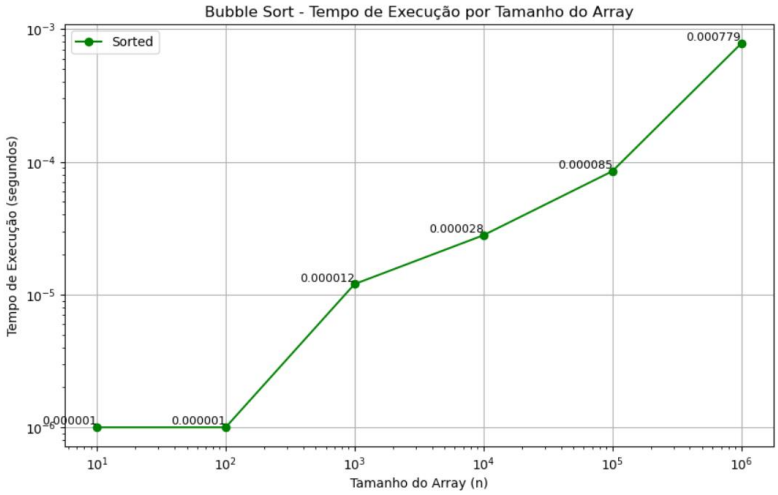
RELATÓRIO DA ANÁLISE DOS ALGORITMOS DE ORDENAÇÃO

BRENO BERNAL MARQUES

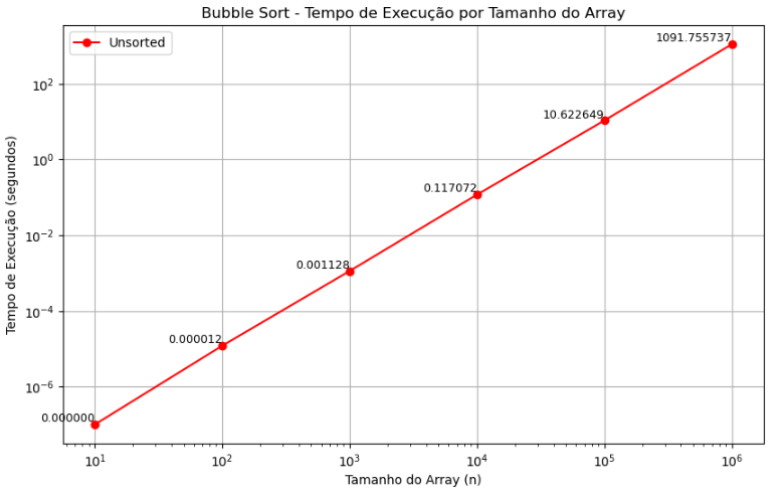
GABRIEL MARTINS BRUM

1 – Gráficos

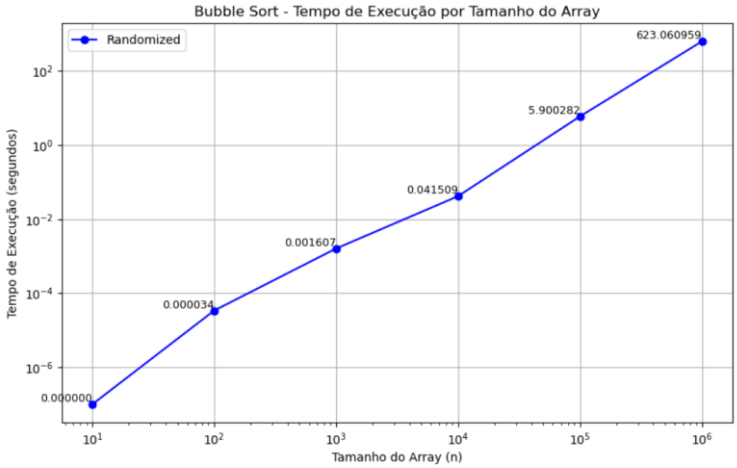
- Bubble Sort – $O(n^2)$
  - Melhor caso



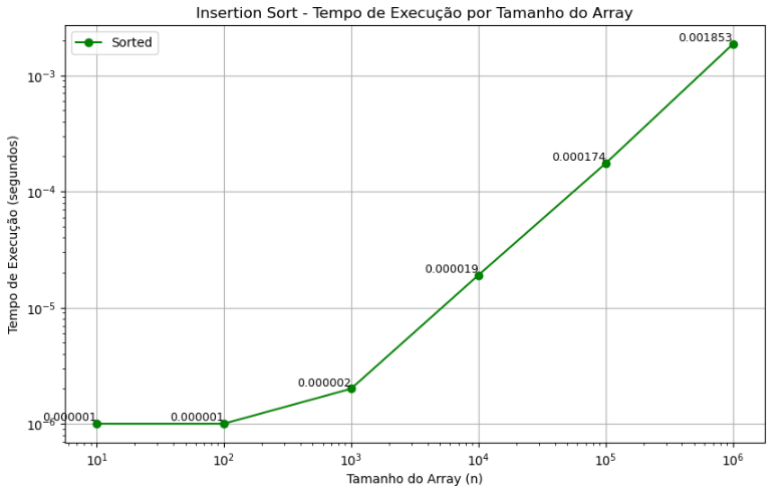
  - Pior caso



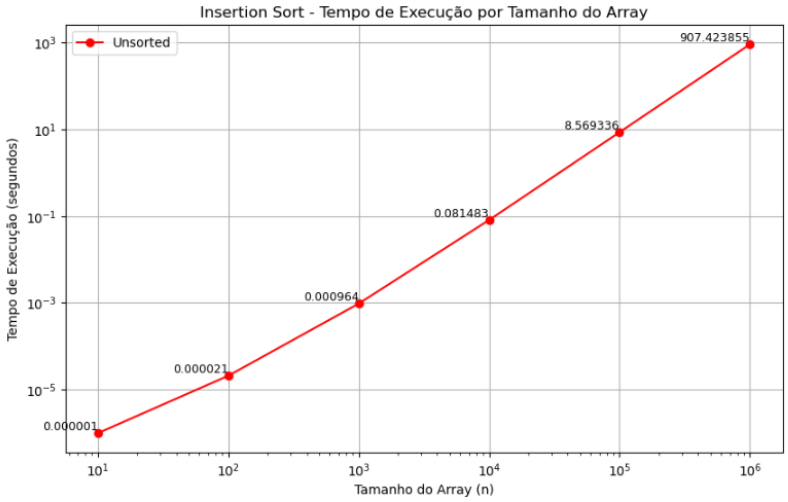
  - Caso aleatório



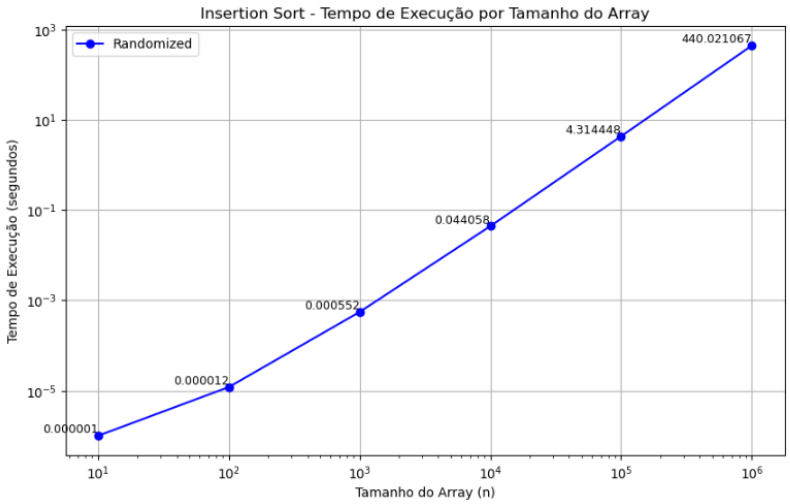
- Insertion Sort – $O(n^2)$
  - Melhor caso



  - Pior caso

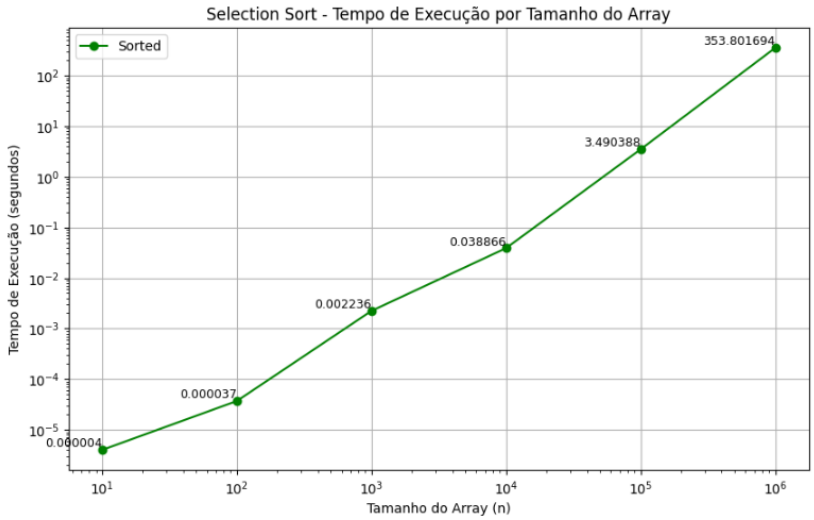


  - Caso aleatório

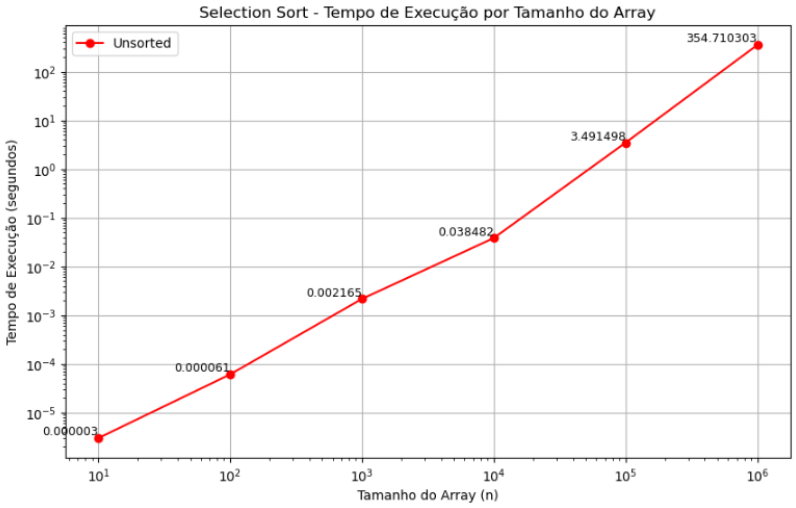


- Selection Sort – $O(n^2)$

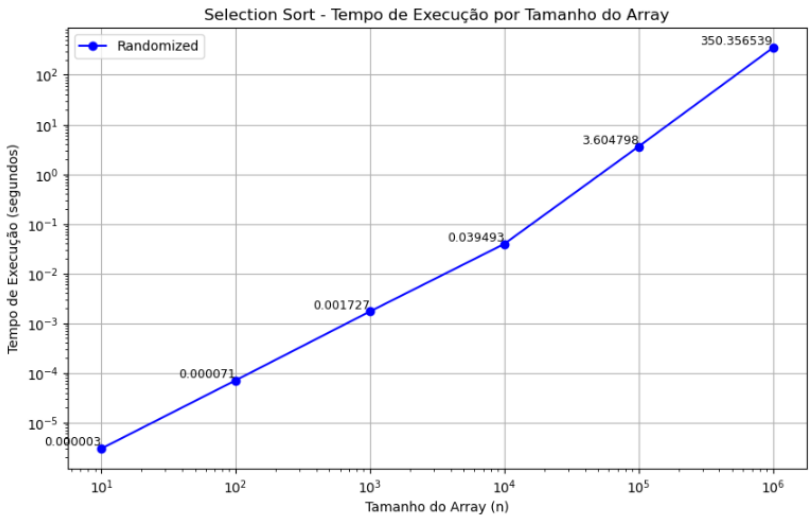
- Melhor caso



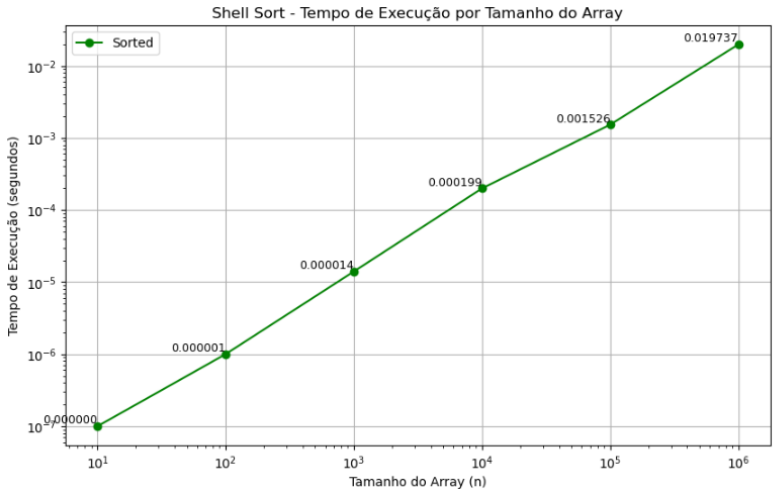
- Pior caso



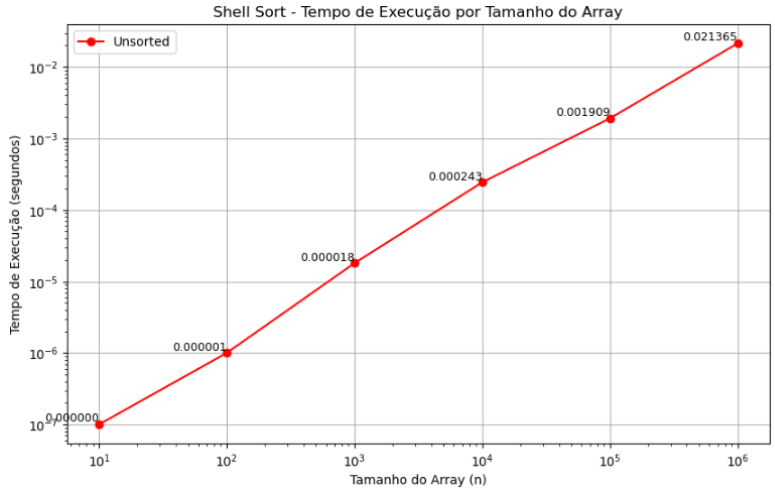
- Caso aleatório



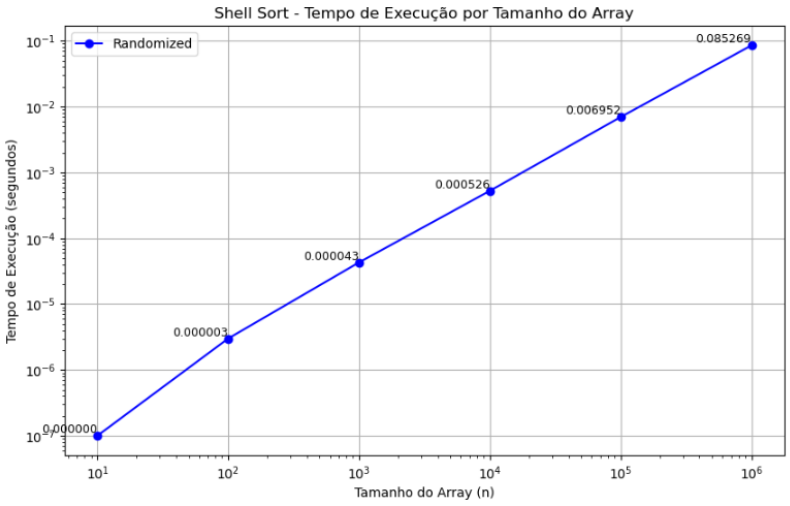
- Shell Sort (sequência de Knuth) – $O(n^{3/2})$
  - Melhor caso



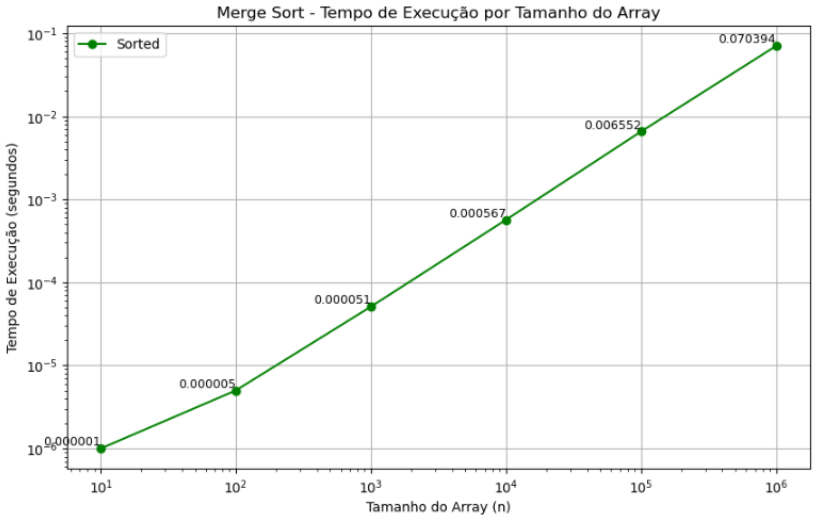
  - Pior caso



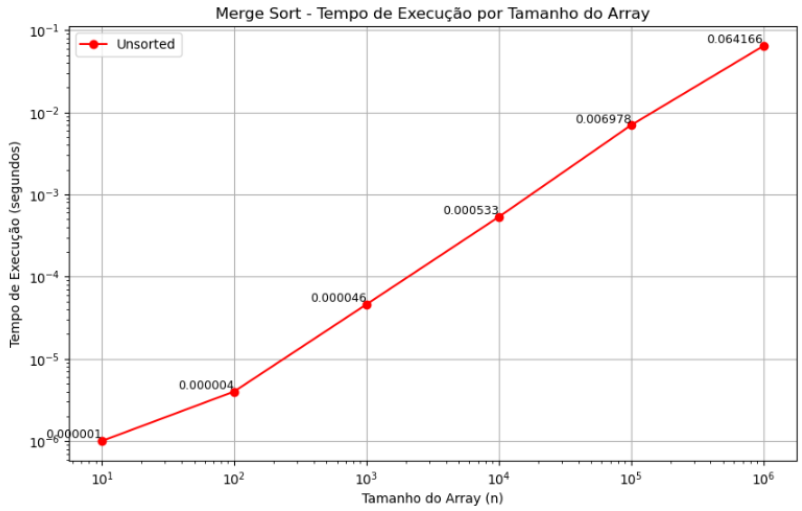
  - Caso aleatório



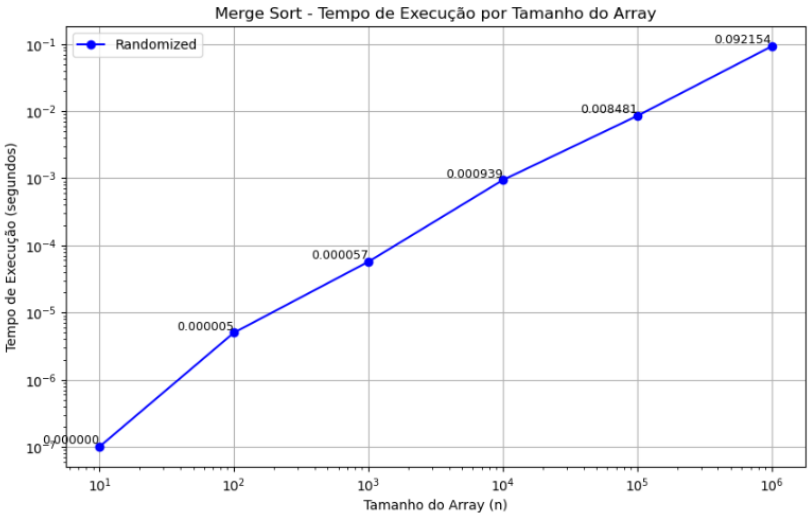
- Merge Sort – O(n log n)
  - Melhor caso



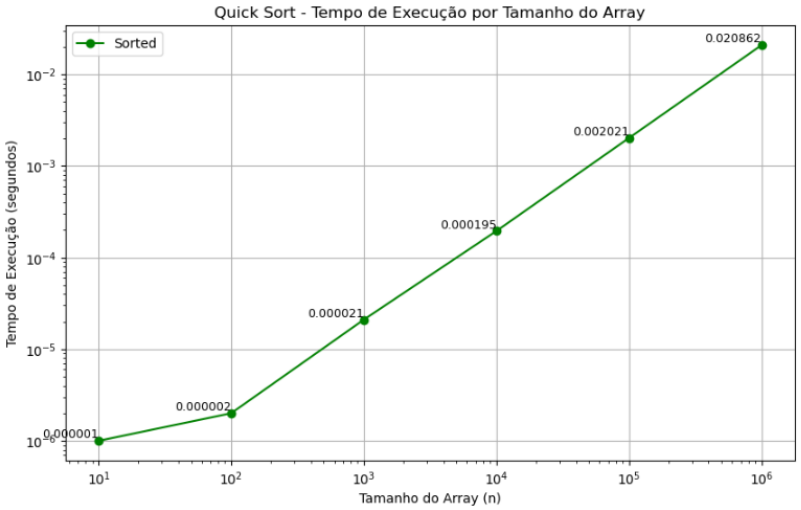
  - Pior caso



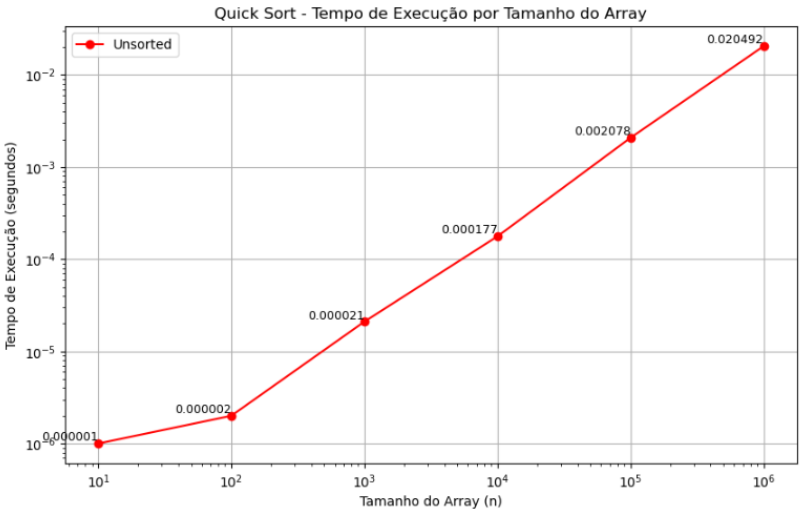
  - Caso aleatório



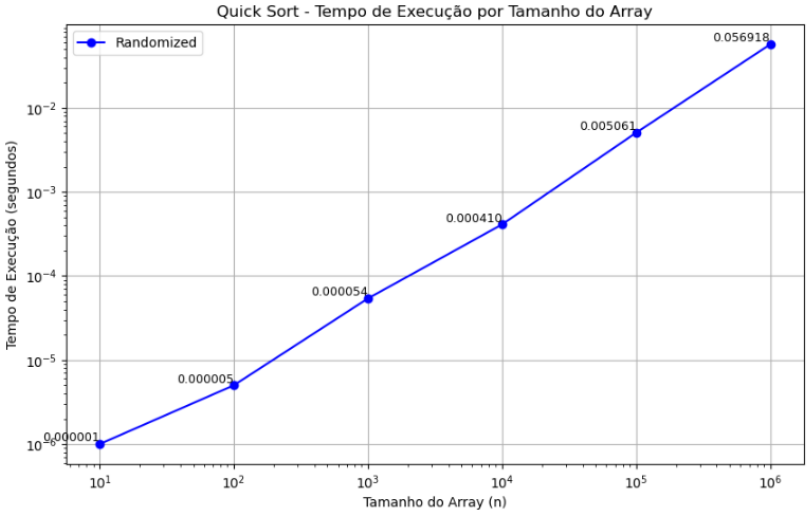
- Quick Sort – O(n log n)
  - Melhor caso



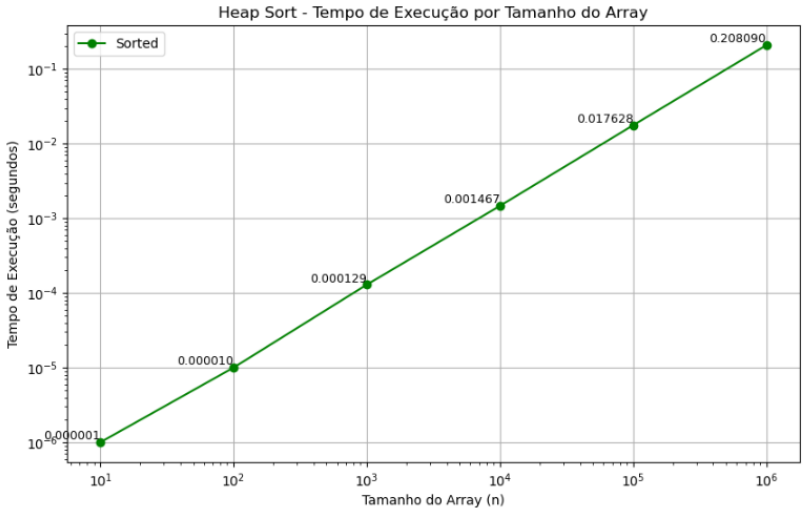
  - Pior caso



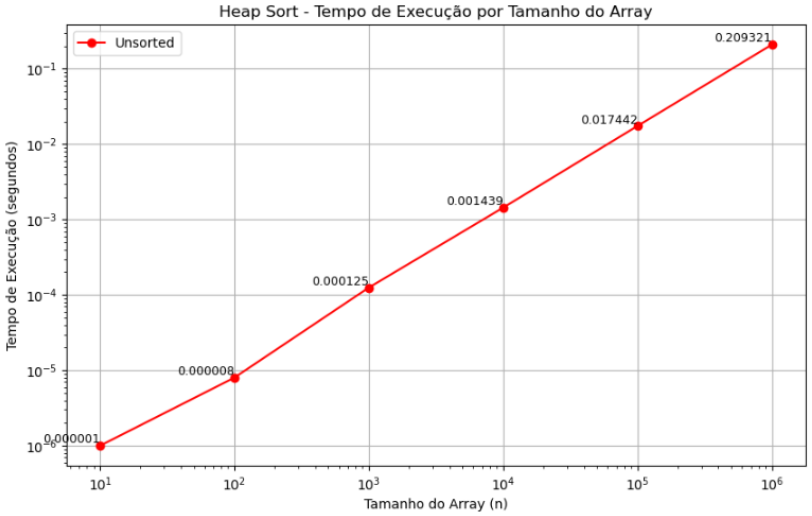
  - Caso aleatório



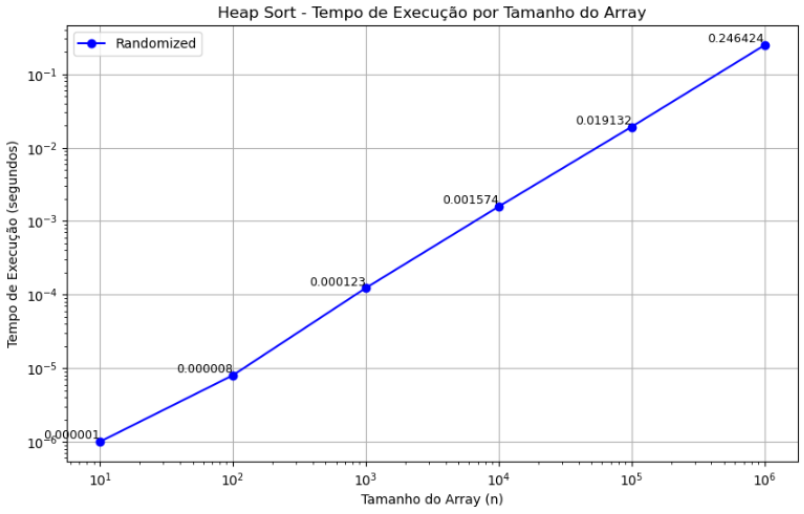
- Heap Sort – O(n log n)
  - Melhor caso



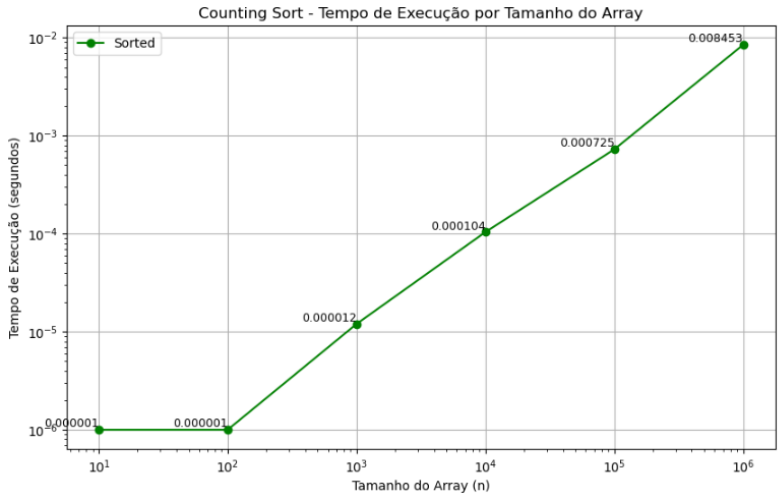
  - Pior caso



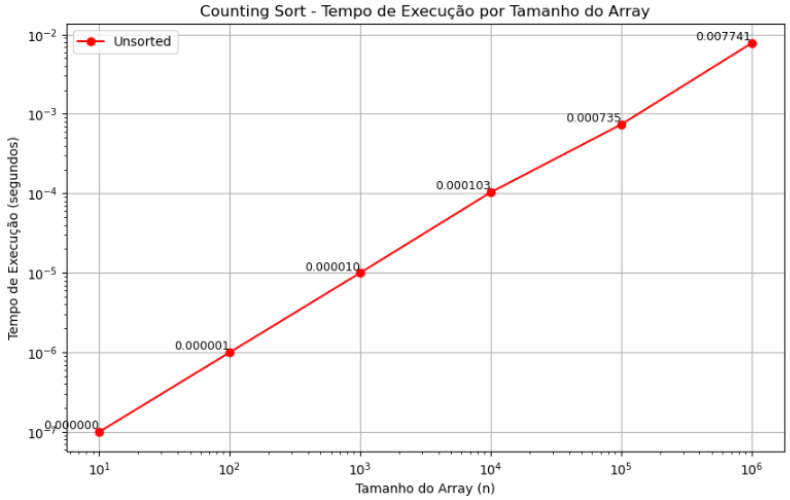
  - Caso aleatório



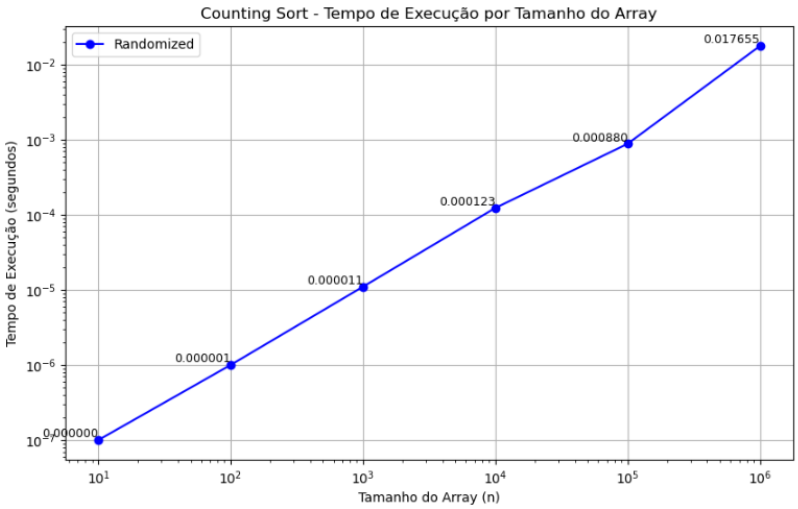
- Counting Sort – O(n)
  - Melhor caso



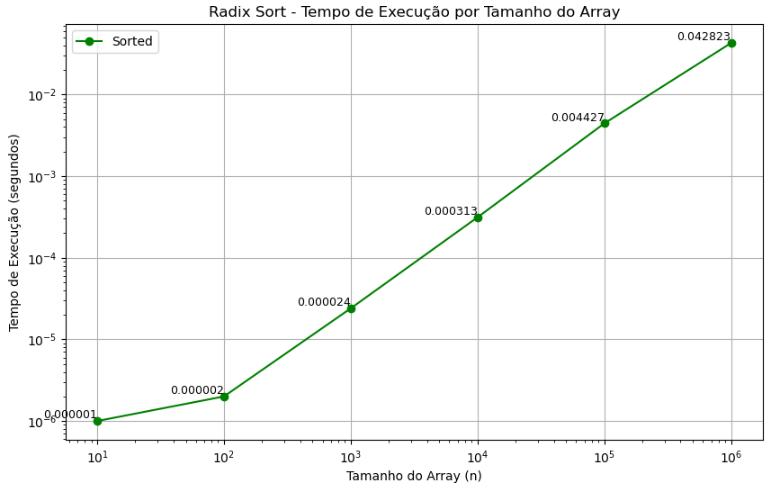
  - Pior caso



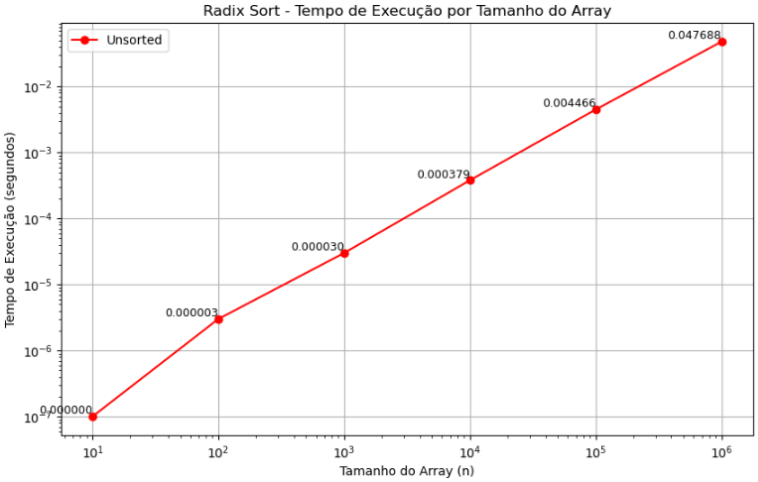
  - Caso aleatório



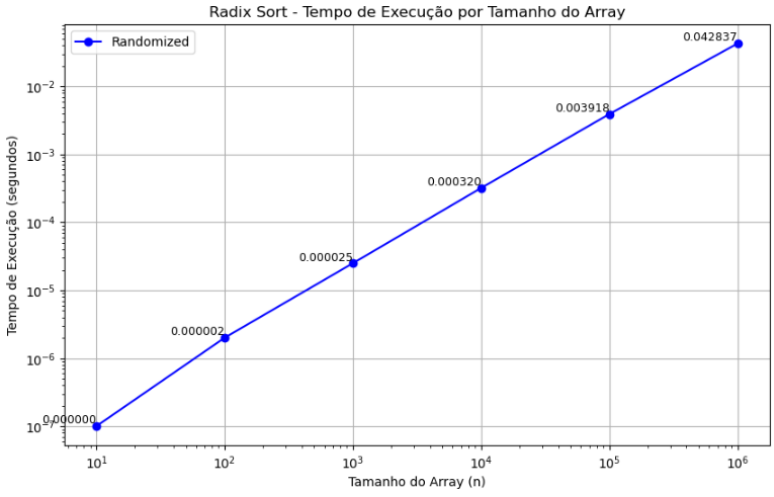
- Radix Sort – O(n)
  - Melhor caso



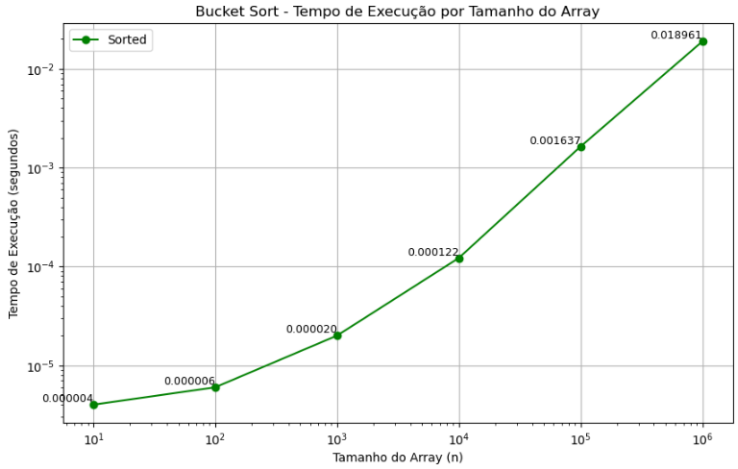
  - Pior caso



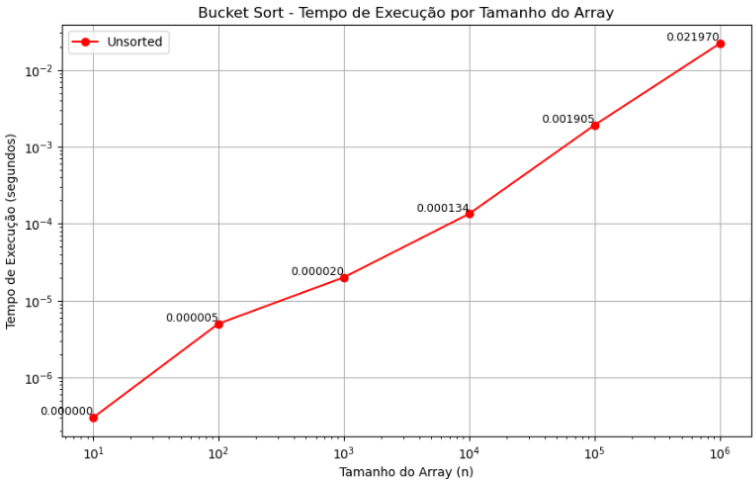
  - Caso aleatório



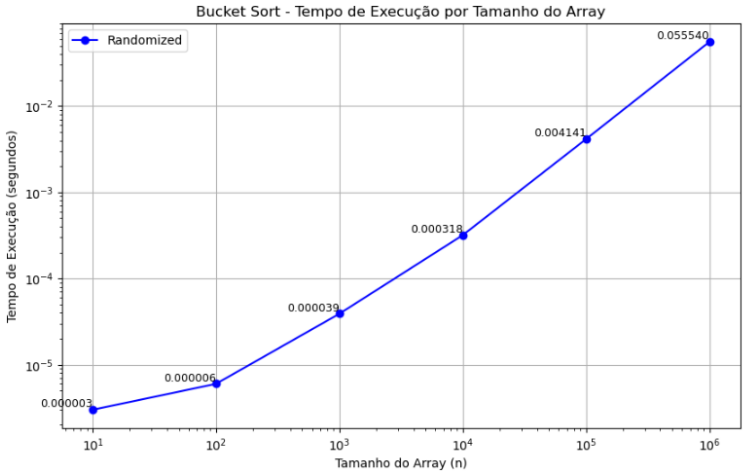
- Bucket Sort – O(n)
  - Melhor caso



  - Pior caso

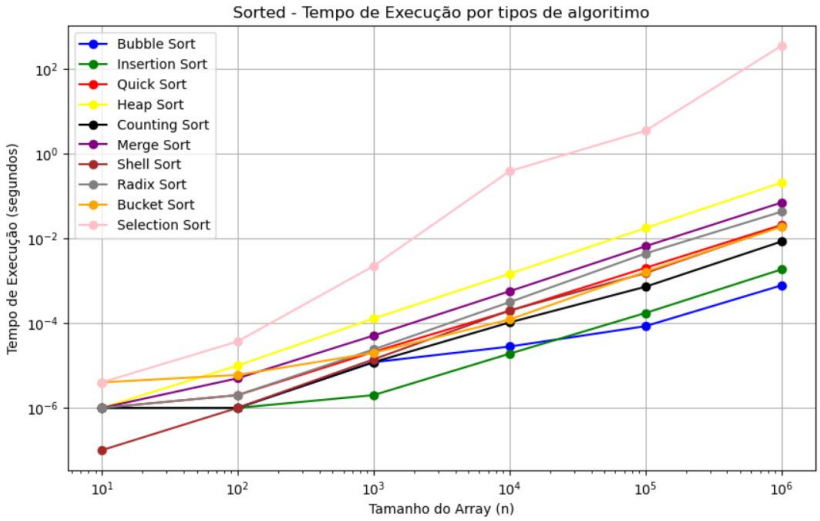


  - Caso aleatório

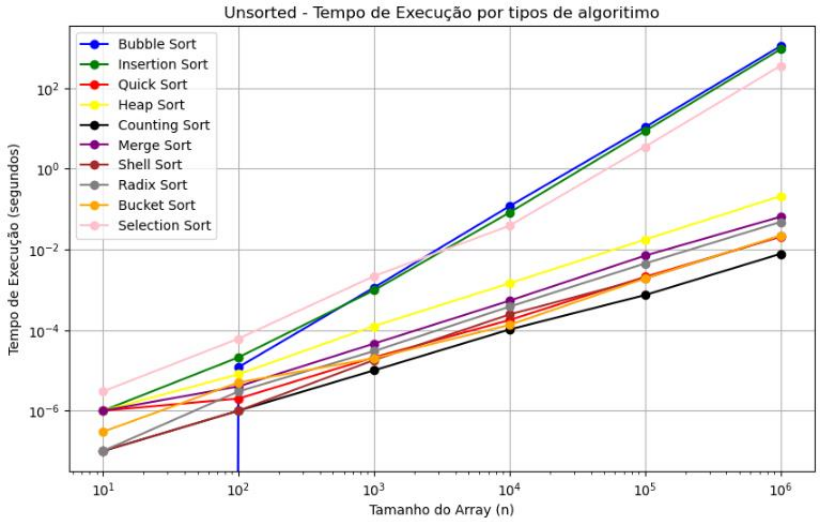


2 - Comparação dos algoritmos

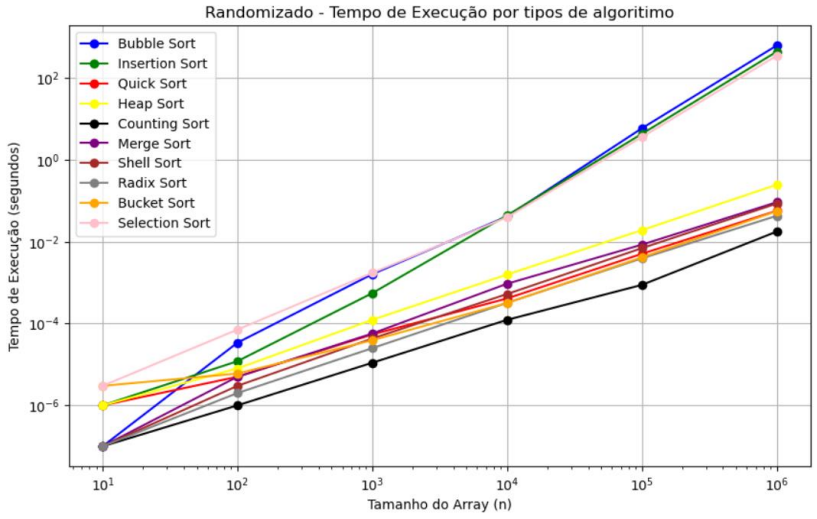
Melhor caso



Pior caso



Caso randômico



3 – Análise dos resultados

3.1 Análise individual dos algoritmos

Para discutir o comportamento dos algoritmos, iremos dividir os algoritmos para realizar uma análise especializada para cada processo de ordenação, porém, inicialmente é necessário deixar claro que quando referimos ao melhor caso, estamos falando sobre o vetor já estar ordenado, já o pior caso é quando ele está ordenado de forma decrescente, e por último, o caso aleatório é quando o vetor é preenchido com valores aleatórios.

Primeiramente, o Bubble Sort apresentou resultados que seguiram fielmente a eficiência teórica, pois em seu melhor caso ($\Omega(n)$), ele segue a ordem n, onde ao aumentar o tamanho do array 10 vezes, o tempo de execução não aumentava mais do que 10 vezes. Já no pior caso ($O(n^2)$), o aumento seria quadrático, ou seja, ao aumentar o array 10 vezes, o tempo aumentaria 100 vezes, e isso foi constatado e amostrado pelo gráfico exposto anteriormente. Olhando para o caso aleatório, o comportamento deveria estar entre o melhor caso e pior caso, e assim ocorreu.

O Insertion Sort também apresentou um comportamento alinhado com sua eficiência teórica. No melhor caso ($\Omega(n)$), resultando em um tempo de execução diretamente proporcional ao tamanho do array. Assim, ao aumentar o array 10 vezes, o tempo de execução aumentou na mesma proporção. Por outro lado, no pior caso ($O(n^2)$), o algoritmo realizou o máximo de comparações e trocas, resultando em um crescimento quadrático no tempo de execução — um aumento de 100 vezes no tempo ao ampliar o array em 10 vezes, conforme observado nos resultados empíricos. Para o caso aleatório, o desempenho ficou entre esses dois extremos, conforme esperado, com o tempo de execução seguindo uma tendência próxima ao caso quadrático.

Partindo para o Selection Sort, cuja complexidade é $O(n^2)$, observamos que os três casos obtiveram resultados parecidos, sendo que foi um pouco surpreeendente de início, porém ao analisar o procedimento utilizado no algoritmo, entende-se que a sua complexidade é definida pelo varrimento do vetor por completo, independentemente de como ele está ordenado.

Analisando o Shell Sort, obtivemos que no melhor caso ($\Omega(n \log n)$) ele cresceu de maneira linear, onde realmente segue a teoria, em que ao aumentar 10 vezes o tamanho do vetor, o tempo também aumentaria 10 vezes. No pior caso ($O(n^{3/2})$) obtivemos valores

bem próximos a linearidade, o que realmente é esperado devido a sua complexidade. Já no caso aleatório, observamos uma mudança de panorama, onde o aleatório demandou mais tempo para ser processado, porém continua seguindo a ordem de complexidade do algoritmo, que em que pior caso é próximo a linearidade.

No Merge Sort, ao contrário de muitos outros algoritmos, o melhor e pior caso (vetores ordenados crescente e decrescente) apresentaram tempos melhores do que o caso aleatório. Isso pode ser explicado pelo padrão de acesso à memória do algoritmo. Em vetores ordenados, os dados podem estar mais próximos na memória, resultando em menos cache misses. Já em vetores aleatórios, a mesclagem dos subarrays força o processador a acessar regiões da memória menos previsíveis e mais distantes, aumentando a quantidade de cache misses, o que causa um tempo de execução maior.

O algoritmo de Quick Sort apresentou resultados parecidos no melhor caso e no pior caso, isso é causado por conta da maneira com que escolhemos o pivô divisor. Esse é escolhido pela posição central do array, a qual é a mediana no melhor, e pior, caso. Então, ótimos resultados são encontrados. Entretanto, para casos aleatórios, ele obteve resultados quase três vezes mais lentos (mas manteve a complexidade espearada O(n log n)), portanto, para utilizar esse algoritmo da maneira mais eficiente, seria necessário utilizar o pivô sendo escolhido de maneira aleatória.

Ao verificar os tempos marcados pela ordenação do Heap Sort, verificamos que no melhor caso e no pior caso seus tempos são praticamente idênticos. Isso ocorre por conta do bom desempenho e baixo número de trocas necessárias para levar o maior valor ao topo da árvore binária. Já para o caso aleatório, foi onde tivemos os maiores valores, portanto, compreendemos que no caso aleatório tivemos uma disposição em que ao se montar a árvore binária, muitas trocas foram necessárias para levar o maior valor ao topo, entretanto, a ordem de grandeza do algoritmo nunca ultrapassou O(n log n).

No Counting Sort obtivemos resultados parecidos nos três casos. A razão para isso é que nesse algoritmo a ordem em que os valores estão dispostos não é tão relevante para sua complexidade de ordenação. Primeiramente, é realizado uma contagem de ocorrências de cada valor, e isso é semelhante em todos os casos. Após isso, define-se o index limite de cada valor encontrado. Logo, a ordem dos números não é tão relevante nesse algoritmo e pode-se observar a sua grande O(n + k), onde k é o maior valor dentro do array.

Assim como o Counting Sort, no Radix Sort a ordem em que os valores estão no vetor pouco importa, então, assim como observamos no gráfico, os três casos tiveram resultados muito próximo e todos seguiram a complexidade esperada: O(n + k) - utiliza o counting sort, então o k é o maior valor dentro do array. O motivo de a ordem dos valores não ser tão relevante, é por conta de essa ordenação utilizar os digitos dos valores, começando do mais significativo até o menos significativo.

No Bucket Sort, ao contrário do Counting Sort e Radix Sort, a ordem dos valores pode impactar significativamente o desempenho. Nos testes, o vetor aleatório teve pior desempenho comparado aos casos ordenados (crescente e decrescente) devido à distribuição desbalanceada dos elementos entre os baldes. Em vetores ordenados, os elementos são distribuídos de maneira mais uniforme, permitindo uma ordenação interna mais eficiente. Já em vetores aleatórios, alguns baldes ficam sobrecarregados enquanto outros permanecem quase vazios, resultando em maior tempo de execução, mas sempre respeitandoa complexidade de O(n+k).

3.2 – Análise geral

Ao analisar os últimos três gráficos, observamos que para o caso ordenado, obtivemos melhores resultados com o Bubble Sort e o Insertion Sort, os quais poderiam aparentar levar um tempo maior, pois ambos tem $O(n^2)$, entretanto, no melhor caso eles obtêm um desempenho de $\Omega(n)$. Já olhando para o pior resultado obtido, temos que o selection sort obteve o pior resultado, o que é compreensível ao analisar que independentemente da maneira que o vetor estiver ordenado, a sua complexidade será de $O(n^2)$.

O pior caso, temos que Insertion Sort e Bubble Sort obtiveram os piores resultados, logo em seguida tivemos o Selection Sort. Os resultados são coerentes pela razão da complexidade já exposta. Ademais, os dois piores tiveram esse tempo pela quantidade de comparações e *swaps* necessários para ordenação. Por outro lado, os melhores algoritmos fooram o Counting Sort, Quick Sort e Bucket Sort. O Quick Sort tem uma complexidade maior que os outros dois, porém, por conta da escolha do pivô ser a própria mediana, os resultados obtidos foram próximos de algoritmos de ordem linear.

Por fim, o caso aleatório resultou em valores próximos ao pior caso, entretanto, o Radix Sort teve melhores resultados e o Quick Sort perdeu desempenho, pelo motivo de o pivô não ser a mediana dos valores presentes no vetor.

4 – Conclusão

Concluindo o trabalho de análise e implementação de algoritmos de ordenação, observamos que a relação entre o comportamento prático e a teoria de complexidade dos algoritmos foi confirmada pelos experimentos. Cada algoritmo teve seu desempenho variado de acordo com a natureza do vetor de entrada — seja ele ordenado, em ordem decrescente ou aleatório — refletindo fielmente as complexidades de tempo teóricas esperadas.

Algoritmos como Bubble Sort, Insertion Sort e Selection Sort, com complexidade quadrática no pior caso ($O(n^2)$), mostraram-se ineficientes para grandes conjuntos de dados, especialmente em vetores não ordenados. No entanto, no melhor caso, o Bubble Sort e o Insertion Sort foram eficientes ($\Omega(n)$), uma vez que a quantidade de comparações e trocas é significativamente reduzida. Por outro lado, algoritmos de complexidade $O(n \log n)$, como o Merge Sort, Quick Sort e Heap Sort, apresentaram desempenho consistentemente superior, validando sua adequação para ordenações em grandes volumes de dados.

Os algoritmos baseados em técnicas especializadas, como Counting Sort, Radix Sort e Bucket Sort, mantiveram excelente desempenho, confirmando suas complexidades lineares em relação ao tamanho do vetor ($O(n + k)$). Eles se destacaram por sua eficiência quando comparados aos algoritmos quadráticos e até aos algoritmos de divisão e conquista em muitos casos.

Em síntese, a escolha do algoritmo de ordenação mais adequado depende fortemente do cenário específico de aplicação e do estado inicial dos dados a serem ordenados. Algoritmos quadráticos podem ser viáveis para pequenos conjuntos de dados ou para dados já ordenados, enquanto algoritmos mais eficientes, como Quick Sort e Merge Sort, são mais indicados para grandes volumes e situações em que o estado inicial dos dados é incerto. Assim, compreender a complexidade dos algoritmos é crucial para otimizar o desempenho e aplicar a melhor solução para cada caso.