Universidade Estadual Paulista
“Júlio Mesquita Filho”
Faculdade De Ciências E Tecnologia - Câmpus De Presidente Prudente

Abigail Sayury Nakashima
Miguel De Campos Rodrigues Moret

Relatório do Trabalho Prático I:
Implementação E Análise Experimental De Algoritmos De Ordenação

Presidente Prudente, São Paulo
2024

Introdução

Neste relatório estão os resultados da implementação e análise de nove versões de algoritmos de ordenação, comparando seus tempos de execução com a análise assintótica teórica, conforme os materiais fornecidos em aula e bibliografias recomendadas.

A análise utilizou 11 vetores de tamanhos entre 5 a 100.000 elementos (5, 10, 20, 50, 100, 1000, 10.000, 20.000, 50.000 e 100.000), abrangendo os três tipos de entrada pedidos: gerados em ordem natural (crescente), ordem inversa (decrescente) e ordem aleatória, totalizando 33 vetores distintos. Cada vetor foi executado 100 vezes em cada uma das implementações dos algoritmo de ordenação: BubbleSort (versão original e otimizada), QuickSort (com pivô no início e no meio do vetor), Insertion Sort, Shell Sort, Selection Sort, Heap Sort e Merge Sort.

As execuções permitiram analisar a eficiência e o comportamento com diversos tipos de cenários de entrada, observando e comparando o desempenho real e a complexidade teórica estudada.

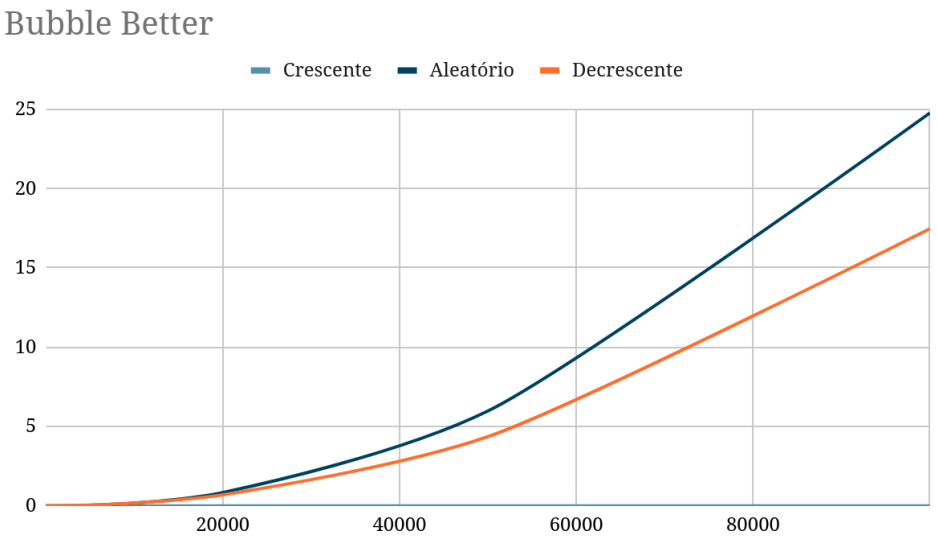
Análise de Algoritmos de Ordenação

Esta análise tem foco principal no tempo de execução total de ordenação dos vetores nos três casos casos de entrada: crescente, decrescente e aleatório. O equipamento utilizado foi um computador pessoal com o processador Intel i3 Gen 11th, 12GB de RAM, usando um sistema operacional Linux e com a linguagem de programação C.

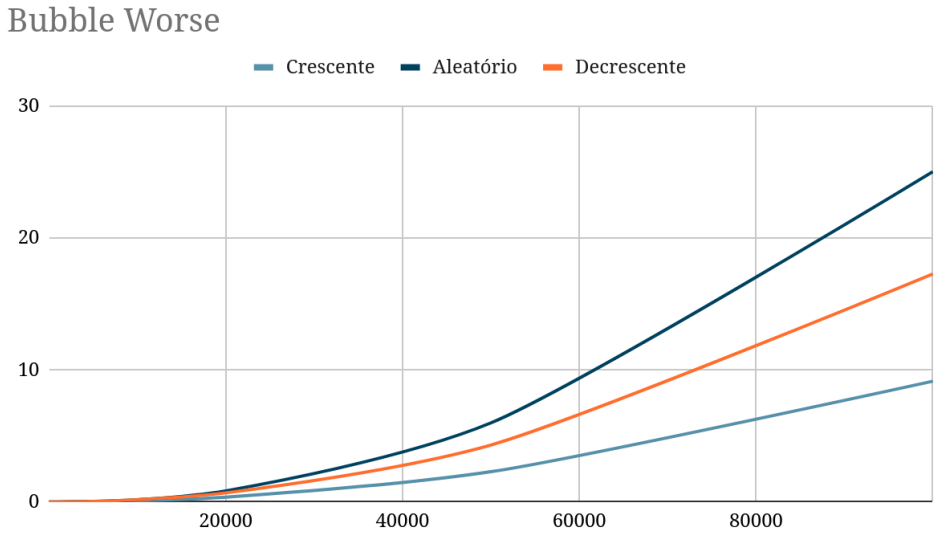
Bubble Sort

O Bubble Sort é um algoritmo de ordenação que se assemelha a bolhas, em que você pega um elemento e o move ao comparar com o próximo elemento, percorrendo todo o vetor, trocando elementos adjacentes que estiverem fora de ordem, até que ela esteja ordenada. Existe o Bubble Sort Original (Worse) e o Melhorado (Better), em que a diferença entre os dois, é que o melhorado possui uma flag que indica se está ordenado ou não, se não houve trocas ou se houve.

A complexidade teórica do Bubble Sort Melhorado é de $O(n^2)$, isso porque, embora o melhor caso (crescente) possua complexidade O(n), no pior caso (e na maioria das ordenações) a complexidade é $O(n^2)$. Isso pode ser observado pelo gráfico abaixo obtido através das experimentações, note que o melhor caso não aparece no gráfico, isso acontece porque a complexidade do melhor caso é muito pequena comparado com a complexidade do pior e médio caso que segue a complexidade teórica estudada.



Da mesma forma, percebe-se que a complexidade teórica do Bubble Sort Original é de $O(n^2)$ assim como a complexidade observada pelas experimentações, isto porque, tanto para o melhor e pior caso, o código do Bubble Sort percorre o vetor n^2 vezes. A diferença de tempo é perceptível, observando que a curvatura do gráfico obtido segue a função n^2.



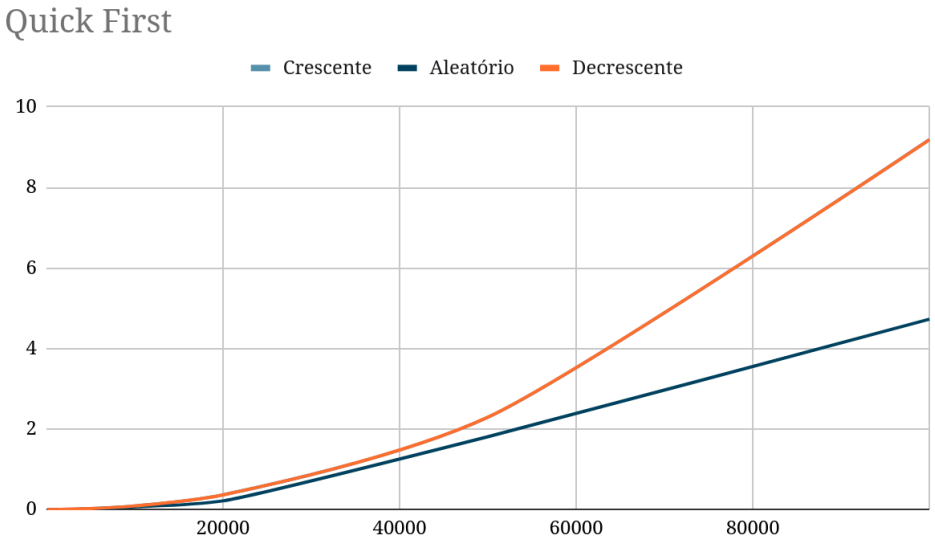
Quick Sort

O Quick Sort é um algoritmo de ordenação que utiliza a técnica de divisão e conquista, utilizando um elemento do vetor como pivô para dividir o vetor, usando-o como base para ordenar o vetor, de forma que todos os elementos menores que o pivô fiquem à sua esquerda, e todos os elementos maiores fiquem à sua direita. Após essa etapa de partição, o algoritmo recursivamente aplica o mesmo processo às sublistas à esquerda e à direita de um novo pivô.

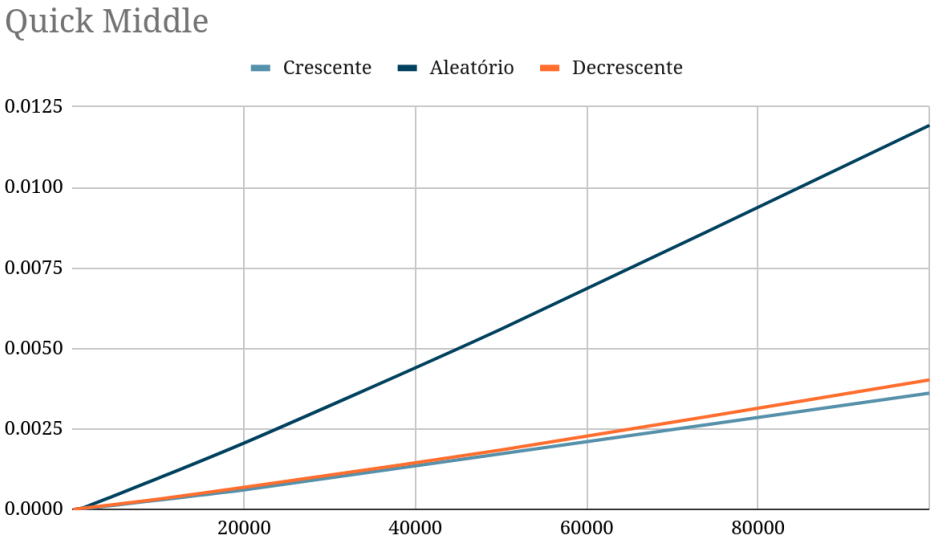
Existem vários métodos para determinar o pivô sendo esta a parte mais importante para determinar a eficiência do Quick Sort, mas tratamos apenas de dois exemplos, que foram o pivô sendo o elemento inicial do vetor e o pivô como o elemento ao meio do vetor.

Para quando o primeiro elemento é tratado como pivô, a complexidade teórica do Quick Sort é avaliada como $O(n^2)$, tanto para o melhor quanto para o pior caso. Observando o gráfico e os tempos obtidos, é possível reconhecer que o melhor caso do Quick Sort com pivô inicial é o aleatório, e apesar de não bem

visível no gráfico, ao analisarmos os tempos obtidos, verificamos que o tempo para decrescente e crescente são muito similares, mas o caso crescente apresenta um tempo “superior”.



E para o caso de o elemento ao meio ser o pivô, a complexidade teórica é de O(n.log(n)), assim também observado no gráfico abaixo.

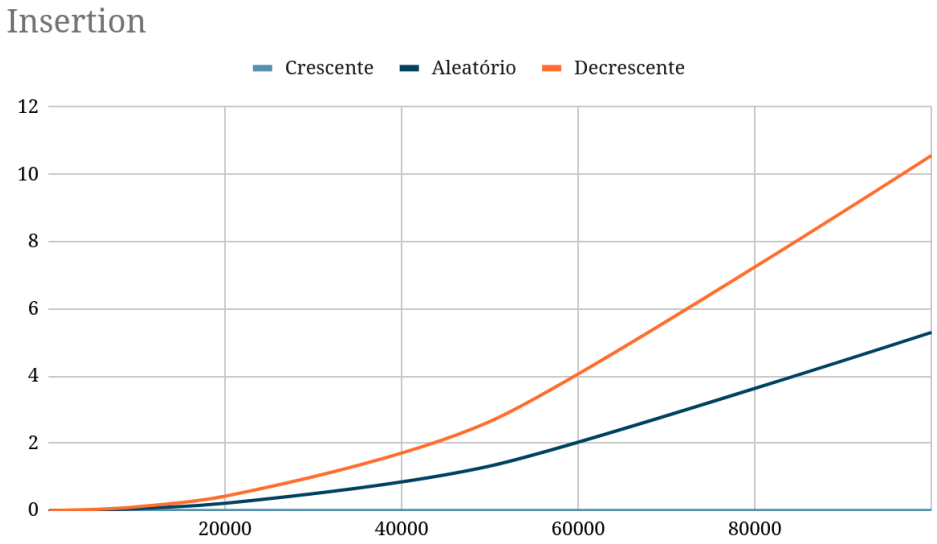


Insertion Sort

O Insertion Sort funciona percorrendo o vetor da esquerda para a direita e, a cada elemento, o insere na posição correta em relação aos elementos anteriores. Para isso, o algoritmo compara o elemento atual com os elementos à sua esquerda

e os desloca até encontrar a posição adequada. Esse processo é repetido para todos os elementos, resultando em uma lista ordenada.

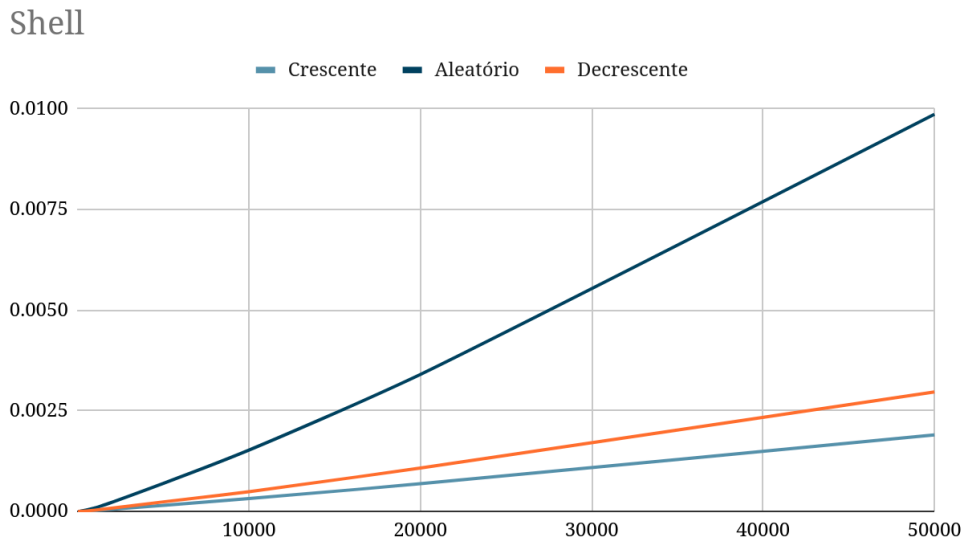
A complexidade teórica do Insertion Sort é de $O(n^2)$, sendo seu melhor caso $O(n)$ para caso esteja ordenado, observando o gráfico da implementação é perceptível, assim como no Bubble Sort, que seu melhor caso, o crescente/ordenado, é muito pequeno comparado ao caso aleatório e decrescente, sendo o decrescente seu pior caso.



Shell Sort

Shell Sort é uma melhoria do algoritmo de ordenação por inserção, ele melhora o algoritmo ao quebrar o vetor em sub-vetores, e esses sub-vetores são ordenados por inserção. A escolha de como os sub-vetores são escolhidos é o principal do Shell Sort, que também determina a sua complexidade, tanto que é observado a dificuldade em um consenso pelas diferentes maneiras de se definir o “gap” escolhido.

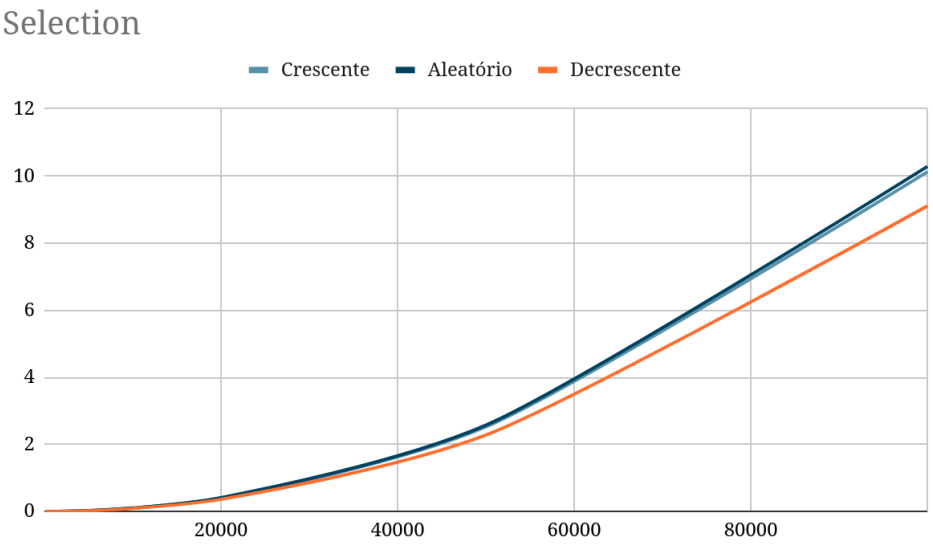
A complexidade teórica do Shell Sort, neste caso, é $O(n.n^{1/2})$, observando o gráfico não se percebe a presença da curvatura fortemente, mas em análise aproximada de um gráfico semelhante, é possível dizer que a quantidade de entrada não foi suficiente para observar o aumento do declive.



Selection Sort

O Selection Sort é um algoritmo de ordenação simples, onde o vetor é ordenado selecionando, a cada passo, o menor elemento do subconjunto não ordenado e trocando-o com o primeiro elemento desse subconjunto. Esse processo se repete até que todo o vetor esteja ordenado. Diferente do Bubble Sort, o Selection Sort minimiza o número de trocas ao realizar, no máximo, n-1 trocas ao longo de toda a ordenação, porém ainda precisa percorrer o vetor repetidamente para encontrar o menor elemento em cada iteração.

A complexidade teórica do Selection Sort é de $O(n^2)$, pois, independentemente do caso (melhor, médio ou pior), ele percorre o vetor várias vezes para encontrar o menor valor em cada passo, o que resulta em um número quadrático de comparações. Isso pode ser visualizado nos gráficos de desempenho, onde a complexidade $O(n^2)$ é predominante e a diferença entre os casos é pouco significativa devido ao comportamento consistente do algoritmo.

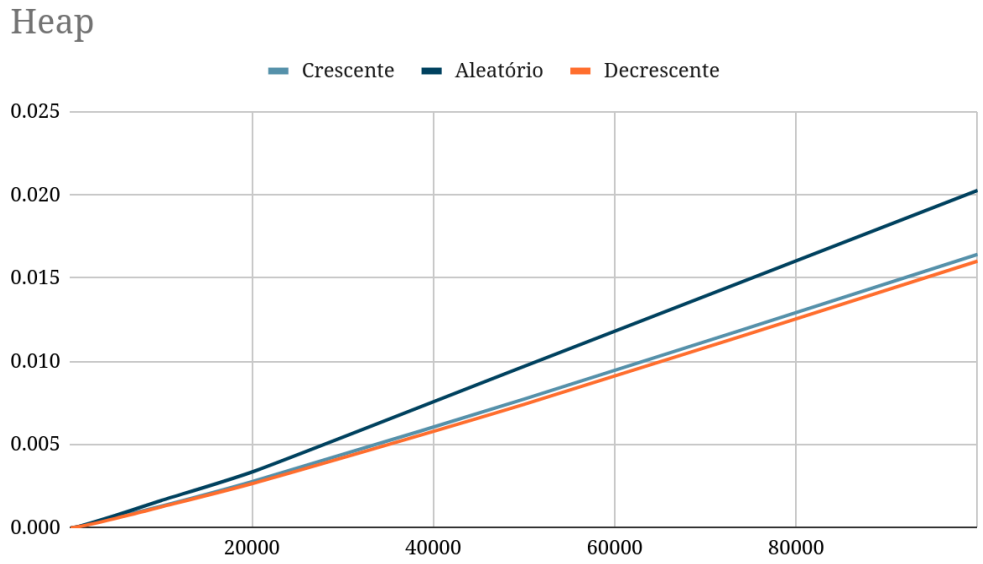


Heap Sort

O Heap Sort é um algoritmo de ordenação que utiliza a estrutura de dados chamada heap, que permite acessar rapidamente o maior (ou menor) elemento.

O algoritmo constroi inicialmente um heap máximo (ou mínimo) a partir do vetor, o que organiza o maior elemento na raiz da estrutura. Em seguida, o Heap Sort remove o maior elemento (no caso do heap máximo) e o coloca na posição final do vetor. Esse processo é repetido, reconstruindo o heap a cada remoção até que todos os elementos estejam ordenados.

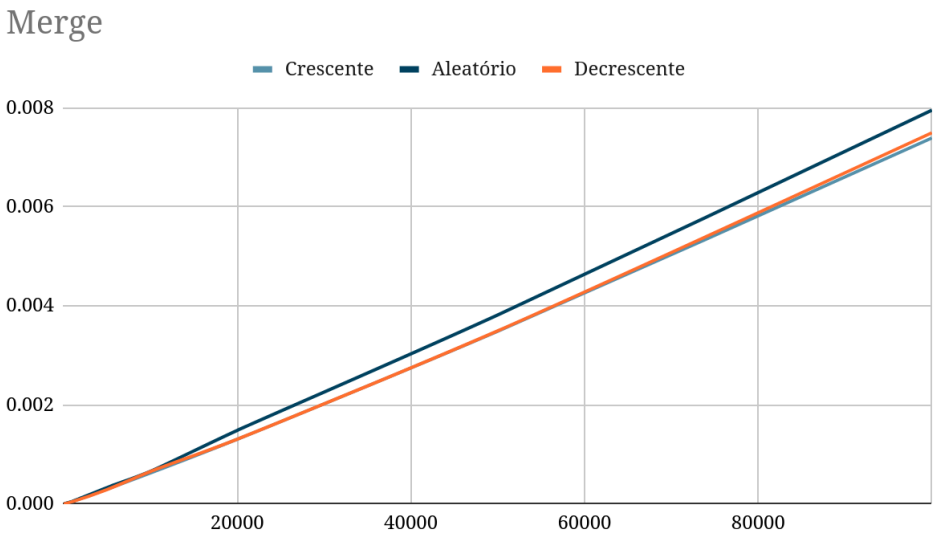
A complexidade teórica do Heap Sort é de O(n.log(n)) tanto para o pior, quanto o melhor, e o médio, pois a operação de reconstruir o heap após cada remoção leva O(log(n)) e é repetida n vezes.



Merge Sort

O Merge Sort é um algoritmo de ordenação baseado na técnica de divisão e conquista. Ele funciona dividindo o vetor em duas metades, recursivamente ordenando cada metade e, por fim, combinando (ou "mesclando") essas metades ordenadas em um único vetor ordenado. O processo de mesclagem é feito comparando os elementos das duas metades e organizando-os de forma ordenada à medida que são combinados.

A complexidade teórica do Merge Sort é de O(n.log(n)) no pior, médio e melhor caso. Mesmo no melhor caso, onde o vetor já está parcialmente ordenado, o algoritmo ainda precisa fazer a mesclagem completa, garantindo uma complexidade de O(n.log(n)). Isto também pode ser observado no gráfico abaixo gerado pela implementação do Merge Sort.



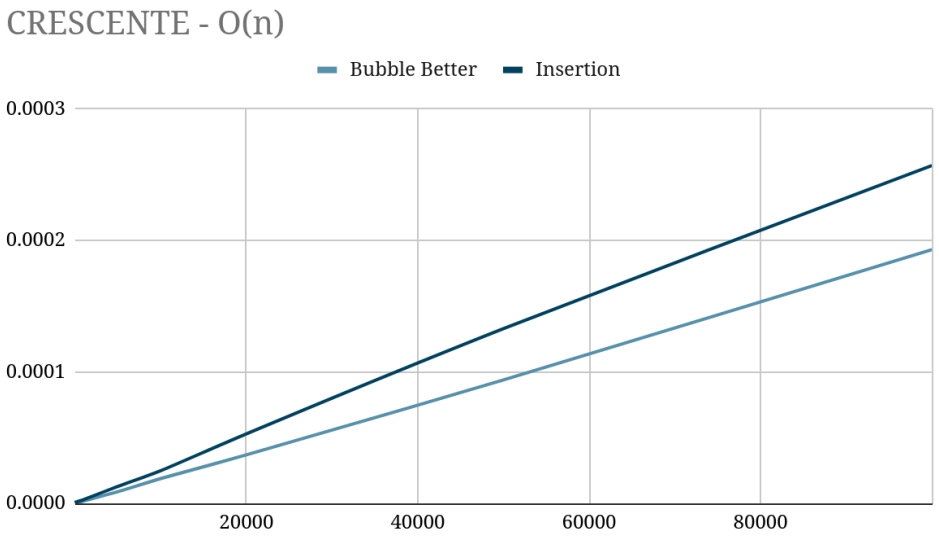
Comportamentos Gerais

Durante a análise dos algoritmos é possível perceber que alguns deles possuem comportamentos semelhantes quanto a sua complexidade.

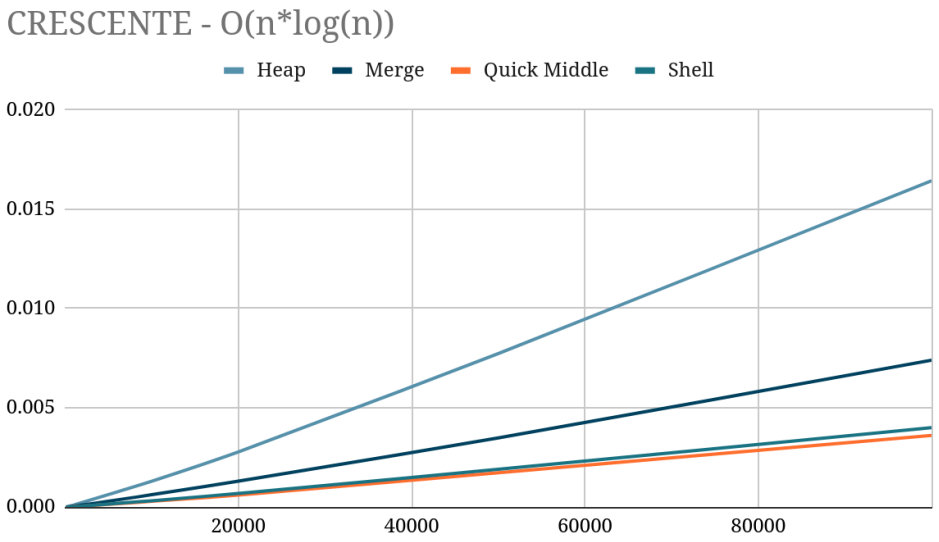
O gráfico abaixo mostra o comportamento dos algoritmos aplicados a arrays ordenados em ordem crescente (melhor caso). É possível observar que algoritmos como o Quick Sort (com pivo no elemento inicial), o Bubble Sort (sem melhoria) e o Selection Sort apresentam tempos de execução muito superiores aos dos outros algoritmos. Isso se deve ao seu comportamento $O(n^2)$, que resulta em um grande número de comparações e trocas, mesmo no melhor caso.

Separando o gráfico anterior com base nos comportamentos dos algoritmos, temos:

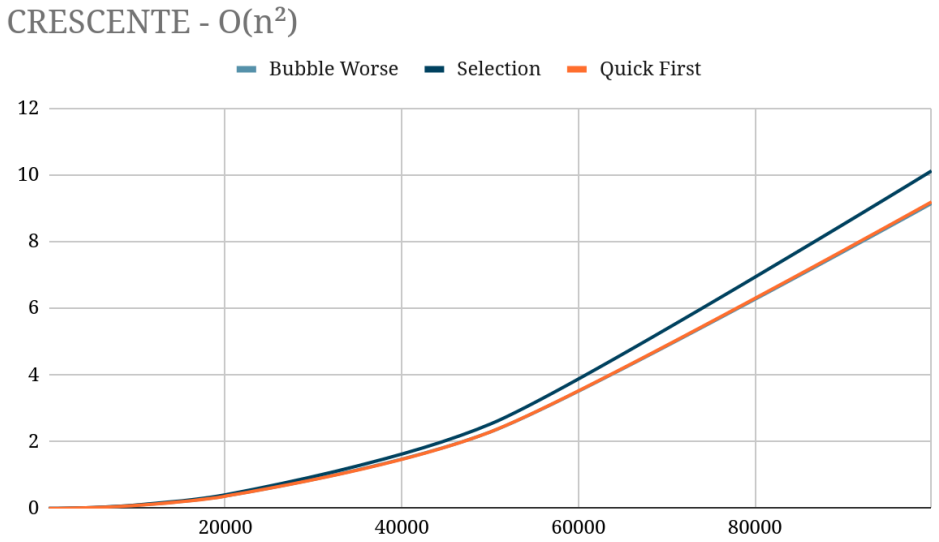
Algoritmos com verificação de ordenação - tanto o Bubble Sort melhorado e o Insertion Sort verificam se o array está ordenado no início de suas implementações. Isso faz com que, quando o array já está ordenado, eles tenham complexidade de O(n).



Algoritmos Recursivos - o Heap Sort, Merge Sort, Shell Sort e Quick Sort (com pivô no elemento central) são algoritmos recursivos que dividem o array de maneira quase central em suas implementações. Isso faz com que, quando o array está ordenado, eles apresentam uma complexidade de O(n.log(n)).

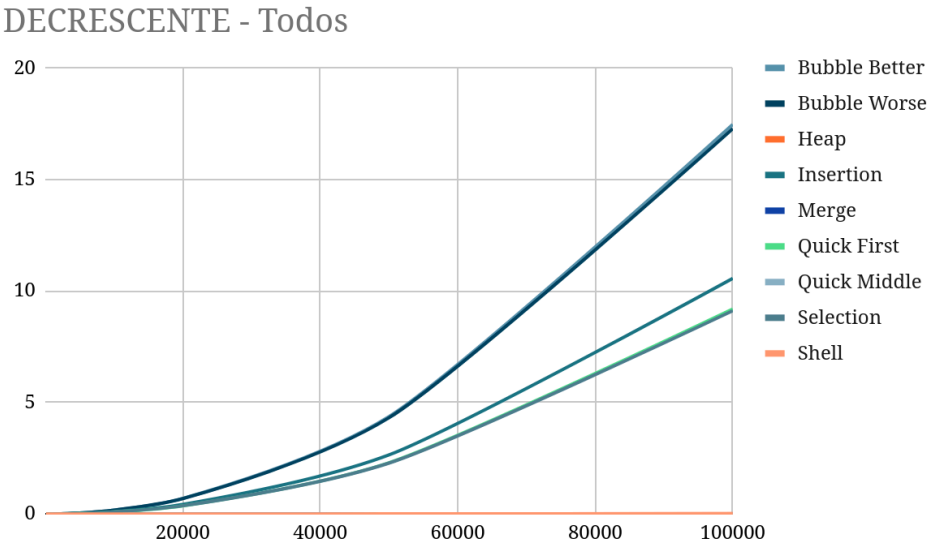


Algoritmos de Loops Duplos - o Selection Sort, Quick Sort (com pivô no elemento inicial), Bubble Sort (sem melhoria) são algoritmos que possuem loops duplos ou múltiplos para comparar elementos e realizar trocas. Isso faz com que, quando o array está ordenado, eles apresentam uma complexidade de $O(n^2)$.



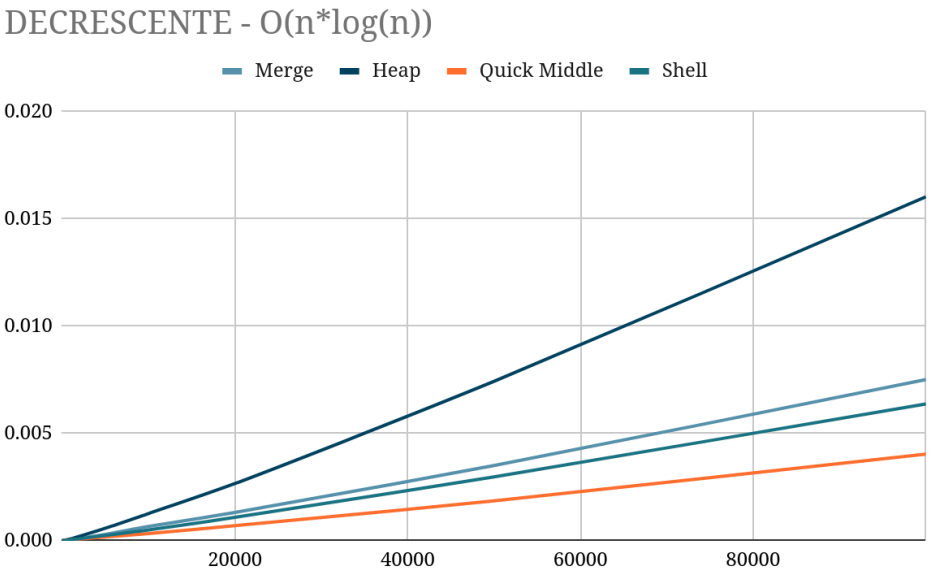
O gráfico abaixo mostra o comportamento dos algoritmos aplicados a arrays ordenados em ordem decrescente (pior caso). É possível observar que alguns algoritmos como o Quick Sort (com pivo sendo o elemento inicial), os Bubble Sort (com e sem melhoria), o Insertion Sort e o Selection Sort apresentam tempos de execução muito superiores aos dos outros algoritmos, isso ocorre devido ao seu

comportamento O(n²), que resulta em um grande número de comparações e trocas no pior caso.

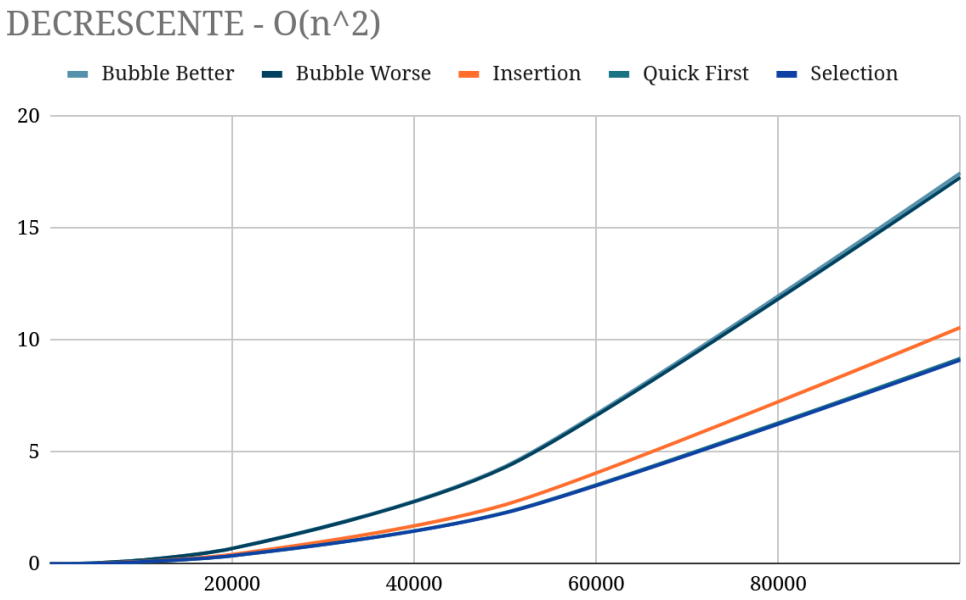


Separando o gráfico anterior com base nos comportamentos dos algoritmos, temos:

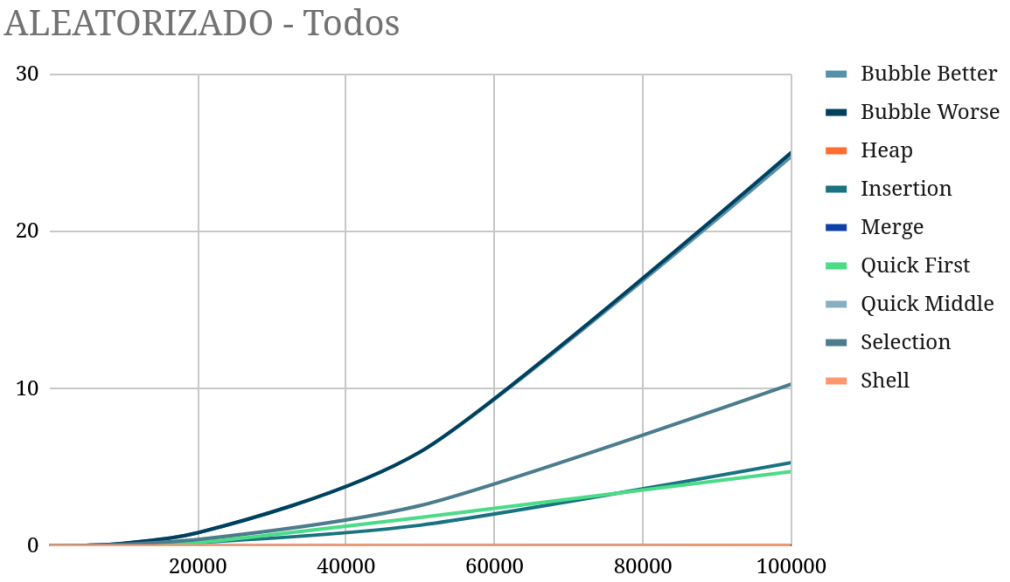
Algoritmos Recursivos - o Heap Sort, Merge Sort, Shell Sort e Quick Sort (com pivô no elemento central) são algoritmos recursivos que dividem o array de maneira quase central em suas implementações. Isso faz com que, quando o array está desordenado, eles apresentam uma complexidade de O(n.log(n)).



Algoritmos de Loops Duplos - o Selection Sort, Quick Sort (com pivô no elemento inicial), Bubble Sort (sem melhoria) são algoritmos que possuem loops duplos ou múltiplos para comparar elementos e realizar trocas - Isso faz com que, quando o array está desordenado, eles tenham complexidade de $O(n^2)$.

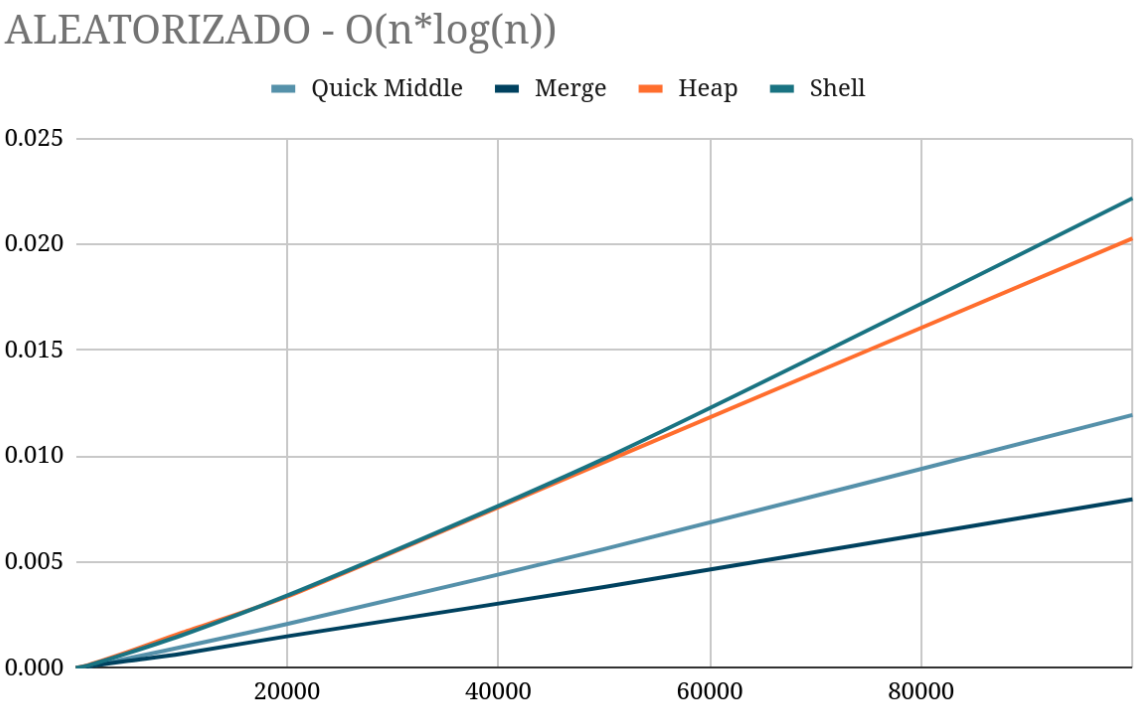


O gráfico abaixo mostra o comportamento dos algoritmos aplicados a arrays ordenados aleatoriamente. É possível observar que algoritmos como o Quick Sort (com pivo sendo o elemento inicial), os Bubble Sorts (com e sem melhoria), Insertion Sort e o Selection Sort apresentam tempos de execução muito superiores aos dos outros algoritmos, isso ocorre devido ao seu comportamento $O(n^2)$, que resulta em um grande número de comparações e trocas.

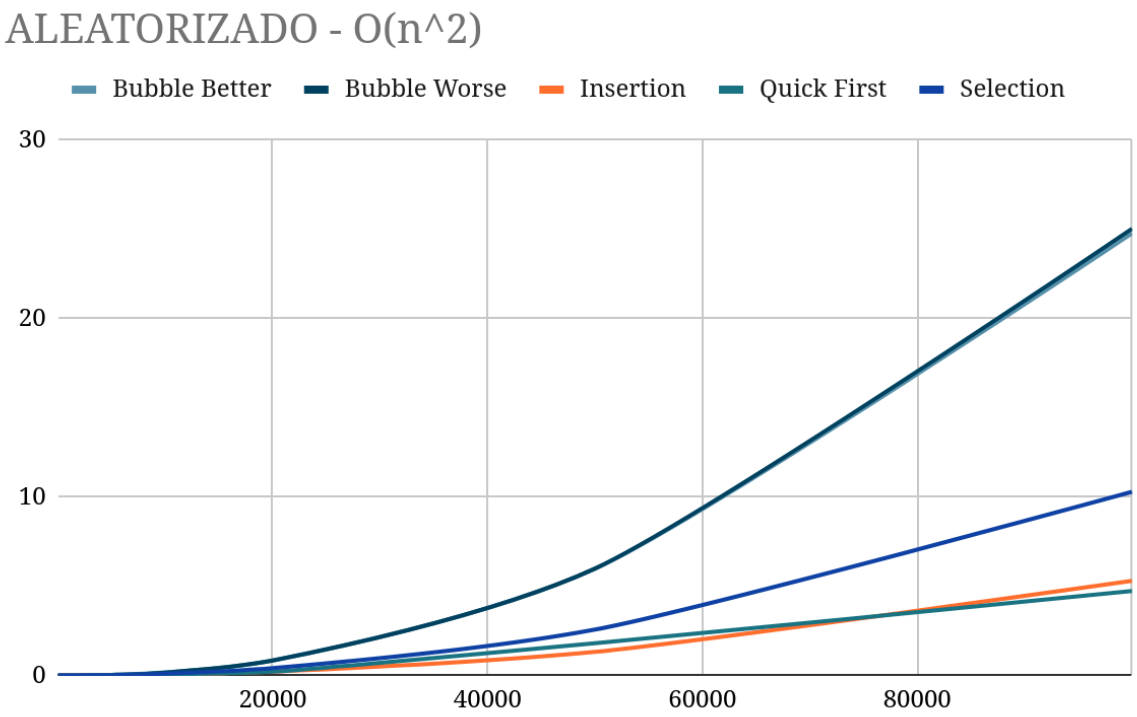


Separando o gráfico anterior baseando-se nos comportamentos dos algoritmos, temos:

Algoritmos Recursivos - o Heap Sort, Merge Sort, Shell Sort e Quick Sort (com pivô no elemento central) são algoritmos recursivos que dividem o array de maneira quase central em suas implementações. Isso faz com que, quando o array está desordenado, eles apresentam uma complexidade de O(n.log(n)).



Algoritmos de Loops Duplos - o Selection Sort, Quick Sort (com pivô no elemento inicial), Bubble Sort (sem melhoria) são algoritmos que possuem loops duplos ou múltiplos para comparar elementos e realizar trocas. Isso faz com que, quando o array está ordenado, eles apresentam uma complexidade de $O(n^2)$.



Tabelas de tempos

Método de coleta dos dados apresentados: cada algoritmo foi testado 100 vezes no mesmo array, e a média desses tempos foi calculada para obter o resultado final.

Note que alguns valores estão 'incorretos' (diminuem em comparação com valores anteriores) em casos com poucos elementos. Isso ocorre porque a ferramenta de medição não foi capaz de registrar tempos extremamente pequenos.

Tabela de tempos com os elementos ordenados em ordem crescente

elementos	Bubble Sort (Original)	Bubble Sort (Melhorado)	Heap Sort	Insertion Sort	Merge Sort	Quick Sort (pivô inicial)	Quick Sort (pivô central)	Selection Sort	Shell Sort
5	**0**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**
10	**0.000001**	**0.000001**	**0.000002**	**0.000001**	**0.000002**	**0.000001**	**0.000001**	**0.000001**	**0.000001**
20	**0.000001**	**0.000001**	**0.000002**	**0.000001**	**0.000002**	**0.000002**	**0.000001**	**0.000001**	**0.000001**
50	**0**	**0.000004**	**0.000004**	**0**	**0.000003**	**0.000006**	**0.000002**	**0.000004**	**0.000001**
100	**0**	**0.000012**	**0.000007**	**0.000001**	**0.000004**	**0.000012**	**0.000002**	**0.000011**	**0.000002**
1000	**0.000002**	**0.000977**	**0.0001**	**0.000003**	**0.000049**	**0.000946**	**0.000021**	**0.001044**	**0.000022**
5000	**0.000009**	**0.023645**	**0.000635**	**0.000013**	**0.000298**	**0.023375**	**0.000138**	**0.025766**	**0.000154**
10000	**0.000019**	**0.093367**	**0.001304**	**0.000025**	**0.000624**	**0.091258**	**0.000294**	**0.103799**	**0.000321**
20000	**0.000037**	**0.36842**	**0.002777**	**0.000053**	**0.001306**	**0.3662**	**0.000608**	**0.405534**	**0.00069**
50000	**0.000094**	**2.289087**	**0.007733**	**0.000133**	**0.003488**	**2.292298**	**0.001734**	**2.526278**	**0.001902**
100000	**0.000193**	**9.149791**	**0.01643**	**0.000257**	**0.007394**	**9.193688**	**0.003612**	**10.12379**	**0.003998**

Tabela de tempos com os elementos ordenados em ordem decrescente

elementos	Bubble Sort (Original)	Bubble Sort (Melhorado)	Heap Sort	Insertion Sort	Merge Sort	Quick Sort (pivô inicial)	Quick Sort (pivô central)	Selection Sort	Shell Sort
5	**0.000001**	**0.000001**	**0.000001**	**0**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**
10	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**
20	**0.000002**	**0.000002**	**0.000002**	**0.000001**	**0.000002**	**0.000002**	**0.000001**	**0.000001**	**0.000001**
50	**0.000009**	**0.000009**	**0.000003**	**0.000004**	**0.000002**	**0.000004**	**0.000002**	**0.000004**	**0.000001**
100	**0.00003**	**0.000029**	**0.000006**	**0.000014**	**0.000004**	**0.000012**	**0.000002**	**0.00001**	**0.000002**
1000	**0.002077**	**0.001933**	**0.000083**	**0.001084**	**0.000047**	**0.000983**	**0.000034**	**0.000999**	**0.000035**
5000	**0.04435**	**0.044249**	**0.000577**	**0.02656**	**0.000291**	**0.022916**	**0.000163**	**0.023085**	**0.000241**
10000	**0.172222**	**0.171936**	**0.001262**	**0.106074**	**0.000655**	**0.091356**	**0.000325**	**0.091574**	**0.000494**
20000	**0.694487**	**0.688647**	**0.002659**	**0.423916**	**0.001308**	**0.364719**	**0.000692**	**0.366185**	**0.001081**
50000	**4.362226**	**4.311733**	**0.00742**	**2.645754**	**0.003499**	**2.295404**	**0.001851**	**2.280823**	**0.002971**
100000	**17.466386**	**17.274501**	**0.016028**	**10.564141**	**0.007501**	**9.187982**	**0.004025**	**9.11073**	**0.006366**

Tabela de tempos com os elementos ordenados aleatoriamente

elementos	Bubble Sort (Original)	Bubble Sort (Melhorado)	Heap Sort	Insertion Sort	Merge Sort	Quick Sort (pivô inicial)	Quick Sort (pivô central)	Selection Sort	Shell Sort
5	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**
10	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**	**0.000001**
20	**0.000002**	**0.000002**	**0.000002**	**0.000001**	**0.000002**	**0.000001**	**0.000001**	**0.000001**	**0.000001**
50	**0.000007**	**0.000006**	**0.000003**	**0.000002**	**0.000002**	**0.000002**	**0.000001**	**0.000003**	**0.000002**
100	**0.00002**	**0.00002**	**0.000007**	**0.000007**	**0.000005**	**0.000008**	**0.000004**	**0.000014**	**0.000004**
1000	**0.001737**	**0.001596**	**0.000105**	**0.000536**	**0.000049**	**0.000635**	**0.000035**	**0.001201**	**0.000089**
5000	**0.039472**	**0.039289**	**0.000758**	**0.013676**	**0.00033**	**0.016465**	**0.000447**	**0.026749**	**0.000701**
10000	**0.170846**	**0.168022**	**0.001647**	**0.054751**	**0.000659**	**0.068008**	**0.000974**	**0.104067**	**0.001527**
20000	**0.839108**	**0.841088**	**0.003357**	**0.212983**	**0.001486**	**0.215918**	**0.002066**	**0.413518**	**0.003405**
50000	**5.982549**	**5.990516**	**0.009704**	**1.32114**	**0.003818**	**1.810289**	**0.005604**	**2.574607**	**0.009867**
100000	**24.767637**	**25.025105**	**0.020276**	**5.29955**	**0.007956**	**4.729376**	**0.011931**	**10.287254**	**0.022166**

Bibliografia

CORMEN, Thomas H. *Introduction to Algorithms*.

ELLER, Danilo. *Slides de Aula*.

USP Panda. *Shell Sort*. Disponível em: https://panda.ime.usp.br/panda/static/pythonds_pt/05-OrdenacaoBusca/OShellSort.html. Acesso em: 11 nov. 2024.

JOÃO ARTHUR. *Quick Sort*. Disponível em: https://joaoarthurbm.github.io/eda/posts/quick-sort/. Acesso em: 11 nov. 2024.