n^2)$ no pior(um vetor inversamente ordenado) e médio caso.
Com base no gráfico, fica evidente que o algoritmo apresenta um tempo de execução menor no vetor ordenado/crescente, enquanto nos outros vetores ele apresenta um tempo de execução bem superior, o que confirma as complexidades apresentadas anteriormente.

3.2.2. Shell Sort

A Inserção simples ou Insertion Sort que vimos anteriormente é eficiente para arquivos/conjuntos quase ordenados.
A partir de agora veremos o Shell Sort, um algoritmo considerado como uma melhoria da inserção simples, cuja ideia básica se resume em dividir uma entrada em k sub-conjuntos e aplicar a inserção simples a cada um. Esse valor “k” é reduzido sucessivamente e a cada nova iteração, o vetor original fica mais perto de estar completamente ordenado.
Obs: esse código depende fortemente desse valor “k”, podendo produzir diferentes resultados dependendo da forma como o algoritmo foi construído.

Abaixo estão listados os resultados dos 5 testes realizados com o algoritmo do Shell Sort, para diferentes entradas e diferentes tamanhos de entrada. Obs: os resultados estão em ms(milissegundos).

Shell Sort - Teste 1							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	3	0	0	0	8	15
Decrescente	2	1	0	0	0	9	13
Aleatório	2	2	1	2	3	3	8

Shell Sort - Teste 2							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	0	0	0	0	0	1
Decrescente	2	0	0	0	0	0	1
Aleatório	1	0	1	2	3	4	8

Shell Sort - Teste 3							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	1
Decrescente	0	0	0	0	0	0	1
Aleatório	0	0	1	2	3	4	9

Shell Sort - Teste 4							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	1
Decrescente	0	0	0	0	0	0	1
Aleatório	0	0	1	2	3	4	9

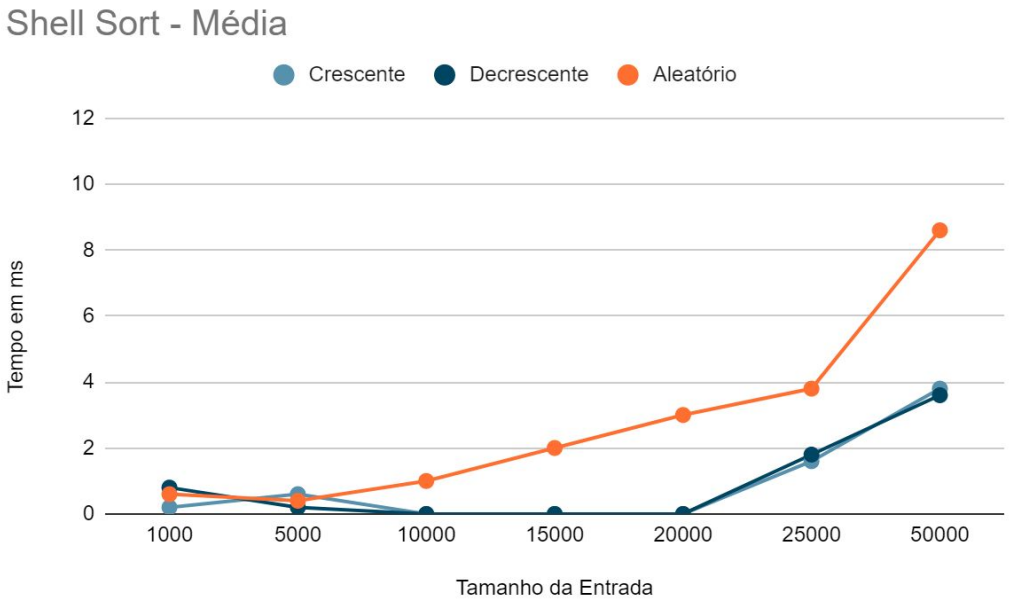
Shell Sort - Teste 5							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	0	0	0	0	1
Decrescente	0	0	0	0	0	0	2

Aleatório	0	0	1	2	3	4	9

Com os resultados dos 5 testes, foi possível a criação de uma tabela com os valores médios de cada vetor e tamanho de entrada respectivo:

Shell Sort - Média							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0.2	0.6	0	0	0	1.6	3.8
Decrescente	0.8	0.2	0	0	0	1.8	3.6
Aleatório	0.6	0.4	1	2	3	3.8	8.6

Também podemos ver a diferença do tempo de execução entre as entradas no gráfico comparativo abaixo:



Com base nos dados observados no gráfico e nos testes, pode-se dizer que o algoritmo Shell Sort é claramente bem rápido, apresentando um dos melhores e menores tempos de execução dentre todos os outros algoritmos.
Porém, tal algoritmo é considerado bem difícil de se calcular a ordem de complexidade, principalmente pelo fato de depender de um vetor de incrementos, cujos valores podem alterar totalmente o resultado.
Alguns teoremas específicos indicam complexidades para o algoritmo do Shell Sort:

Teorema1: Para a sequência de incrementos 1, 3, 7, 15, 31, 127, …, $2^k - 1$, o Shell Sort precisa de $O(n^{3/2})$ passos para ordenar uma sequência de tamanho n.

Teorema2: Para a sequência de incrementos 1,2, 3, 6, 8, 9, 12,16, …, $2^p 3^q$, o Shell Sort precisa de $O(n.log(n)^2)$ passos para ordenar uma sequência de tamanho n.
Esses são alguns teoremas usados para se descobrir a complexidade do algoritmo Shell Sort, embora existam muitos outros e cada um deles dependa de como está organizado o vetor de incrementos.

3.3. Algoritmos de Ordenação Baseados em Seleção

Nesta seção serão abordados os algoritmos de ordenação baseados em seleção: Selection Sort e Heap Sort.

3.3.1. Selection Sort

Entrando em outro campo de algoritmos de ordenação, finalmente chegamos aos algoritmos baseados em seleção. O Selection Sort ou Seleção direta é um dos mais simples e intuitivos algoritmos desta seção.
De forma geral, ele se baseia na ideia de selecionar um elemento e colocá-lo na posição correta no vetor. Essa ideia poderia ser resumida em três passos simples:
1 - selecionar o elemento de menor valor no vetor;
2 - trocar o elemento de lugar com o primeiro elemento do conjunto;
3 - repetir esse processo para todos os n-1 elementos restantes;

Abaixo estão listados os resultados dos 5 testes realizados com o algoritmo do Selection Sort, para diferentes entradas e diferentes tamanhos de entrada. Obs: os resultados estão em ms(milissegundos).

Selection Sort - Teste 1							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	11	12	154	350	619	110	478
Decrescente	11	27	155	348	617	968	4039
Aleatório	1	11	154	347	619	280	1077

Selection Sort - Teste 2

Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	41	154	350	622	280	1085
Decrescente	1	38	155	349	618	377	1672
Aleatório	1	38	155	347	618	280	1086

Selection Sort - Teste 3							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	41	155	362	619	314	1086
Decrescente	1	38	156	348	618	481	1622
Aleatório	1	40	155	348	617	393	1068

Selection Sort - Teste 4							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	38	157	349	620	286	1091
Decrescente	1	38	158	356	618	395	1609
Aleatório	1	38	154	349	618	285	1101

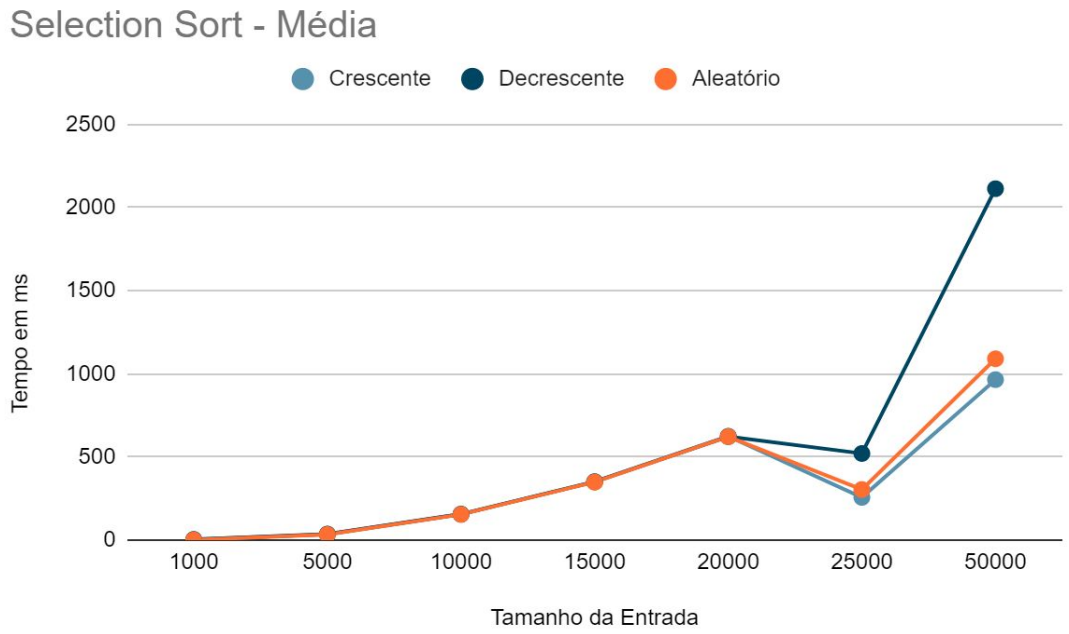
Selection Sort - Teste 5							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	39	153	348	642	293	1084
Decrescente	2	38	155	348	641	382	1624
Aleatório	1	38	153	351	636	282	1119

Com os resultados dos 5 testes, foi possível a criação de uma tabela com os valores médios de cada vetor e tamanho de entrada respectivo:

Selection Sort - Média							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	3	34.2	154.6	351.8	624.4	256.6	964.8
Decrescente	3.2	35.8	155.8	349.8	622.4	520.6	2113.2

Aleatório	1	33	154.2	348.4	621.6	304	1090.2

Também podemos ver a diferença do tempo de execução entre as entradas no gráfico comparativo abaixo:



Em resumo, o algoritmo Selection Sort apresenta complexidade Θ(n²) para todos os casos(pior, médio e melhor), e isso ocorre, justamente por causa do número de comparações envolvidas. Basicamente, enquanto no primeiro passo ocorrem n-1 comparações, no segundo ocorrem n-2, e assim por diante, o que configura uma complexidade quadrática para o algoritmo.
O fato de um vetor estar ordenado ou desordenado não causa nenhuma melhoria no tempo de execução do algoritmo, além disso, embora o algoritmo seja considerado melhor que o Bubble Sort, ele é acaba sendo útil apenas quando n é pequeno(quando o tamanho do vetor é pequeno).

3.3.2. Heap Sort

O Heap Sort trabalha de forma similar ao Selection Sort, porém, é utilizada uma estrutura de dados chamada Heap e através dessa estrutura o custo para recuperar o menor elemento é reduzido. Sua abordagem consiste em construir um heap máximo, trocar o primeiro elemento com o último, diminuir o heap em 1, e caso necessário o heap será rearranjado e esse procedimento será repetido n-1 vezes.

Abaixo estão listados os resultados dos 5 testes realizados com o algoritmo do Heap Sort, para diferentes entradas e diferentes tamanhos de entrada. Obs: os resultados estão em ms(milissegundos).

Heap Sort - Teste 1							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	2	1	1	2	10	15
Decrescente	0	1	1	1	2	3	9
Aleatório	0	1	2	2	2	4	8

Heap Sort - Teste 2							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	1	1	1	2	4	8
Decrescente	0	0	1	1	2	4	10
Aleatório	0	0	1	2	2	5	10

Heap Sort - Teste 3							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	1	1	1	2	6	7
Decrescente	0	1	1	1	2	4	5
Aleatório	0	1	1	2	2	4	8

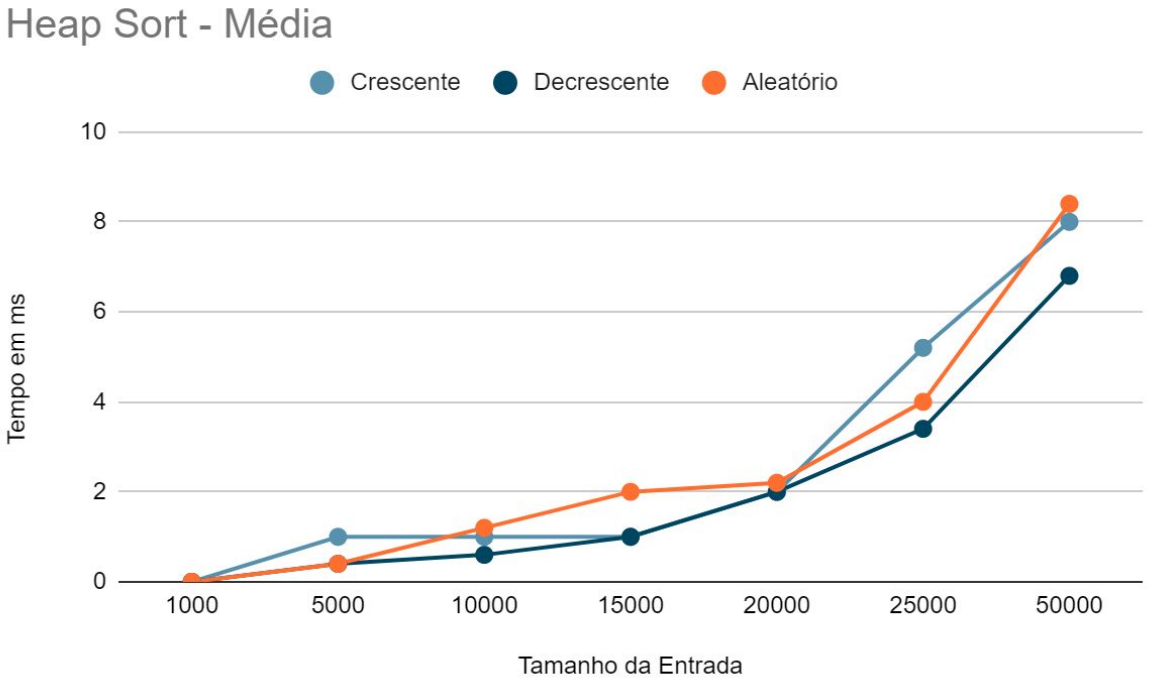
Heap Sort - Teste 4							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	1	1	1	2	4	5
Decrescente	0	0	0	1	2	4	5
Aleatório	0	0	1	2	2	4	8

Heap Sort- Teste 5							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	0	1	1	2	2	5
Decrescente	0	0	0	1	2	2	5
Aleatório	0	0	1	2	3	3	8

Com os resultados dos 5 testes, foi possível a criação de uma tabela com os valores médios de cada vetor e tamanho de entrada respectivo:

Heap Sort - Média							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	1	1	1	2	5.2	8
Decrescente	0	0.4	0.6	1	2	3.4	6.8
Aleatório	0	0.4	1.2	2	2.2	4	8.4

Também podemos ver a diferença do tempo de execução entre as entradas no gráfico comparativo abaixo:



O algoritmo analisado apresenta complexidade Θ(n log n) em todos os casos (pior, médio e melhor) devido ao procedimento para criar o heap ter complexidade Θ(log n) e esse procedimento ser repetido n vezes, então esse tipo de abordagem tem complexidade Θ(n log n), o que pode ser confirmado ao se observar o gráfico, onde as entradas crescente,decrescente e aleatória apresentam tempo de execução bem similar.
Além disso, fica claramente visível, o fato do algoritmo também ser um dos mais rápidos quando se observa seu pequeno tempo de execução.

3.4. Merge Sort

Por fim, mas não menos importante, temos o famoso algoritmo Merge Sort.
A ordenação por intercalação ou Merge Sort é um método que utiliza o princípio da divisão e conquista para fazer a ordenação. Um vetor será divido em duas partes e após estas partes serem ordenadas com o uso de um vetor auxiliar, elas serão intercaladas para formar o vetor ordenado.

Abaixo estão listados os resultados dos 5 testes realizados com o algoritmo do Merge Sort, para diferentes entradas e diferentes tamanhos de entrada. Obs: os resultados estão em ms(milissegundos).

Merge Sort - Teste 1							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	3	42	55	98	168	406	1207
Decrescente	3	30	100	85	157	270	1394
Aleatório	5	22	153	95	187	221	1113

Merge Sort - Teste 2							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	1	16	52	90	175	246	1135
Decrescente	6	23	42	84	167	254	1148
Aleatório	0	18	100	99	171	212	1133

Merge Sort - Teste 3							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	3	14	52	202	155	400	979
Decrescente	3	12	42	92	156	417	890
Aleatório	2	14	55	83	146	333	878

Merge Sort - Teste 4							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	3	13	46	99	158	272	980
Decrescente	0	11	36	85	162	233	1025

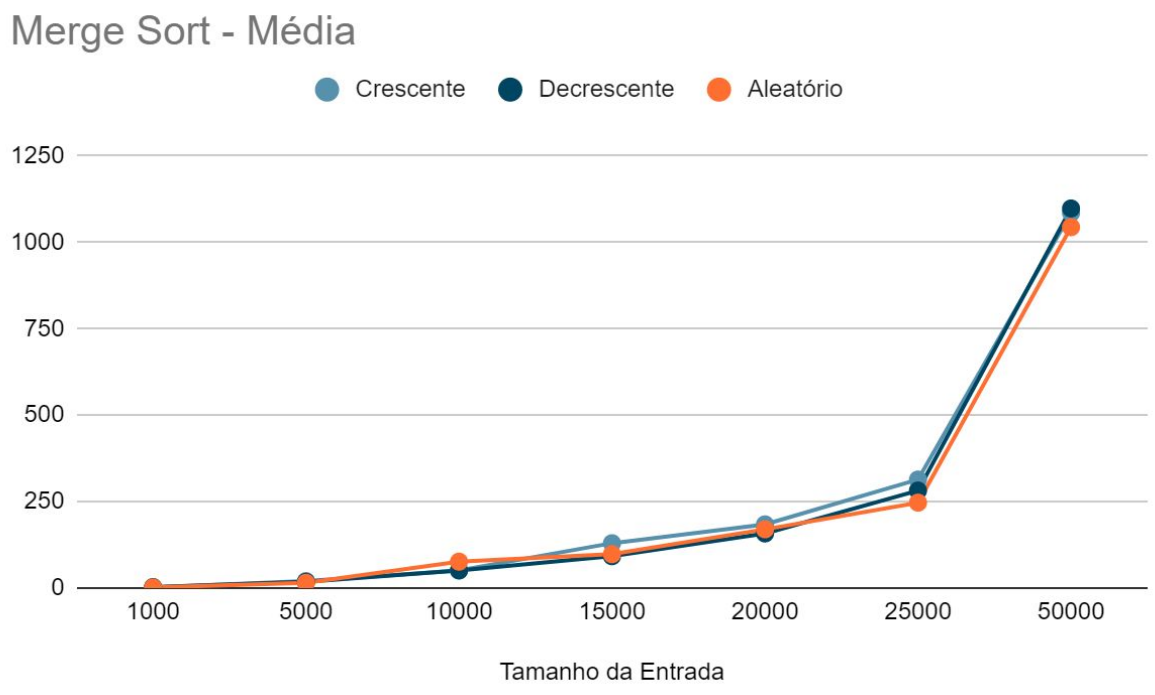
Aleatório	0	11	37	98	152	239	1166

Merge Sort - Teste 5							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	0	15	50	159	264	245	1125
Decrescente	1	12	34	114	147	236	1033
Aleatório	1	10	36	114	193	230	930

Com os resultados dos 5 testes, foi possível a criação de uma tabela com os valores médios de cada vetor e tamanho de entrada respectivo:

Merge Sort - Média							
Entrada/Tamanho	1000	5000	10000	15000	20000	25000	50000
Crescente	2	20	51	129.6	184	313.8	1085.2
Decrescente	2.6	17.6	50.8	92	157.8	282	1098
Aleatório	1.6	15	76.2	97.8	169.8	247	1044

Também podemos ver a diferença do tempo de execução entre as entradas no gráfico comparativo abaixo:



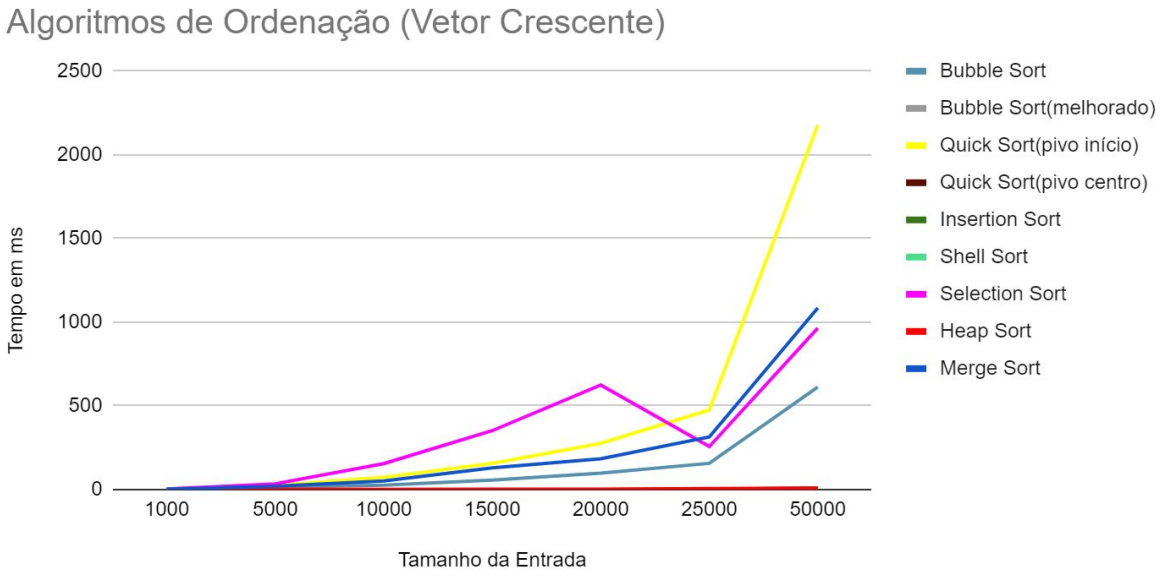
O Merge Sort foi implementado em sua versão iterativa(usando uma pilha auxiliar), e não na versão recursiva.
Isso se deu em razão da enorme quantidade de elementos acabar estourando a stack/memória pelo grande número de recursões realizadas.
O algoritmo analisado apresenta um desempenho estável, com complexidade Θ(n log n) em todos os casos (pior, médio e melhor) pois o Merge Sort divide o vetor em duas metades e leva tempo linear para fazer a mesclagem dessas duas metades.
Essa complexidade Θ(n log n) para todos os vetores/entradas fica ainda mais visível quando olhamos o gráfico, onde é possível ver como os valores das três entradas ficam próximos uns dos outros(variam pouco).

4. Comparações entre os algoritmos

Nesta seção iremos demonstrar o desempenho/tempo de execução dos algoritmos em cada tipo de entrada(crescente, decrescente, aleatória), de forma a deixar mais visível como cada algoritmo se saiu em relação aos outros algoritmos.
Obs: Os valores em questão são as médias dos teste de cada algoritmo.

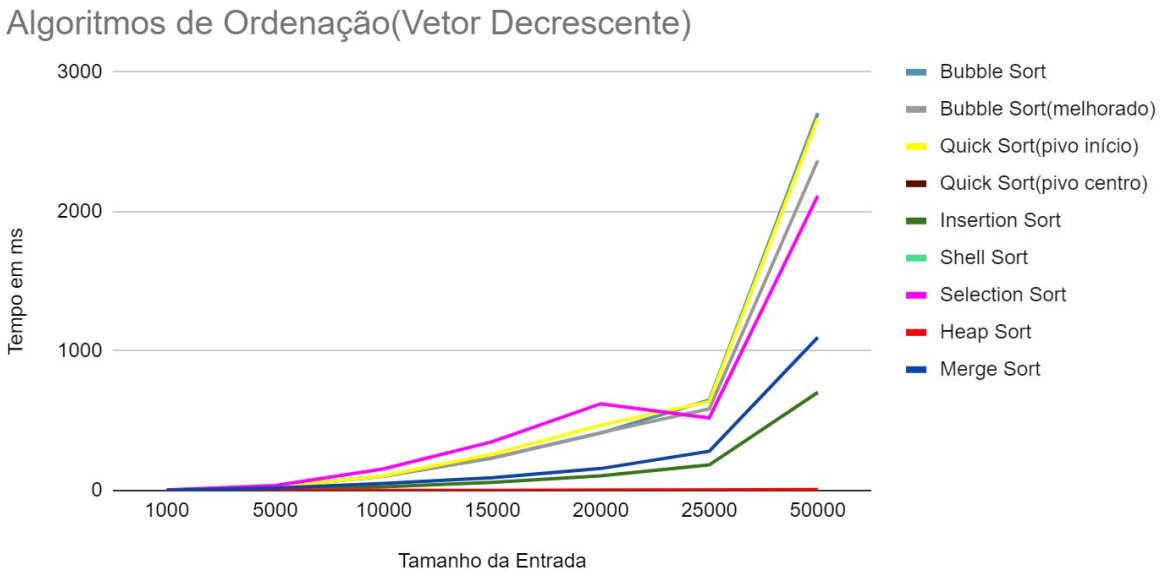
Primeiramente, vamos demonstrar os tempos de execução de cada algoritmo no vetor com números ordenados de forma crescente:

Algoritmos de Ordenação (Vetor Crescente)							
Algoritmo/Tamanho	1000	5000	10000	15000	20000	25000	50000
Bubble Sort	2.6	14.6	26.4	56.4	97.6	156.4	612
Bubble Sort(melhorado)	0	0	0	0	0	0	0.2
Quick Sort(pivô início)	3.2	24.2	72.6	156	276	475.4	2178.8
Quick Sort(pivô centro)	0	0.6	0	0.2	0	2.4	8.2
Insertion Sort	0	0	0	0	0	0.2	1.8
Shell Sort	0.2	0.6	0	0	0	1.6	3.8
Selection Sort	3	34.2	154.6	351.8	624.4	256.6	964.8
Heap Sort	0	1	1	1	2	5.2	8
Merge Sort	2	20	51	129.6	184	313.8	1085.2



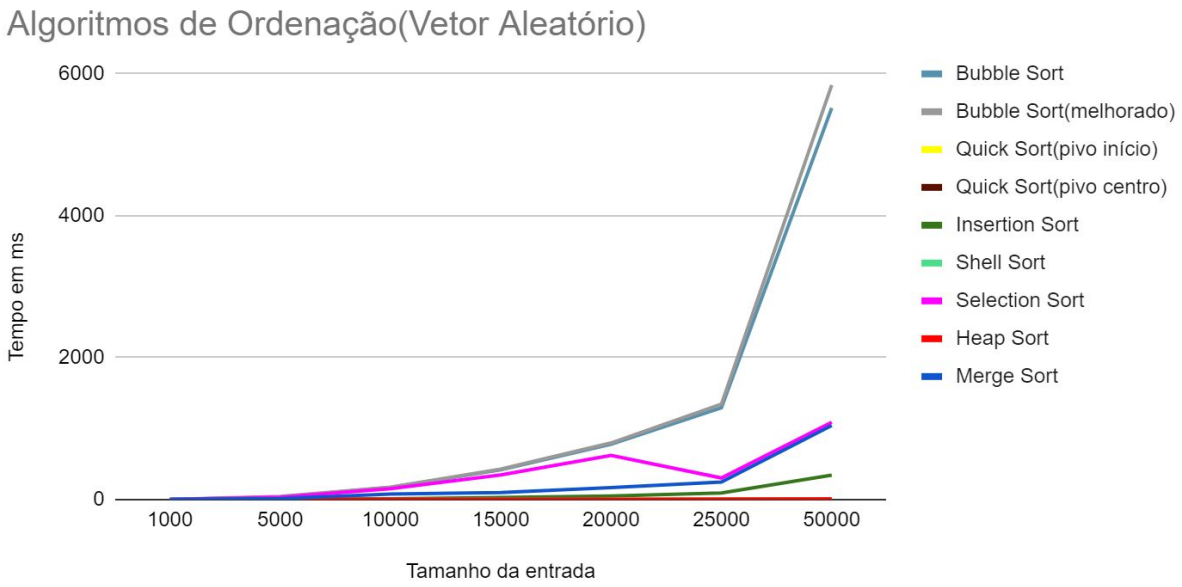
Agora, demonstraremos o tempo de execução de cada algoritmo no vetor com números ordenados de forma decrescente:

Algoritmos de Ordenação (Vetor Decrescente)							
Algoritmo/Tamanho	1000	5000	10000	15000	20000	25000	50000
Bubble Sort	2.2	27.2	102.2	232	412.4	649.2	2706.2
Bubble Sort(melhorado)	3.2	26.8	103.6	234.2	415.4	585.8	2367.4
Quick Sort(pivô início)	1.4	23.4	105.8	260.8	468.6	637.8	2668.8
Quick Sort(pivô centro)	0	0.4	0	0	0.2	1	2.2
Insertion Sort	1.8	7.6	26.8	59.2	106	183.6	703.8
Shell Sort	0.8	0.2	0	0	0	1.8	3.6
Selection Sort	3.2	35.8	155.8	349.8	622.4	520.6	2113.2
Heap Sort	0	0.4	0.6	1	2	3.4	6.8
Merge Sort	2.6	17.6	50.8	92	157.8	282	1098



Por fim, temos o tempo de execução de cada algoritmo no vetor com números aleatórios:

Algoritmos de Ordenação (Vetor Aleatório)							
Algoritmo/Tamanho	1000	5000	10000	15000	20000	25000	50000
Bubble Sort	1.2	36	172.6	418.4	779.2	1294.8	5519.4
Bubble Sort(melhorado)	1.4	43.4	175.4	428.6	798	1346	5840.8
Quick Sort(pivô início)	0	0.4	1	1	2	3.8	7.6
Quick Sort(pivô centro)	0	0.2	1	1	2	3	5.4
Insertion Sort	0.8	3.6	12	28.4	50.6	93.8	344.2
Shell Sort	0.6	0.4	1	2	3	3.8	8.6
Selection Sort	1	33	154.2	348.4	621.6	304	1090.2
Heap Sort	0	0.4	1.2	2	2.2	4	8.4
Merge Sort	1.6	15	76.2	97.8	169.8	247	1044



Em resumo, cada um dos algoritmos apresentados ao longo do trabalho, possui diferentes vantagens e desvantagens, sendo os fatores: tamanho do vetor e ordem dos elementos(crescente, decrescente e aleatória), muito importantes no processo de testar como cada algoritmo se comporta de acordo com entrada submetida a ele. Foi possível notar muitas vezes que vários algoritmos mudaram sua eficiência em relação aos demais, graças a entrada que eles obtiveram.

Durante os testes, para vetores em ordem crescente, destacaram-se(obtiveram os melhores tempos) os algoritmos BubbleSort(melhorado), Insertion Sort, Quick Sort com pivô no centro e Shell Sort.
Para vetores de ordem decrescente destacaram-se os algoritmos Quick Sort com pivô no centro,Shell Sort e Heap Sort.
Por fim, para vetores com elementos aleatórios destacaram-se os algoritmos Quick sort(tanto o com pivô no início, como o com pivô no centro) e Shell sort.
O Bubble Sort e o Bubble Sort(melhorado) apesar de serem os algoritmos mais fáceis de serem implementados(dos algoritmos estudados), não apresentam resultados satisfatórios, tendo em vista seu exacerbado número de comparações. Sua aplicação só pode ser indicada para vetores de poucas entradas, além de não serem muito aplicados em projetos, já que na maioria dos casos acabam sendo ineficientes.
O Quick Sort ao fazer as subdivisões do vetor e fazer as inserções usando o pivô, reduz o tempo de execução, porém, ainda é alto o número de comparações e trocas. Esse algoritmo é uma boa opção quando se deseja execuções em tempos pequenos ao custo de recursos computacionais, embora ele seja bem imprevisível em diversos casos, mesmo com pivô no centro(já que ele pode cair no caso de complexidade quadrática).
O Insertion Sort é útil para ordenar vetores com poucas entradas. Uma das vantagens desse algoritmo é possuir uma quantidade menor de comparações, mas ele possui como desvantagem um elevado número de trocas.
O Shell Sort, em relação aos algoritmos observados, foi o que teve resultados mais satisfatórios com tempos pequenos e resultados estáveis, pode-se dizer que ele foi um dos melhores, senão o melhor algoritmo estudado.
O Selection Sort é útil em casos similares ao do Insertion Sort, porém, ele trabalha melhor do que o Insertion Sort com grandes quantidades. A vantagem do uso desse algoritmo é o seu número de trocas ser inferior ao número de comparações.
O Heap Sort é um algoritmo de ordenação que apresentou resultados satisfatórios, com tempos de execução pequenos, porém, assim como o Quick Sort ele não é estável e são efetuadas muitas comparações.
O Merge Sort é um algoritmo de ordenação bastante estável, porém, seus resultados são medianos. Esse algoritmo é indicado em usos onde a divisão das estruturas em estruturas menores seja o objetivo.
De forma geral, por meio da análise dos resultados e das implementações de cada um deles, foi possível descobrir diversas vantagens e desvantagens de cada algoritmo, além da complexidade e dos tempos de execução, que eram as grandes vertentes desse trabalho.

5. Conclusão

Com base nos resultados obtidos ao longo do trabalho, expandimos nossos conhecimentos com relação aos algoritmos de ordenação, ao mesmo tempo que estudamos as diferentes complexidades dos mesmos por meio da análise assintótica e das experimentações realizadas.

Após analisar tais algoritmos, concluímos o quão importante é o estudo da complexidade de um algoritmo na computação, afinal quanto menor o tempo de execução e menor o gasto de memória, mais eficiente um algoritmo pode-se tornar.

Porém, cabe ao programador pensar na forma como se irá implementar um algoritmo, já que essa escolha pode tanto beneficiar um projeto, como destruí-lo por completo.

Assim, no que tange aos algoritmos de ordenação, o uso de cada um deles dependerá da situação, ou seja, sempre vale a pena realizar um estudo para determinar qual método de ordenação se encaixará melhor, eliminando quaisquer outros que sejam menos eficientes e consequentemente evitando os piores casos de complexidade no projeto.

6. Referências Bibliográficas

COELHO, HERBERT; FÉLIX, NADIA. **Métodos de Ordenação: Selection, Insertion, Bubble, Merge (Sort)**. Disponível em: https://ww2.inf.ufg.br/~hebert/disc/aed1/AED1_04_ordenacao1.pdf. Acesso em: 22 de fev. 2021.

ELER, DANILO MEDEIROS. **Análise de Algoritmos de Ordenação Parte 1**. Disponível em: https://daniloeler.github.io/teaching/PAA2020/files/PAA-aula05-AnaliseAlgoritmosOrdenacao-Parte1.pdf. Acesso em: 22 de fev. 2021.

ELER, DANILO MEDEIROS. **Análise de Algoritmos de Ordenação Parte 2**. Disponível em: https://daniloeler.github.io/teaching/PAA2020/files/PAA-aula05-AnaliseAlgoritmosOrdenacao-Parte2.pdf. Acesso em: 22 de fev. 2021.

ELER, DANILO MEDEIROS. **Análise de Algoritmos de Ordenação Parte 3**. Disponível em: https://daniloeler.github.io/teaching/PAA2020/files/PAA-aula05-AnaliseAlgoritmosOrdenacao-Parte3.pdf. Acesso em: 22 de fev. 2021.

FORTES, REINALDO. **Aula 12: Ordenação: Bubble, Selection e Insertion Sort**. Disponível em:http://www.decom.ufop.br/reinaldo/site_media/uploads/2013-02-bcc202/aula_12_-_bubblesort-selectionsort-insertionsort_(v1).pdf. Acesso em: 22 de fev. 2021.

FORTES, REINALDO. **Aula 13: Ordenação: MergeSort**. Disponível em: http://www.decom.ufop.br/reinaldo/site_media/uploads/2013-02-bcc202/aula_13_-_mergesort_(v2).pdf. Acesso em: 22 de fev. 2021.

PRESTES, EDSON. **Complexidade de Algoritmos**. Disponível em: http://www.inf.ufrgs.br/~prestes/Courses/Complexity/aula12.pdf. Acesso em: 22 de fev. 2021.