Algoritmos de ordenação

Gabriel Campaneli Iamato
N° USP 15452920
E
Renan Banci Catarin
N° USP 14658181

1 Introdução

A ordenação é uma das operações fundamentais da computação, desempenhando um papel em diversas aplicações, como buscas, análise de dados e processamento de informações. Quando estão sendo manipulados pequenos conjuntos de dados, o algoritmo escolhido não possuí tanto impacto, porém, em situações reais, o volume deles é muito grande, e é uma habilidade essencial saber determinar qual método de ordenação é ótimo para determinado caso.

Os algoritmos a serem analisados serão o Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Quick Sort, Heap Sort, Merge Sort, Radix Sort e Contagem dos menores. Todos testados com entradas já ordenadas(crescente), em ordem inversa(decrescente) e entradas aleatórias(feita uma média com 5 delas). Importante mencionar que os testes foram feitos par os valores de 100, 1.000, 10.000 e 100.000 elementos, isso para perceber o quão acentuadas são suas curvas de crescimento.

O programa em questão foi implementado em linguagem C no qual o usuário escolhe o método de ordenação. Para melhores comparações, além de medir o tempo de execução de cada aplicação, foram adicionadas duas variáveis, uma que contabiliza a movimentação de registros e a outra a comparação entre as chaves.

Os gráficos e tabelas dos resultados serão comentados e comparados para definir qual o mais eficiente.

2 Apresentação e expectativas

Essa seção serve para familiarizar e relembrar o leitor de como os algoritmos se comportam, além de fornecer a complexidade esperada, feita por análises teóricas.

2.1 Bubble

Simples e popular, consiste em percorrer o vetor dado repetidas vezes, sempre comparando os adjacentes e os trocando se tiverem em ordem errada. Foi implementado sua versão aprimorada que detecta quando o array está ordenado, evitando operações desnecessárias.

Complexidade estimada: • **Melhor:** $O(n)$; • **Médio:** $O(n^2)$; • **Pior:** $O(n^2)$.

2.2 Selection

A cada iteração busca o elemento de menor valor, o armazenando em uma variável e o troca com o elemento da n-ésima posição, precisando realizar esse procedimento para todas as posições.

Complexidade estimada: • **Melhor:** $O(n^2)$; • **Médio:** $O(n^2)$; • **Pior:** $O(n^2)$.

2.3 Insertion

Baseia-se em considerar o primeiro elemento do vetor ordenado e, iterativamente, ir inserindo os elementos na posição correta, sob consideração de que os n elementos à esquerda já estão ordenados.

Complexidade estimada: • **Melhor:** $O(n)$; • **Médio:** $O(n^2)$; • **Pior:** $O(n^2)$.

2.4 Shell

É uma implementação mais eficiente do Insertion Sort. Divide o array em sub listas (*gaps*) arbitrários, e aplica o Insertion para cada *gap*, até que diminua ao ponto de ser apenas 1 após a última iteração o vetor estará ordenado.

Complexidade estimada: • **Melhor:** $O(nlog^2n)$; • **Médio:** $O(n^{\frac{3}{2}})$; • **Pior:** $O(n^2)$. (depende da sequência de *gaps* escolhidos.)

2.5 Quick

Popular pela sua eficiência de tempo, é construído a partir do paradigma de dividir e conquistar. A divisão é comandada por um elemento pivô, implementado como a mediana de 3. Os registros com menor valor serão alocados a sua esquerda, e os maiores a direta. O processo é repetido para cada metade criada até que sobre apenas um elemento.

Complexidade estimada: • **Melhor:** $O(nlogn)$; • **Médio:** $O(nlogn)$; • **Pior:** $O(n^2)$.

2.6 Heap

Utiliza uma estrutura Heap de grau dois. Ela segue uma relação de ordem chamada de Heap máximo, em que os registros dos nós filhos são menores ou iguais aos de seu pai. As suas folhas devem sempre priorizar menores níveis e estarem a esquerda. Para ordenar em Heap, se troca a raiz com o elemento de última posição do vetor, o tamanho dela é diminuído em um e ela é reorganizada. onde o processo é repetido $n-1$ vezes.

Complexidade estimada: • **Melhor:** $O(nlogn)$; • **Médio:** $O(nlogn)$; • **Pior:** $O(nlogn)$.

2.7 Merge

Outro algoritmo de divisão e conquista. O array a ser ordenado será dividido em subarrays, ocorre até que fiquem com apenas um elemento. Depois compara os elementos de cada subarray e os adiciona um a um em ordem no vetor final. Para mesclar os elementos é necessário um vetor auxiliar.

Complexidade estimada: • **Melhor:** $O(nlogn)$; • **Médio:** $O(nlogn)$; • **Pior:** $O(nlogn)$.

2.8 Contagem dos menores

A ideia básica é comparar um elemento com todos os outros para identificar quantos menores que ele existem. Feito isso é definida a sua posição final. Ela é guardada num array auxiliar no mesmo índice que tal elemento ocupa. Por fim, é criado um terceiro array, os elementos serão comparados com as posições e devidamente inseridos. Uma condição a se atentar é elementos duplicados.

Complexidade estimada: • **Melhor:** $O(n^2)$; • **Médio:** $O(n^2)$; • **Pior:** $O(n^2)$.

2.9 Radix Sort

Ordena os números por seus dígitos, de forma contraintuitiva, começa do menos para o mais significativo. São utilizadas estruturas auxiliares para armazenar os números de acordo com seus dígitos e depois serão juntados. O número de vezes que isso será repetido depende diretamente de quantas casas decimais (d) o maior elemento possui. O Radix é inteiramente baseado na questão da estabilidade da ordenação.

Complexidade estimada: • **Melhor:** $O(n*d)$; • **Médio:** $O(n*d)$; • **Pior:** $O(n*d)$.

3 Sobre tempo de execução

Serão apresentados os gráficos individuais de cada método seguido de uma breve análise sobre. A escolha de fazer dessa maneira é devido ao intuito de querer comparar o impacto do estado inicial do array para cada algoritmo.

Após isso haverá uma comparação visual de todos em um único gráfico.

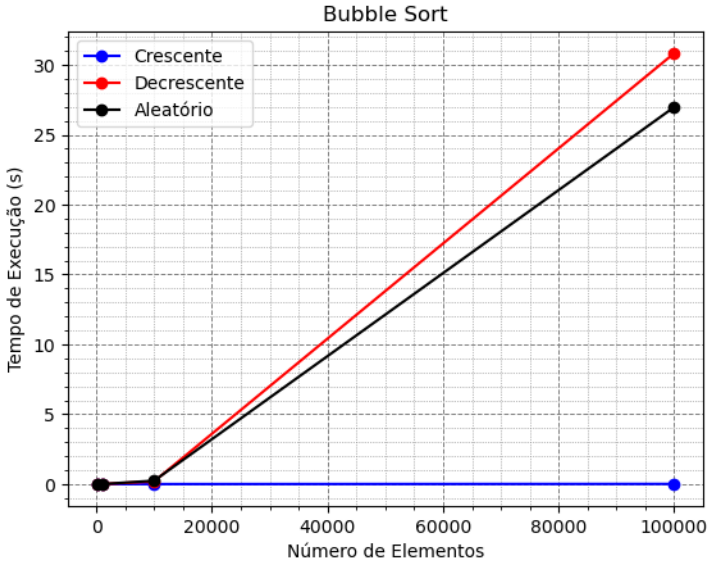

Figura 1: Exibição do gráfico temporal do Bubble Sort.

O gráfico da figura 1 corrobora com o esperado, em casos que o vetor já está ordenado o tempo é praticamente nulo, enquanto uma entrada decrescente leva mais que 30 segundos para terminar o processo. Casos médios se encontram muito mais próximos do pior caso, tornando esse método inviável para ser usado com grandes volumes de dados.

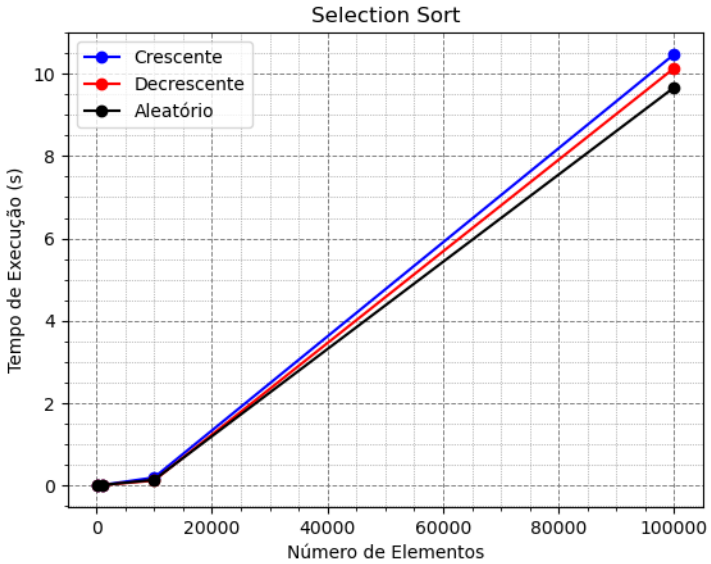

Figura 2: Exibição do gráfico temporal do Selection Sort.

Como esse é um algoritmo simples de se implementar, as comparações com o Bubble Sort são inevitáveis, apesar de os dois possuírem complexidade média de $O(n^2)$, o gráfico da figura 5 revela que o Selection Sort ainda é mais eficiente, possuindo um caso médio de quase 10 segundos, o que ainda é bem ineficiente quando comparamos com os próximos.

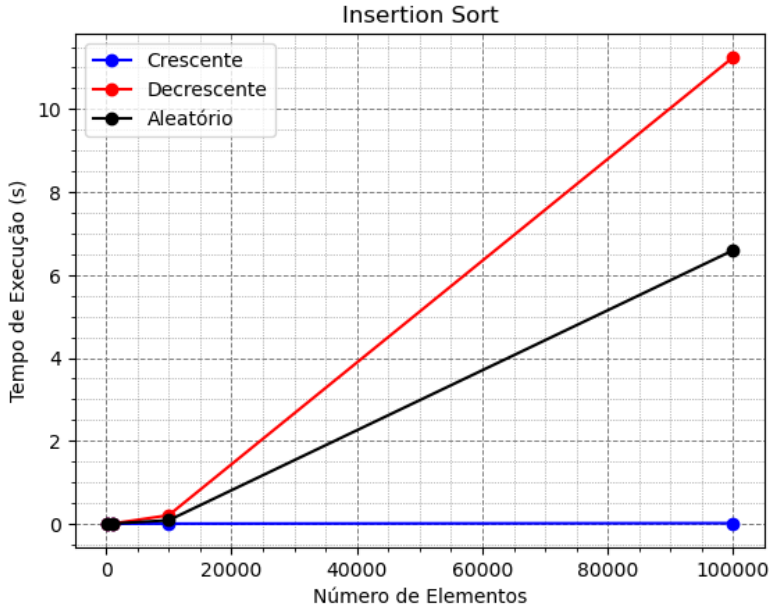

Figura 3: Exibição do gráfico temporal do Insertion Sort.

Outro algoritmo de fácil implementação e pouco eficiente, mas sem dúvida o melhor entre os piores, seus casos médios rodam em quase metade do tempo do Bubble Sort e possuí um ótimo melhor caso, diferente do Selection Sort. Apesar disso, como possuímos outras ferramentas, o Insertion Sort nunca seria recomendado para um caso real.

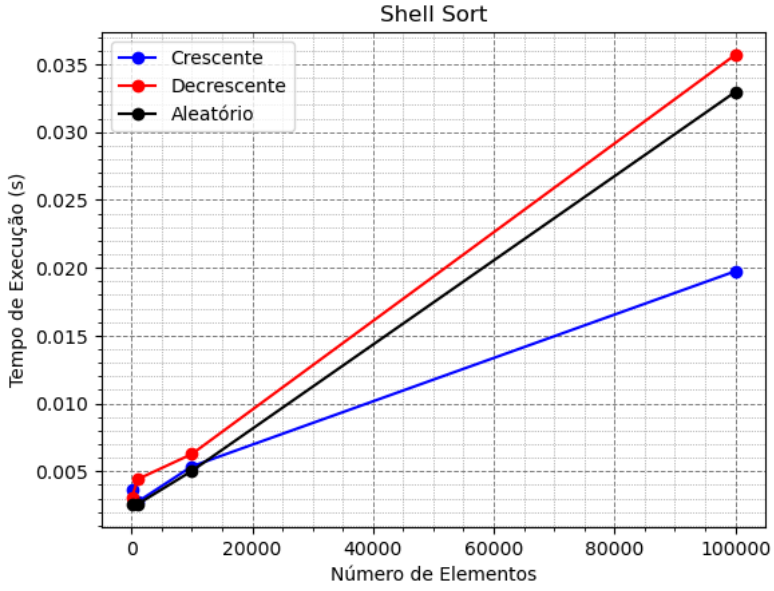

Figura 4: Exibição do gráfico temporal do Shell Sort.

Como prometido na análise teórica, o Shell Sort é uma ótima substituição para o Insertion, apesar de ainda ser perceptível uma demora no processamento dos dados ela nem se compara com os 7 minutos médios. A eficiência é quase 200 vezes maior nesse algoritmo. Claro que dependemos muito de uma sequência de saltos adequadas para seu funcionamento ótimo, mas compensa o tempo de análise para tal.

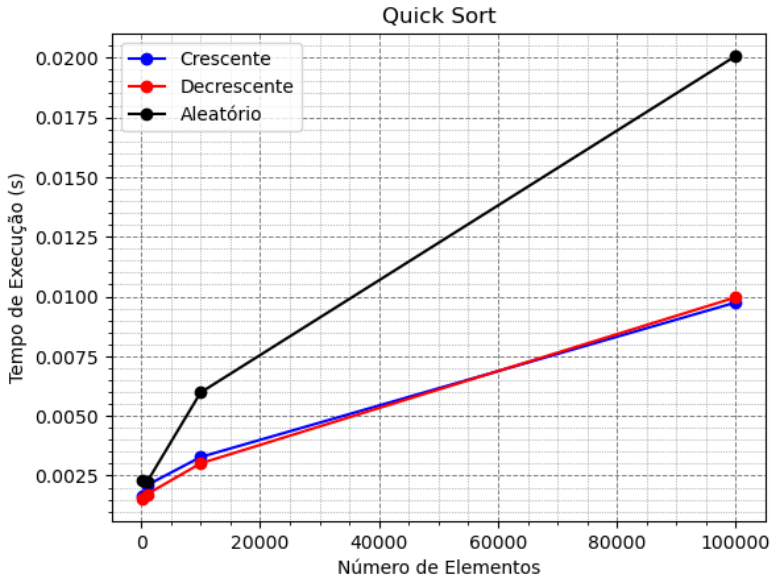

Figura 5: Exibição do gráfico temporal do Quick Sort.

Não à toa é o mais popular, ambas as análises demonstram sua eficiência. Uma coisa importante a se notar é que apesar dos gráficos não possuírem curvatura adequada, o crescimento no Quick Sort é significantemente menor, pois ainda é possível notar que os casos de 10.000 elementos não foram completamente ofuscados pelos de 100.000.

Como o pivô foi decidido com a mediana de 3, isso provocou que as linhas azul e vermelha demarcassem o melhor caso para o algoritmo.

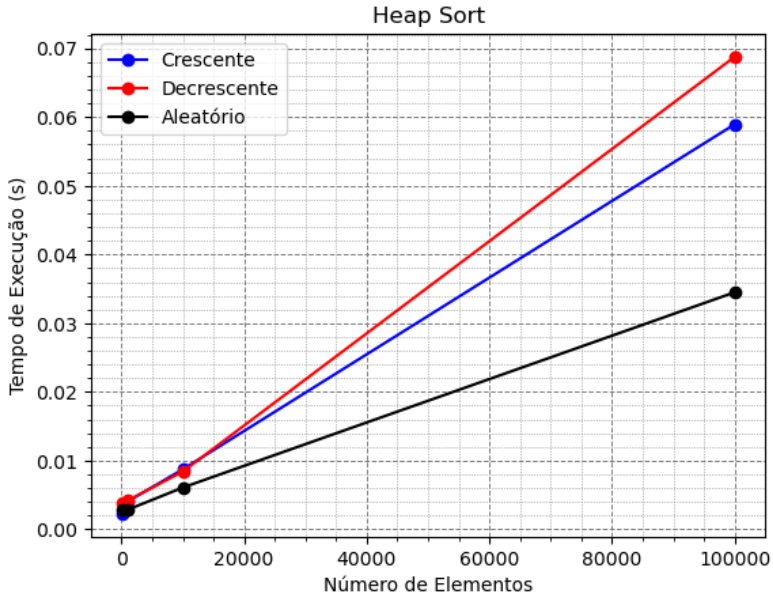

Figura 6: Exibição do gráfico temporal do Heap Sort.

Apesar de não superar o QuickSort, em média o HeapSort se demonstrou um ótimo algoritmo de ordenação. Apesar de ter sido constatado que todos os casos seriam similares, alguns tomaram o dobro do tempo que outros, com o aleatório sendo o melhor resultado com quase 4 centésimos, o que é ótimo. Uma hipótese provável para isso é que rearranjar a heap para o segundo caso é mais rápido.

E sobre os dados um pouco menores, como os de 10.000 elementos, ele até supera o QuickSort em seu caso médio.

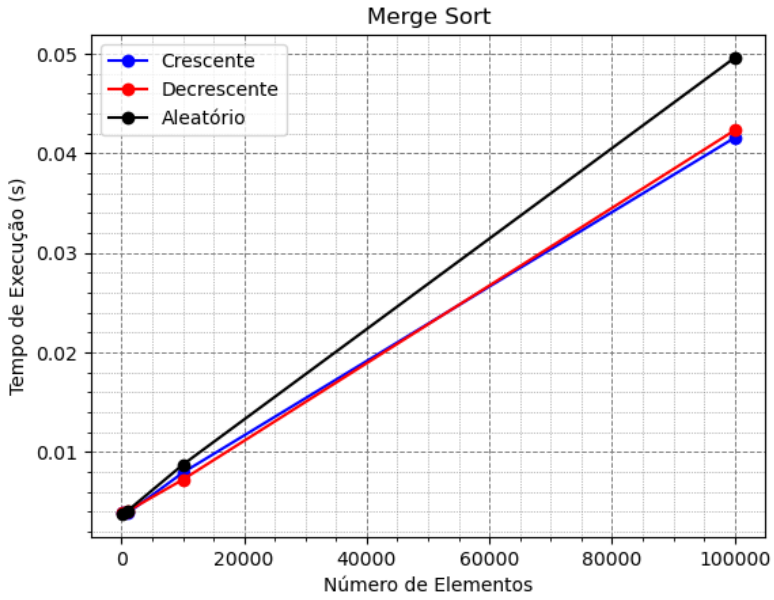

Figura 7: Exibição do gráfico temporal do Merge Sort.

O gráfico a cima revela o comportamento previsto na teoria, pois todos os casos são similares, não se destacando um melhor ou pior caso, apenas o caso médio que dá uma leve deslocada dos outros, mas uma diferença menor que um centésimo de segundo.

Apesar de seu comportamento ser $O(nlogn)$, dados mostram que é mais lento que o Quick e Heap Sort.

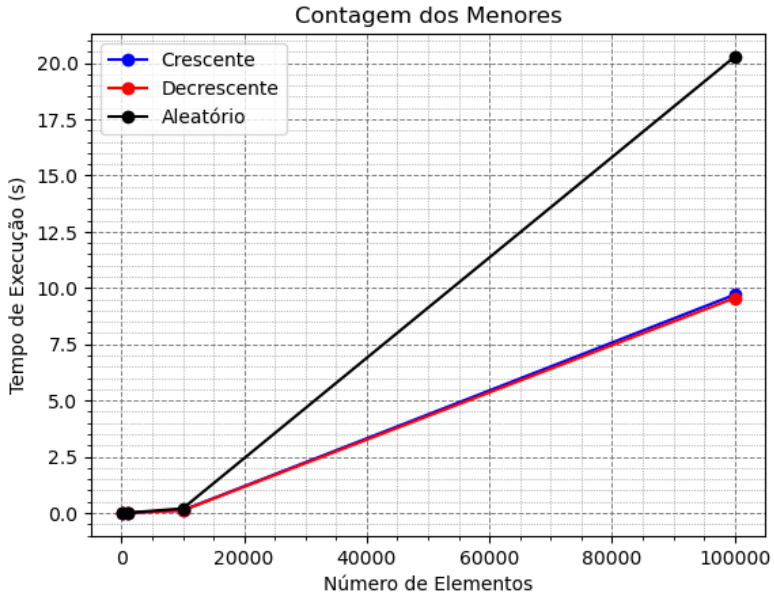

Figura 8: Exibição do gráfico temporal do COntagem de Menores.

Voltando a métodos de complexidade $O(n^2)$, esse, além da baixa eficiência, necessita de memória adicional, com certeza um algoritmo que não é recomendado para nenhuma situação. Sua implementação não é tão simples quanto ao Insertion Sort (o melhor de complexidade exponencial), e nem supera ele em questão de performasse, sendo um caso médio do Insertion 3 vezes mais rápido que o Contagem dos Menores.

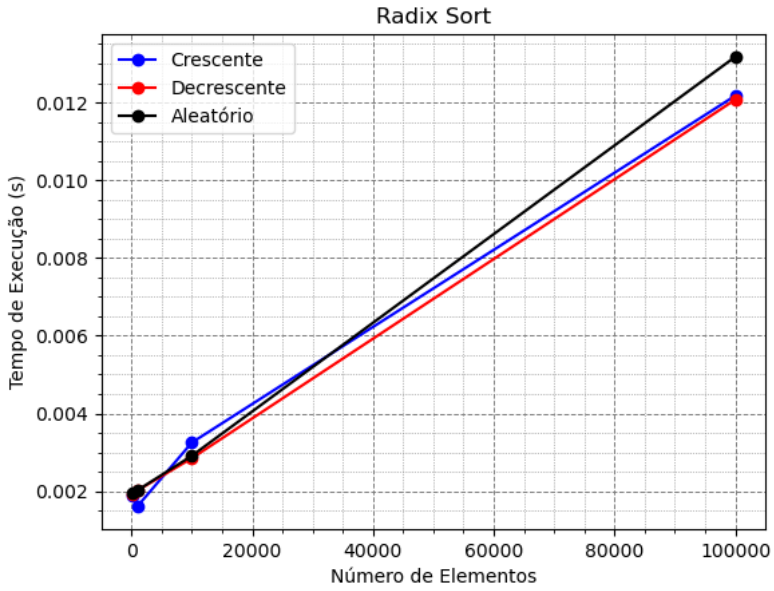

Figura 9: Exibição do gráfico temporal do Radix Sort.

O gráfico 9 mostra como o Radix Sort pode ser eficiente, tanto que nos casos testes realizados superou o Quick Sort em seu caso médio. Porém, é importante lembrar que sua eficiência não depende apenas de n, e, conforme os números manipulados possuam mais casas decimais, mais ineficiente fica para um determinado n.

3.1 Visão geral

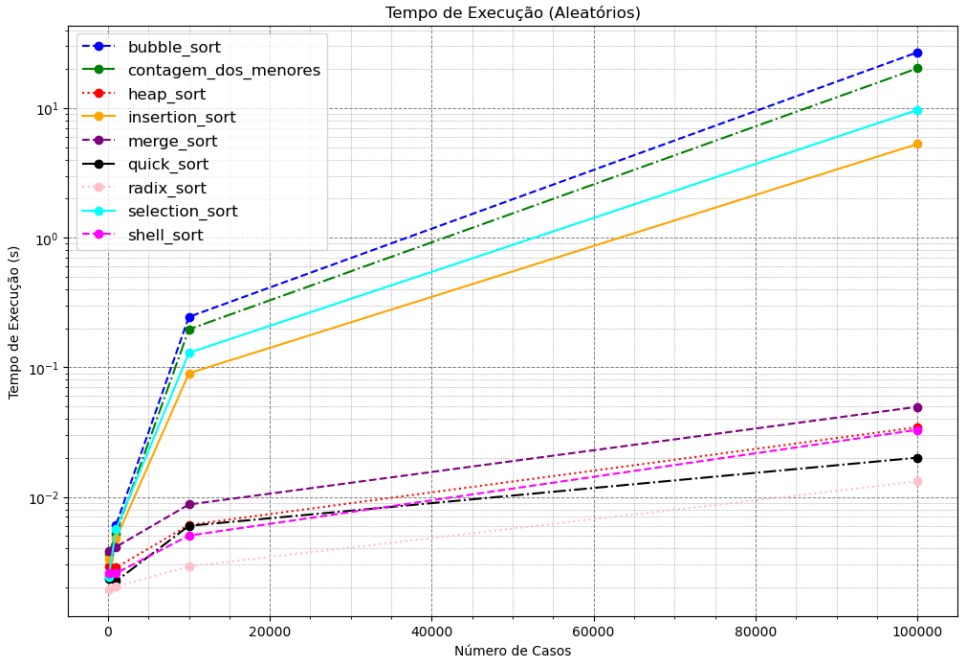

Figura 10: Comparação temporal de todos os algoritmos para casos aleatórios.

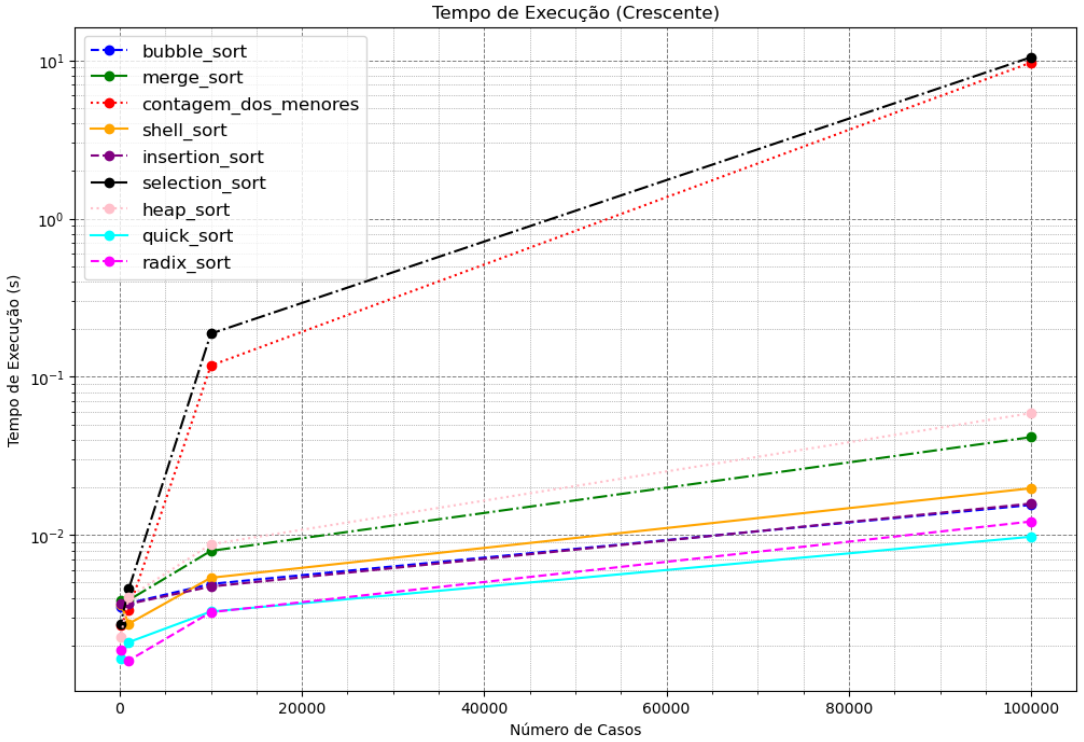

Figura 11: Comparação temporal de todos os algoritmos para casos já ordenados.

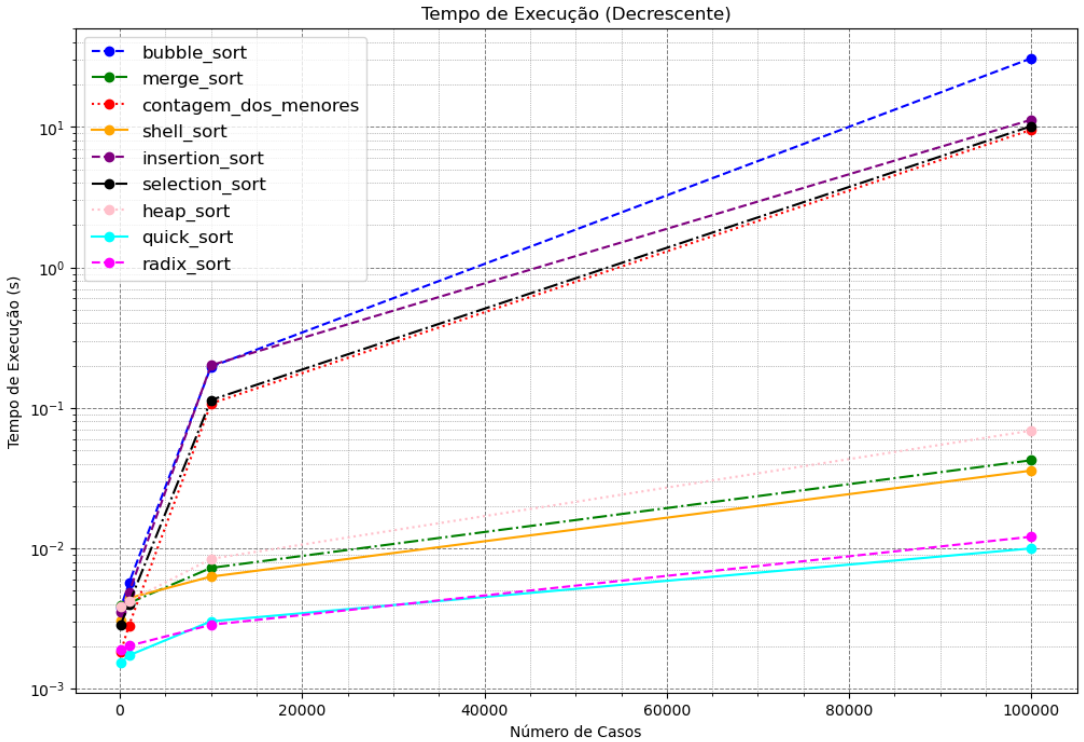

Figura 12: Comparação temporal de todos os algoritmos para casos inversamente ordenados.

4 Sobre Comparações de Chaves

Como já comentado na introdução, foram usadas variáveis para poder extrair mais dados do problema e poder analisá-lo melhor. Para os exibir foram escolhidas tabelas, já que permitem ver repetições melhores que um gráfico e também os valores são discrepantes em muitos casos, o que dificultaria de conseguir a devida compreensão.

Como estão sendo relatados 9 algoritmos diferentes, para uma melhor compreensão serão divididos em duas tabelas, a divisão foi realizada com base na eficiência observada até então, os exponenciais (exceto o ShellSort) em uma e lineares na outra.

4.1 Aleatório

n	Bubble	Selection	Insertion	Contagem
100	4.879	5.049	2.449	4.950
1.000	498.882	500.499	251.293	499.500
10.000	49.992.426	50.004.999	25.033.688	49.995.000
100.000	4.999.906.138	5.000.049.999	2.503.872.312	704.982.704

Tabela 1. Comparação de chaves entre algoritmos lentos

n	Shell	Quick	Heap	Merge	Radix
100	904	825	630	672	99
1.000	15.404	9.579	5.084	9.976	999
10.000	267.687	129.210	42.187	133.616	9.999
100.000	4.346.655	2.093.298	1.624.945	1.668.928	99.999

Tabela 2. Comparação de chaves entre algoritmos rápidos

4.2 Crescente

n	Bubble	Selection	Insertion	Contagem
100	99	5.049	99	4.950
1.000	999	500.499	999	499.500
10.000	9.999	50.004.999	9.999	49.995.000
100.000	99.999	5.000.049.999	99.999	704.982.704

Tabela 3. Comparação de chaves entre algoritmos lentos

n	Shell	Quick	Heap	Merge	Radix
100	763	606	690	672	99
1.000	12.706	9.009	10.208	9.976	999
10.000	182.565	125.439	136.956	133.616	9.999
100.000	2.344.566	1.600.016	1.700.854	1.668.928	99.999

Tabela 4. Comparação de chaves entre algoritmos rápidos

4.3 Decrescente

n	Bubble	Selection	Insertion	Contagem
100	4.950	5.049	5.049	4.950
1.000	499.500	500.499	500.499	499.500
10.000	49.995.000	50.004.999	50.004.999	49.995.000
100.000	4.999.950.000	5.000.049.999	5.000.049.999	704.982.704

Tabela 5. Comparação de chaves entre algoritmos lentos

n	Shell	Quick	Heap	Merge	Radix
100	503	610	566	672	99
1.000	8.006	9.016	8.816	9.976	999
10.000	120.005	125.439	121.696	133.616	9.999
100.000	1.500.006	1.600.030	1.547.434	1.668.928	99.999

Tabela 6. Comparação de chaves entre algoritmos rápidos

4.4 Comentários

Há alguns padrões interessantes aparecendo nas tabelas, um deles é a constancia do Selection, Merge, Contagem dos Menores e Radix que realizam as exatas mesmas quantidades de operações independendo da entrada. Outra coisa interessante a notar é a discrepância entre os melhores e piores casos do Bubble e Insertion (quando o array está ordenado e em ordem inversa, respectivamente). Também é fácil de reparar a correlação entre menor tempo de execução e comparações realizadas.

5 Sobre Movimentação de Registros

5.1 Aleatório

n	Bubble	Selection	Insertion	Contagem
100	2.350	99	1.562	200
1.000	250.294	999	195.688	2.000
10.000	25.023.689	9.999	16.906.351	20.000
100.000	2.503.772.313	99.999	2.309.247.594	200.000

Tabela 7. Movimentação de registros entre algoritmos lentos

n	Shell	Quick	Heap	Merge	Radix
100	904	220	580	672	780
1.000	15.404	2.937	9.079	9.976	10.800
10.000	267.687	37.370	124.210	133.616	144.000
100.000	4.346.655	452.101	1.574.945	1.668.928	1.680.000

Tabela 8. Movimentação de registros entre algoritmos rápidos

5.2 Crescente

n	Bubble	Selection	Insertion	Contagem
100	0	99	99	200
1.000	0	999	999	2.000
10.000	0	9.999	9.999	20.000
100.000	0	99.999	99.999	200.000

Tabela 9. Movimentação de registros entre algoritmos lentos

n	Shell	Quick	Heap	Merge	Radix
100	763	63	640	672	450
1.000	12.706	511	9708	9.976	6.000
10.000	182.565	5.904	131.956	133.616	75.000
100.000	2.344.566	65.535	1.650.854	1.668.928	1.100.000

Tabela 10. Movimentação de registros entre algoritmos rápidos

5.3 Decrescente

n	Bubble	Selection	Insertion	Contagem
100	4.950	99	5.049	200
1.000	499.500	999	500.499	2.000
10.000	49.995.000	9.999	50.004.999	20.000
100.000	4.999.950.000	99.999	5.000.049.999	200.000

Tabela 9. Movimentação de registros entre algoritmos lentos

n	Shell	Quick	Heap	Merge	Radix
100	325	112	640	516	600
1.000	3.606	1.010	9708	8.316	9.000
10.000	1.000.495	10.904	131.956	116.696	120.000
100.000	9.005.906	115.534	1.650.854	1.497.434	1.500.000

Tabela 10. Movimentação de registros entre algoritmos rápidos

5.4 Comentários

Os comentários dessa seção não diferem muito da anterior, os algoritmos apresentam crescimento similar em sua maioria, os constantes permanecem assim (com exceção do Radix que cresce linearmente). A maior inconsistência continua com os mesmos dois métodos e o número de movimentações de registros é proporcional a comparação de chaves vista na seção 4.

6 Conclusão

Com isso, temos uma ampla variedade de dados para tirar conclusões. A princípio, para escolher um desses algoritmos é preciso considerar a questão de melhor eficiência, com isso, a escolha fica entre os 5 métodos que apresentaram menor tempo de execução. Não podemos determinar um como globalmente ótimo, pois as suas características os permitem se destacar em situações específicas. Por exemplo, apesar de o Merge Sort ter obtido o pior tempo dentre os de crescimento linear, ele é o mais recomendado quando queremos manter a ordem de registros com muitas casas decimais. Já no caso de um banco de dados aleatórios que não suporta espaço para alocação auxiliar, o Heap Sort pode ser o mais indicado. Há muitos outros casos específicos, e a análise de melhor opção deve ser concluída baseada nelas.

Apesar dos casos possuírem soluções individuais, é seguro constatar que o Quick Sort é o mais eficiente, isso pode ser constatado pelos gráficos de tempo e com as informações fornecidas na tabela. O Radix supera o Quick em casos aleatórios (que devem receber mais atenção), contudo ele ainda é ocasional, já que sua eficiência não depende apenas da quantidade de elementos no vetor a ser ordenado, como visto em *2.9*.

Por outro lado, dentre os algoritmos quadráticos, os testes indicaram que tanto o Bubble Sort quanto o Contagem dos Menores apresentaram os piores desempenho. Desses de maior complexidade o Insertion Sort superou todos, já realiza menos operações pois não precisa varrer o vetor multiplas vezes e não precisa de memória adicional.

A análise empírica se demonstrou muito importante para ver a verdadeira eficiência de cada algoritmo, isso porque apenas a notação assintótica não é capaz de determinar a maior eficiência de um código frente ao outro.