ICC2 - Projeto 2

Nicolas de Sousa Maia
15481857

Caio Petroncini
15444622

19 de novembro de 2024

1 Introdução

Nesse relatório, nós faremos a comparação e análise dos resultados dos algoritmos de ordenação implementados no projeto, além da análise de seus comportamentos em cenários diferentes.

2 Resolução

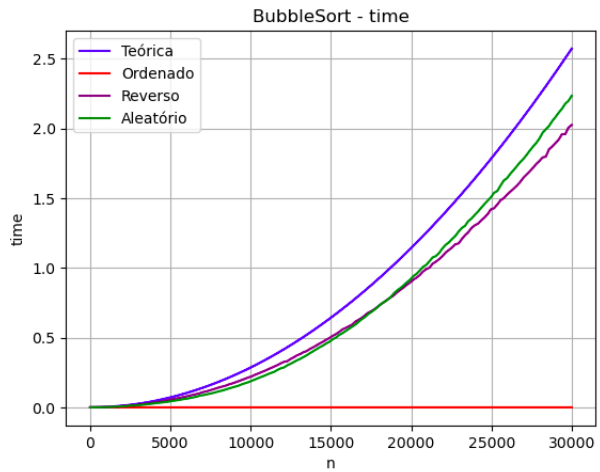
Nossos testes foram feitos com base em 3 tipos de vetores: aleatórios, ordenados e inversamente ordenados.

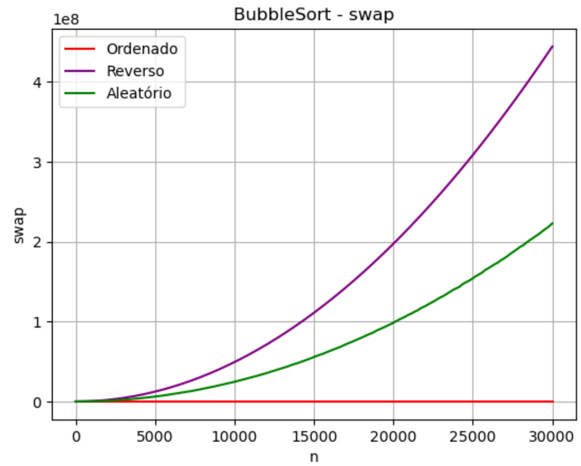
Para a implementação da resolução desse problema os algoritmos de ordenação foram, individualmente, implementados em funções, como está no código-fonte. Para a geração dos dados empíricos foram criadas funções auxiliares para avaliar o tempo de execução e gerar os vetores que foram usados nos testes. Já para a contagem das comparações e trocas de registros foi implementado um contador dentro de cada função de ordenação.

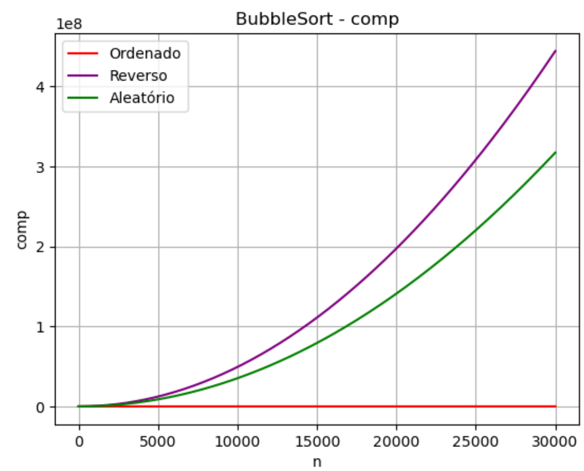
Os dados foram salvos em arquivos csv para análise e geração de informação estatísca posterior, sendo que para os dados dos vetores aleatórios foi feita a média de 5 casos para o mesmo n.

Por motivos práticos, devido a geração dos dados empíricos, os dados de $n = 100.000$ foram colocados no final de cada análise.

3 Bubble Sort







As linhas: Ordenado, Reverso, Aleatório, representam os dados empíricos coletados ao executar os algoritmos. Já a linha azul é a complexidade teórica. Por motivos práticos os valores de n variam de 1 à 30.000.

O Bubble Sort é um algoritmo simples que percorre o vetor repetidamente, comparando elementos adjacentes e os trocando de posição, se necessário. Ele possui complexidade $O(n^2)$ no caso médio e pior caso, sendo eficiente apenas para conjuntos pequenos ou quase ordenados.

O primeiro ponto que percebemos ao analisar os nossos dados é que a ordenação de um vetor ordenado no *BubbleSort* é muito menos custosa computacionalmente do que os outros tipos, o que é bem claro quando vemos o algoritmo, já que o loop é quebrado ao não se fazer nenhuma

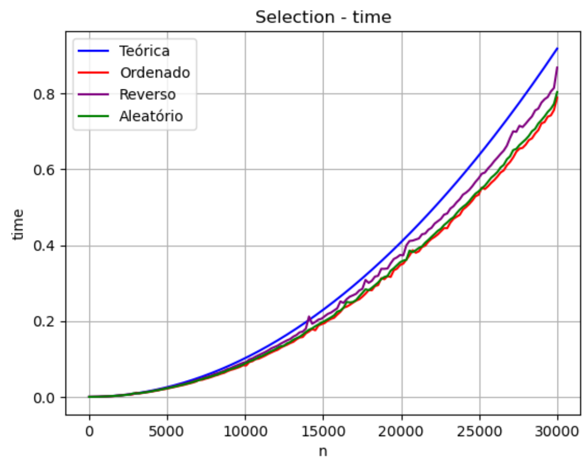
troca.

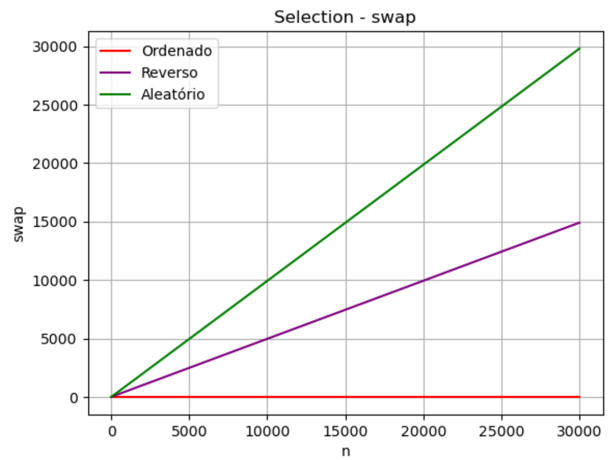
Já para os outros dois tipos, vemos que eles apresentam custo parecido com a complexidade teórica, o que é o esperado, com o Reverso sendo o pior caso para esse algoritmo, já que ele terá que realizar o número máximo de trocas, o que pode ser visto nos gráficos.

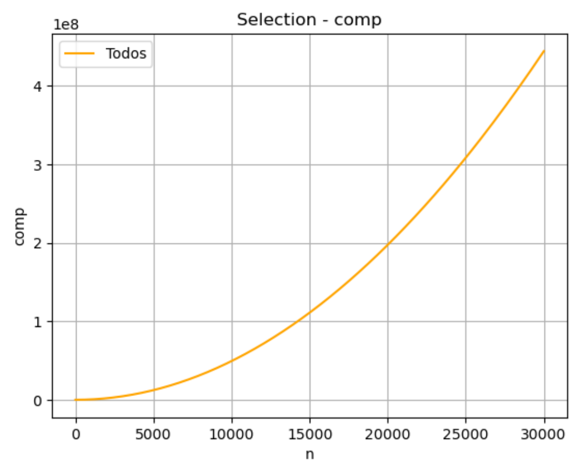
Dados para $n = 100.000$ **(tempo, comps, swaps), para vetores ordenados, reversamente ordenados e aleatórios, respectivamente:**

- 0.000261, 99999, 0
- 14.894536, 4999950000, 4999950000
- 18.071627, 4999892370, 2493390328

4 Selection Sort







O Selection Sort funciona selecionando o menor elemento de uma parte do vetor e o movendo para a posição correta a cada iteração. Sua complexidade é $O(n^2)$ em todos os casos, pois o número de comparações é constante, independente da ordem inicial dos elementos.

Ao analisarmos o gráfico de tempo, vemos que o tempo de execução para os 3 tipos é bem próximo, tanto entre se quanto do teórico, o que é o esperado do ponto de vista analítico, tendo em vista que, no $SelectionSort$, a complexidade será $O(n^2)$ não variando devido ao tipo do vetor, pois a seleção do menor a cada interação será feita sempre.

Analisando o gráfico dos *swaps* vemos que a linha do vetor ordenado é constante igual à 0, o que fica óbvio ao olhar o algoritmo, uma vez

que, a cada interação, o índice do menor já estará no local certo, não sendo necessário o *swap*.

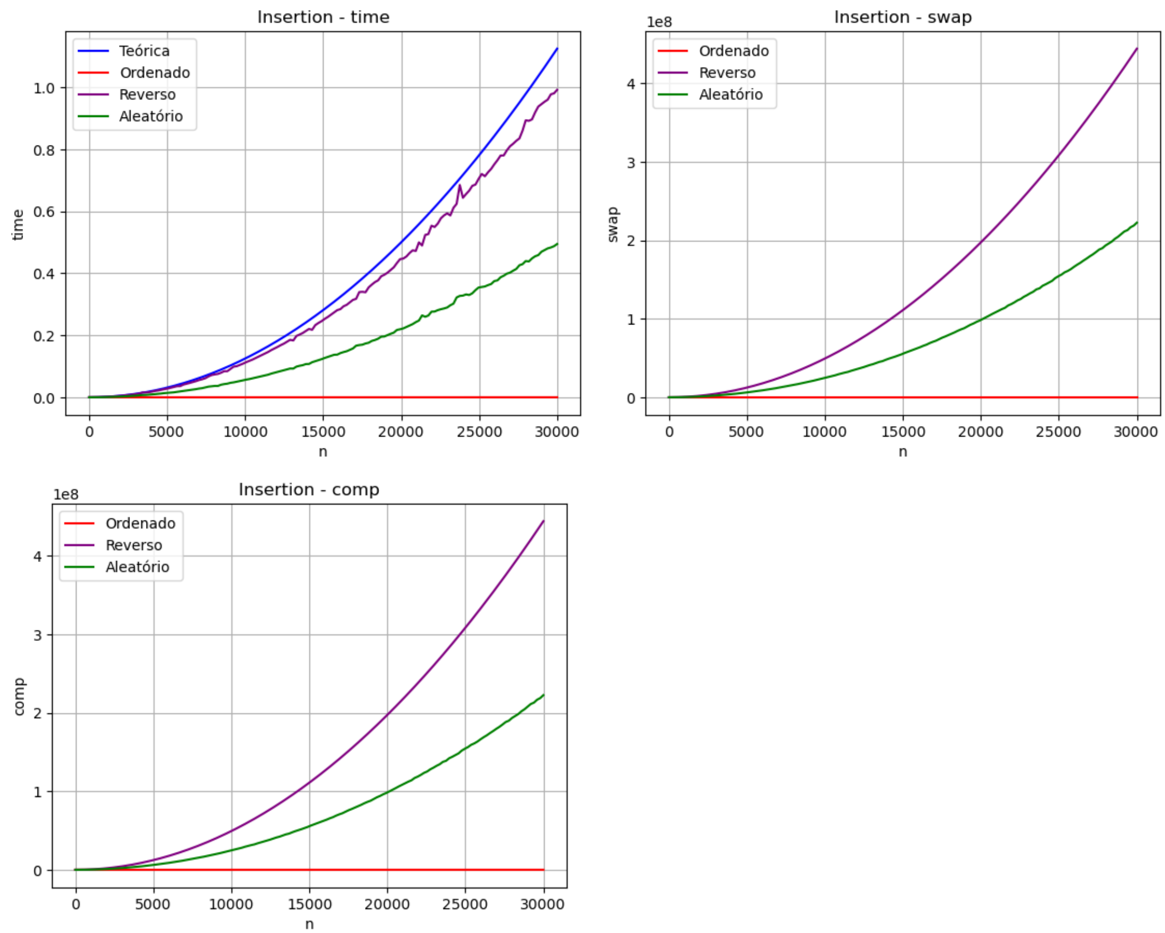
Agora, analisando as outras duas linhas, podemos perceber que a de vetores reversamente ordenados possúi um número de *swaps* menor do que a de vetores aleatórios, o que pode ser explicado ao percebemos que, a medida que vão sendo executados os *swaps*, o vetor reverso vai ficando ordenado, pois, o item de maior chave, que está no começo, é trocado com o de menor, que está no final, necessitando de apenas $\frac{n}{2} swaps$ para a ordenação total. Contudo, tal própriedade não existe no vetor aleatório, o que faz com que ela tenha esse comportamento.

Por fim, para o gráfico das comparações, podemos perceber que todos os casos possuem o mesmo número, devido ao fato que, em todos os cenários, o mesmo número de comparações será realizado em cada interação.

Dados para $n = 100.000$ (tempo, comps, swaps), para vetores ordenados, reversamente ordenados e aleatórios, respectivamente:

- 5.338802, 4999950000, 0
- 7.708148, 4999950000, 50000
- 5.488539, 4999950000, 99991

5 Insertion Sort



O Insertion Sort constrói o vetor ordenado gradualmente, inserindo elementos na posição correta. Ele é eficiente para conjuntos pequenos ou quase ordenados, com complexidade $O(n^2)$ no caso médio e pior caso, mas $O(n)$ no melhor caso.

Ao analisarmos o gráfico de tempo vemos duas peculiaridades, a primeira é que a linha de vetores ordenados teve um tempo muito menor do que os outros casos, o que é claro, uma vez que, no $InsertionSort$, nesse caso específico, ele teria complexidade $O(n)$, bem menor do que a complexidade de $O(n^2)$. Já a segunda peculidaridade é que a linha dos vetores reversamente ordenados tiveram um tempo bem maior do que todos os outros casos, o que também é explicado pelo algoritmo, pois, nesse cenário, ele sempre executará o maior número de

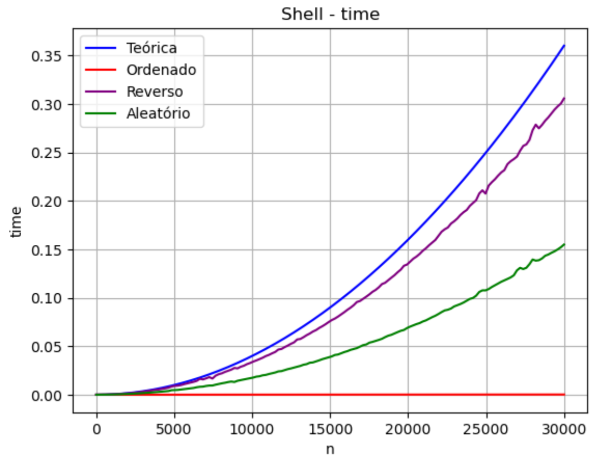
operações. Sendo assim, a linha dos vetores aleatórios um caso intermediário.

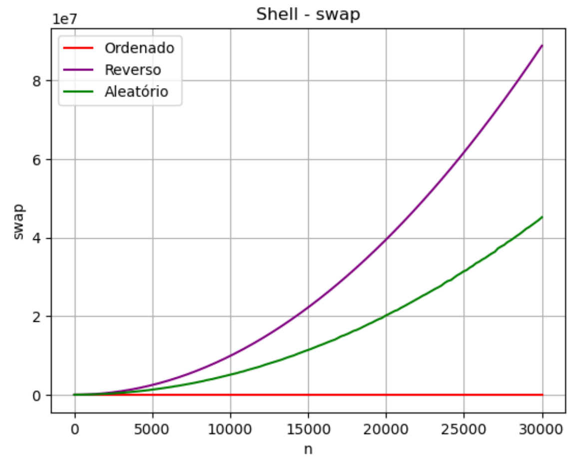
Nesse sentido, vemos que, tanto no gráfico das comparações quanto no gráfico dos $swpas$**, a mesma lógica se mantém.**

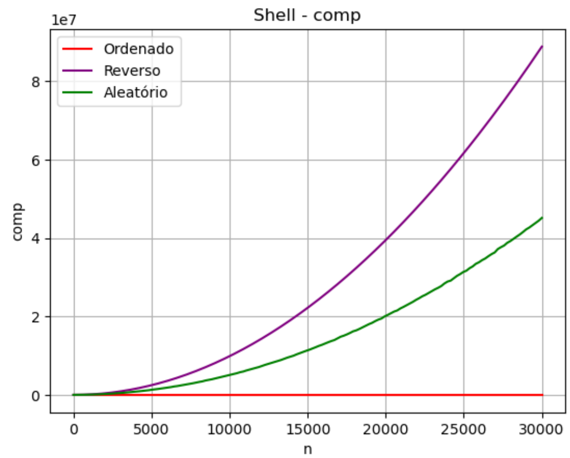
Dados para $n = 100.000$ **(tempo, comps, swaps), para vetores ordenados, reversamente ordenados e aleatórios, respectivamente:**

- 0.000254, 99999, 0
- 8.222522, 4999950000, 5000049999
- 4.517229, 2493490312, 2493490320

6 Shell Sort







O Shell Sort é uma generalização do Insertion Sort que melhora sua eficiência utilizando trocas em intervalos maiores (gaps), reduzidos gradativamente. Sua complexidade depende da sequência de gaps escolhida, geralmente variando entre $O(n\log(n))$ e $O(n^{\frac{3}{2}})$.

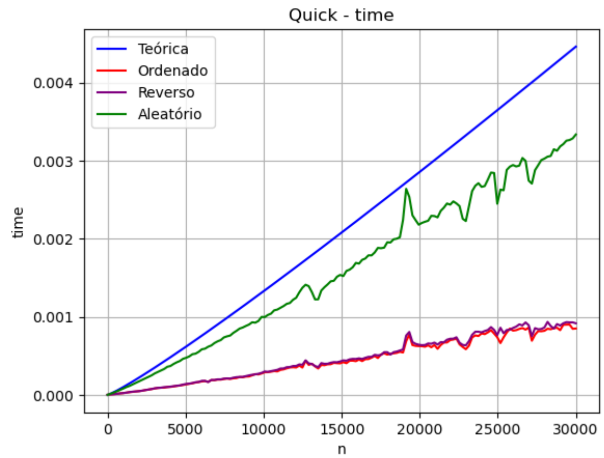
Dessa forma, seu comportamento é análogo ao do $InsertionSort$ para todos os gráficos, tendo, porém, um custo computacional menor, já que tal lista de índices torna o algorítmo mais efeciente, o que pode ser observado pelos dados numéricos. Lista de *gaps* utilizada {5,3,1}.

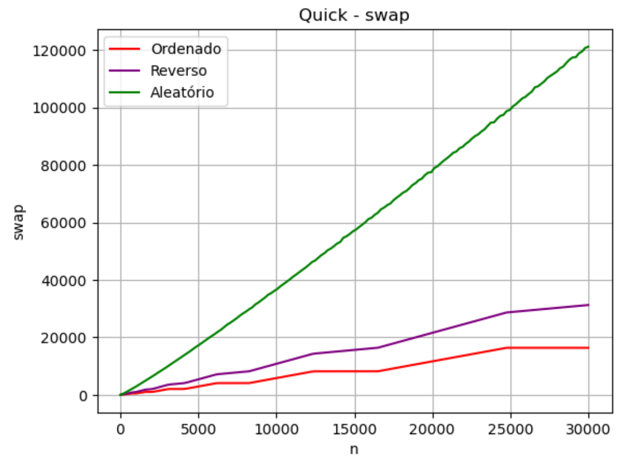
Dados para $n = 100.000$ (tempo, comps, swaps):

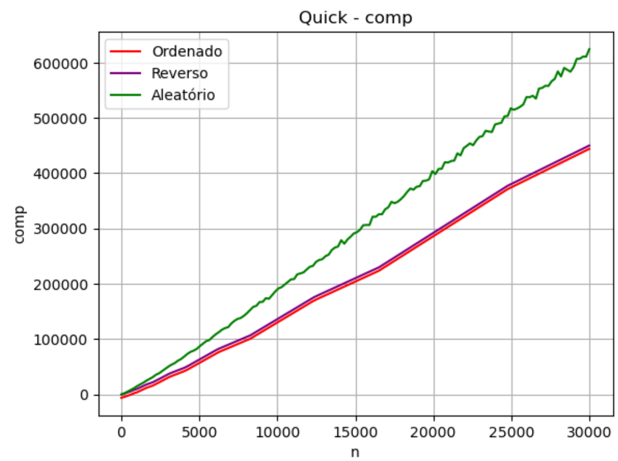
- 0.000990, 0, 0

- 2.226049, 1000030000, 1000209995
- 1.144935, 502914512, 503132092

7 Quick Sort







O Quick Sort é um algoritmo de dividir e conquistar que particiona o vetor em dois subvetores em torno de um pivô e ordena cada um recursivamente. Sua complexidade média é $O(n\log(n))$, mas pode alcançar $O(n^2)$ no pior caso, dependendo da escolha do pivô.

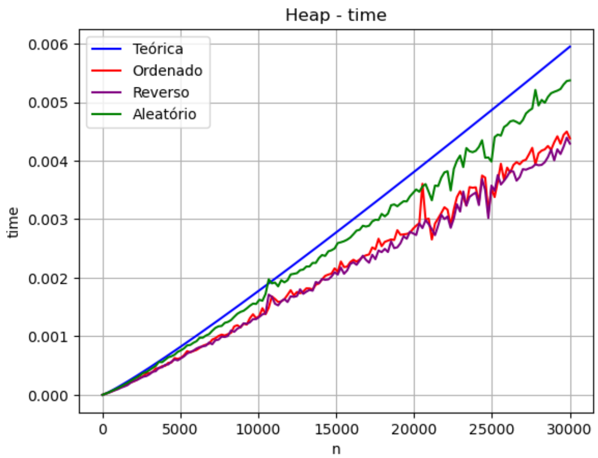
Ao analisarmos o gráfico de tempo vemos que a linha dos vetores aleatórios, mesmo com uma variação relevante, seguiu, de maneira geral, a complexidade teórica de $O(n\log(n))$, o que pode ser visto pelo gráfico. Já para os outros dois casos, vemos que elas tem um comportamento extremamente semelhante, o que é esperado, uma vez que, devido à escolha do pivô ser pela mediana, o pivô será o mesmo para os dois, tendo um comportamento espelhado ao ordenar.

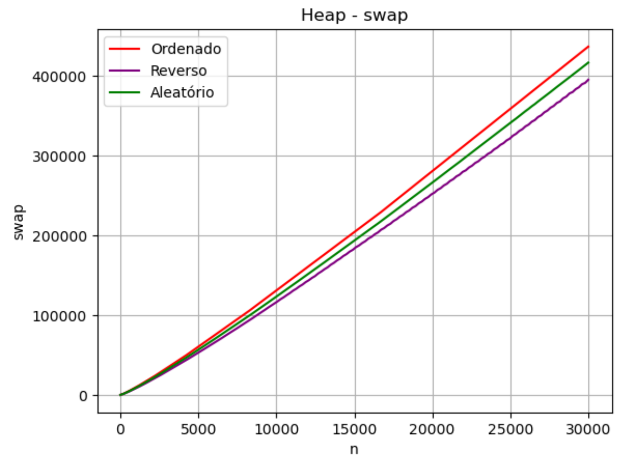
Os outros dois gráficos seguem essa mesma lógica, com a linha do gráfico dos aleatórios seguindo o esperado teórico e as dos ordenados e dos reseversamente ordenados tendo um número muito similar tanto de comparações quanto de *swaps*.

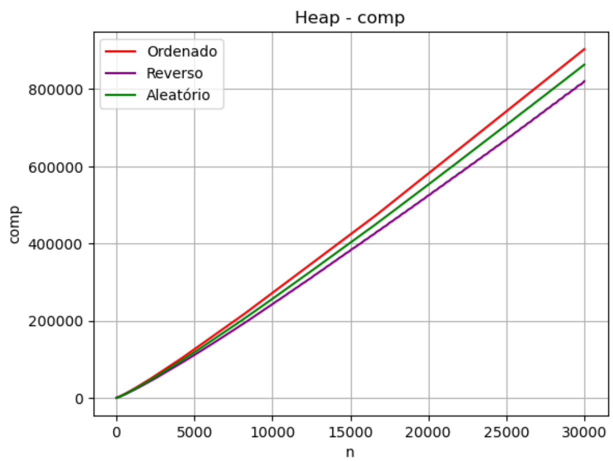
Dados para $n = 100.000$ (tempo, comps, swaps):

- 0.002819, 1731086, 65535
- 0.002903, 1731100, 115534
- 0.012733, 2639820, 434685

8 Heap Sort







O Heap Sort utiliza a estrutura de dados Heap (árvore binária completa) para ordenar o vetor, removendo o maior elemento da heap repetidamente. Ele possui complexidade $O(n \log(n))$ no pior caso e é eficiente para grandes conjuntos de dados.

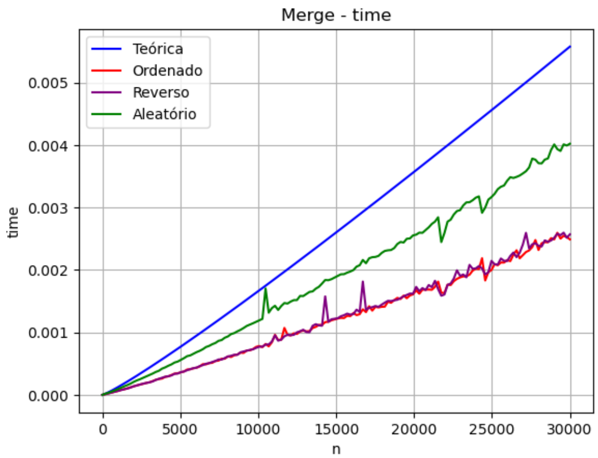
Ao analisarmos o gráfico de tempo vemos que todos os casos seguem de maneira geral a complexidade teórica $n \log(n)$, o que é o esperado, tendo em vista que no $HeapSort$ todos os casos terão esse custo computacional.

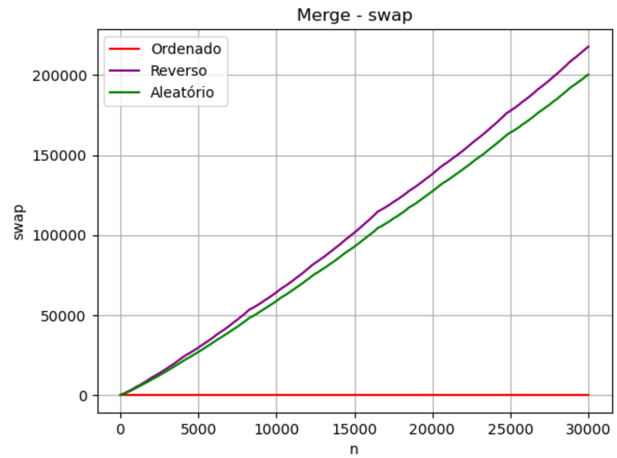
Podemos perceber que o mesmo acontece nos outros dois gráficos, com as linhas tendo valores e taxas de crescimento praticamente iguais.

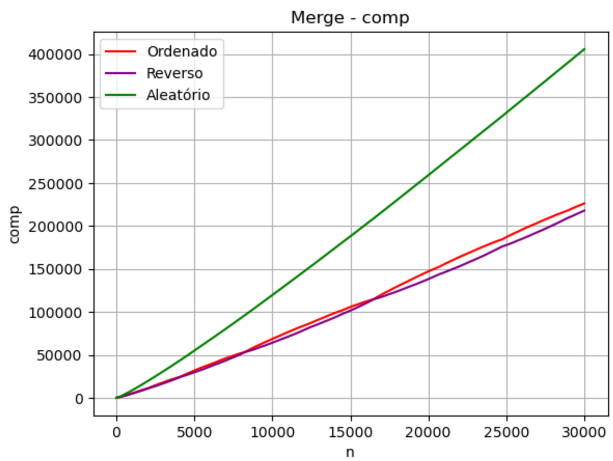
Dados para $n = 100.000$ (tempo, comps, swaps), para vetores ordenados, reversamente ordenados e aleatórios, respectivamente:

- 0.002264, 1731086, 65535
- 0.002451, 1731100, 115534
- 0.011308, 2639820, 434685

9 Merge Sort







O Merge Sort é um algoritmo de dividir e conquistar que divide o vetor ao meio, ordena as partes recursivamente e as intercala. Sua complexidade é $O(n \log(n))$ em todos os casos, sendo eficiente e estável, mas requer espaço extra para a intercalação.

Olhando para o gráfico de tempo, vemos que todas as curvas seguem a taxa de crescimento teórica descrita como $O(n \log(n))$, o que é o esperado, tendo em vista que, no $MergeSort$, a complexidade é constante, explicando esse fato.

Agora, para o gráfico dos *swaps*, nós consideremos como um *swap* quando, na função de intercalação, o item do vetor da direita é menor

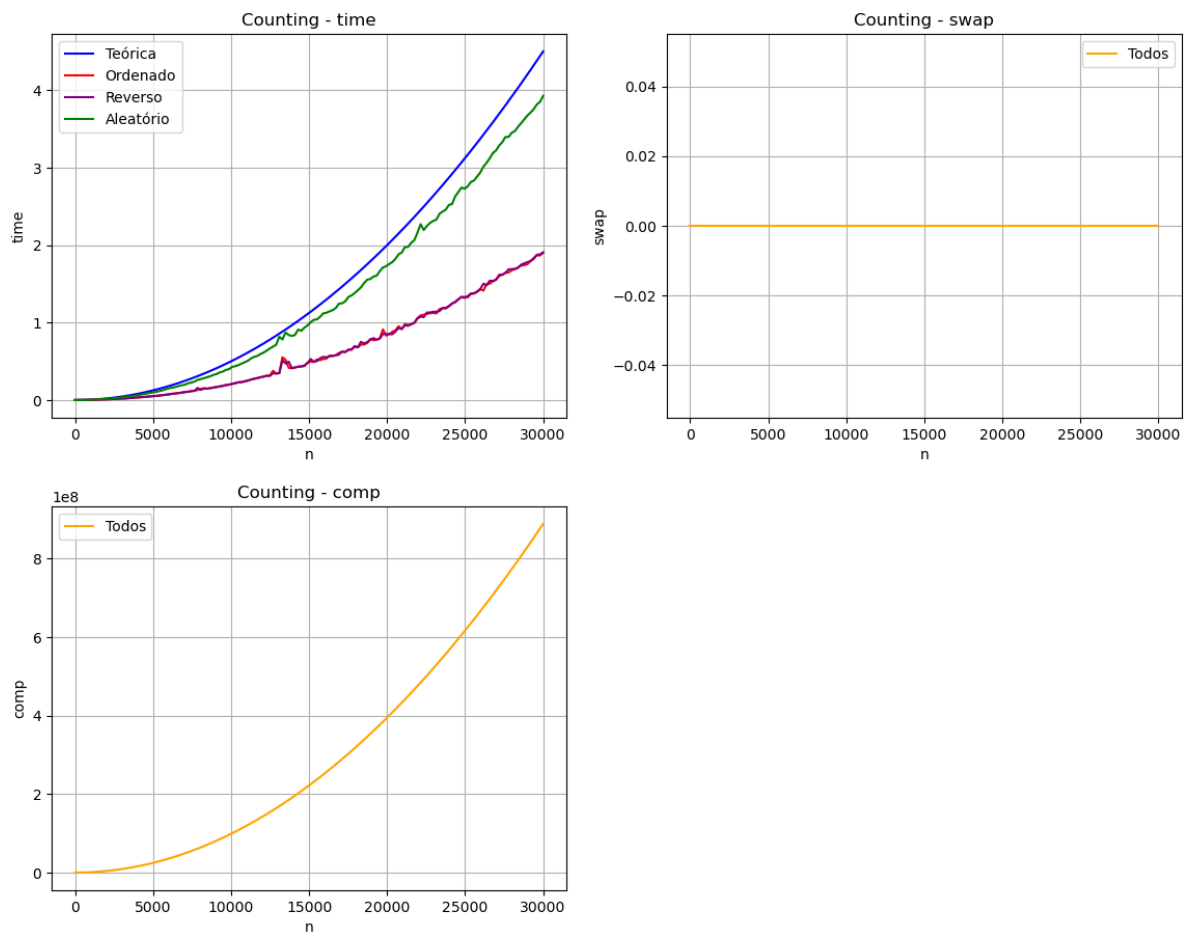
do que o da esquerda, fazendo com que ele venha antes no vetor final. Analisando os dados obtidos a partir disso, vemos que para o vetor ordenado não houve *swaps***, o que é óbvio, uma vez que os itens do vetor auxiliar esquerdo sempre serão menores que os do vetor auxiliar direito. Análogo a isso, podemos explicar o porque de os vetores reversamente ordenados tiveram um maior número de** *swaps***, já que os itens do vetor auxiliar esquerdo sempre serão maiores que os do vetor auxiliar direito. Sendo, por fim, os vetores aleatórios casos intermediários entre eles.**

Analisando o gráfico das comparações, a maneira como nós implementamos explica o fato de os vetores aleatórios terem um número de comparações maior do que os demais. Nós consideramos como uma comparação a ação de verificar se um item do vetor auxiliar esquerdo é menor ou igual à um item do vetor auxiliar direito, sem considerar a comparação com o sentinela, sendo assim, como, tanto no caso ordenado quanto no reversamente ordenado, essa comparação sempre chega ao sentinela, sem nem mudar o ponteiro do outro lado, o número de comparações é significativamente menor quando comparado ao caso dos vetores aleatórios, que seguem a complexidade estipulada de $O(n \log(n)$**.**

Dados para $n = 100.000$ **(tempo, comps, swaps), para vetores ordenados, reversamente ordenados e aleatórios, respectivamente:**

- 0.006200, 1668928, 1668928
- 0.006135, 1668928, 1668928
- 0.012662, 1668928, 1668928

10 Contagem dos Menores



O algoritmo Contagem dos Menores conta quantos elementos no vetor são menores que cada elemento e usa essa informação para colocá-los na posição correta. Sua complexidade é $O(n^2)$ devido às comparações entre todos os pares.

Analisando o gráfico de tempo, vemos que todos os casos seguem, de maneira geral, a taxa de crescimento téorica $O(n^2)$, sendo o caso dos vetores ordenados e reversamente ordenados praticamente iguais, o que é o esperado, tendo em vista que eles possuem os mesmos números.

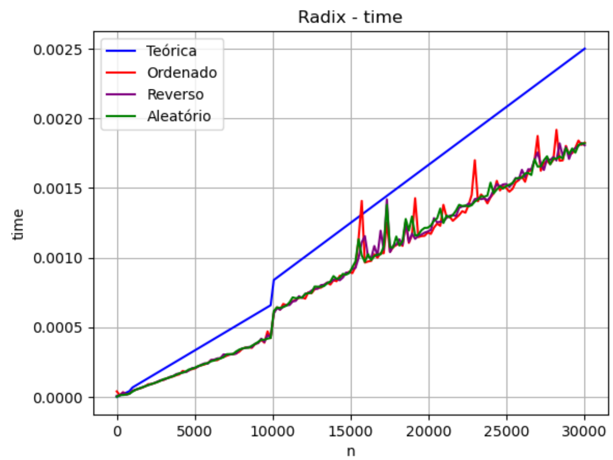
Como o algoritmo *Contagem dos Menores* não usa *swaps* para ordenar o vetor em todos os casos eles foram 0.

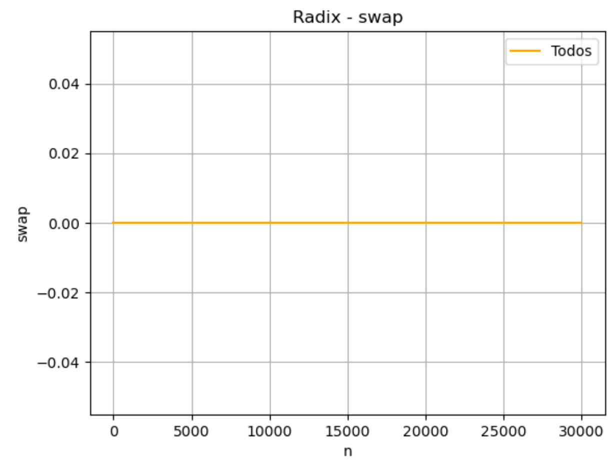
Agora, para o gráfico das comparações, vemos que ele segue a complexidade teórica $O(n^2)$ para todos os casos, já que todas as comparações serão executadas, independente do estado do vetor, fazendo com que as 3 sejão iguais.

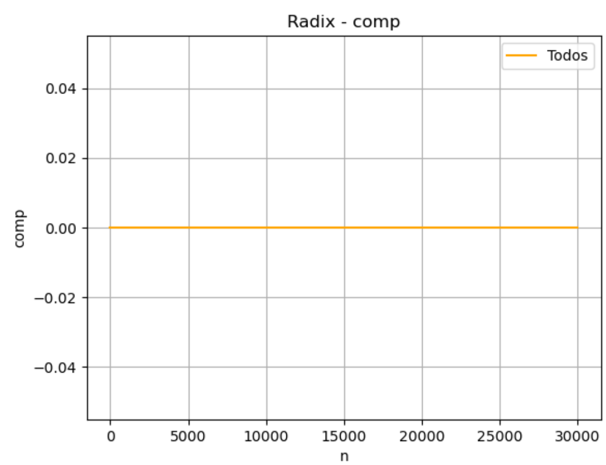
Dados para $n = 100.000$ (tempo, comps, swaps), para vetores ordenados, reversamente ordenados e aleatórios, respectivamente:

- 14.655896, 10000000000, 0
- 15.118312, 10000000000, 0
- 32.913155, 10000000000, 0

11 Radix Sort







O Radix Sort é um algoritmo não baseado em comparações que ordena números inteiros processando cada dígito separadamente, sendo eficiente para conjuntos com chaves de tamanho limitado.

Analisando o gráfico de tempo, podemos ver que os dados empíricos seguiram bem a complexidade teórica de $O(n \cdot m)$, onde m é o número máximo de dígitos que uma chave do vetor teria. Nesse caso, os números foram todos menores que o tamanho do vetor, devido a nossa implementação, fazendo com que a complexidade seja aproximandamente $O(n)$, mas isso poderia mudar muito caso não houvesse esse limite.

Agora, temos que destacar que tanto os *swaps* quanto as comparações foram 0, o que se explica pelo fato de que o algoritmo *RadixSort* não é baseado em comparações, além de que o método que ele usa não faz uso de *swaps* para ordenar o vetor.

Dados para $n = 100.000$ (tempo, comps, swaps), para vetores ordenados, reversamente ordenados e aleatórios, respectivamente:

- 0.004487, 0, 0
- 0.006188, 0, 0
- 0.004661, 0, 0

12 Comparação Final

Primeiramente, o que se saiu melhor quando o vetor já era ordenado foram o *BubbleSort* e o *InsertionSort*, já que, nesse caso, os dois tem complexidade $O(n)$, uma vez que o vetor está ordenado, fazendo com que eles tenham esse desempenho superior. Já o que teve o pior foi a *Contagem de Menores*, devido à sua complexidade constante de $O(n^2)$

Para os vetores reversamente ordenados, o que se saiu melhor foram o *QuickSort* e o *HeapSort*, os quais possuem complexidade $O(n\log(n)$ nesse caso, sendo 3 vezes mais eficiente que o 3º colocado, que foi o *RadixSort*. Já o pior foi novamente a *Contagem de Menores*, pelo mesmo motivo do anterior, tanto que seu tempo apenas um pouco superior a do *BubbleSort*, que, nesse cenário, está no seu pior caso.

Analisando os dados, podemos ver que o algoritmo que se saiu melhor no caso de um vetor aleatório foi o *RadixSort*, mas isso pois os números no vetor eram sempre menores que o tamanho dele, como foi dito em sua seção.Fora ele, o que se saiu melhor foi o *QuickSort*, o que é esperado para um caso geral de vetores aleatórios. Já o pior foi novamente a *Contagem de Menores*, pelo mesmo motivo anterior.