Ordenação

Integrantes:

- Caroline Akimi Kurosaki Ueda - 15445630
- Natalie Isernhagen Coelho - 15481332

Sobre o problema

O problema proposto é comparar os algoritmos de ordenação estudados com base em três critérios: a quantidade de movimentações de registros, a quantidade de comparações feitas e o tempo de execução relativos à ordenação de determinado conjunto. Para isso, cada algoritmo é testado com vetores de inteiros que diferem em tamanho e ordenação dos elementos (ordenado, inversamente ordenado ou aleatório).

Abordagem

Para coletar esses dados e comparar os algoritmos, um arquivo com cada tipo de vetor — variando em tamanho e configuração inicial dos elementos — foi gerado a partir de um script, somando 28 arquivos no total, já que foram gerados cinco arquivos diferentes com registros em ordem aleatória para cada tamanho.

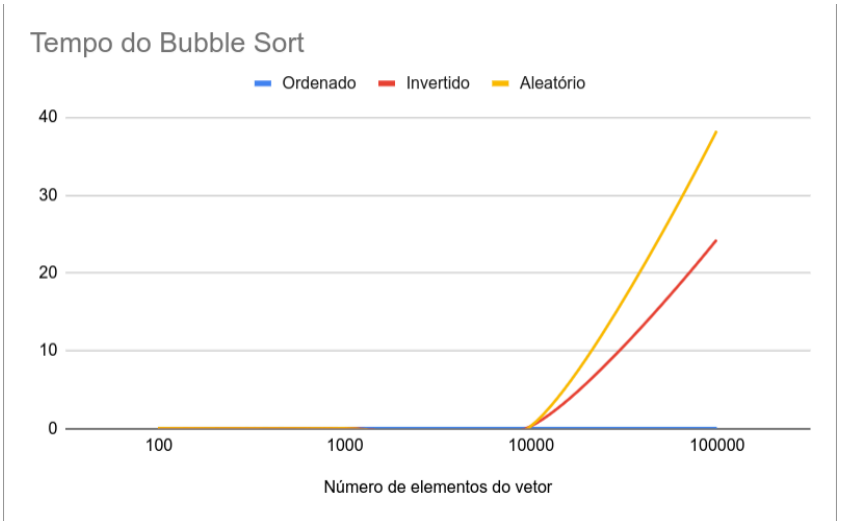
Então, em um arquivo foram reunidas as funções que implementam os algoritmos de ordenação, e a elas foram acrescentadas variáveis responsáveis por monitorar as quantidades de trocas e comparações, no formato de uma estrutura que é passada por referência nos parâmetros de cada algoritmo.

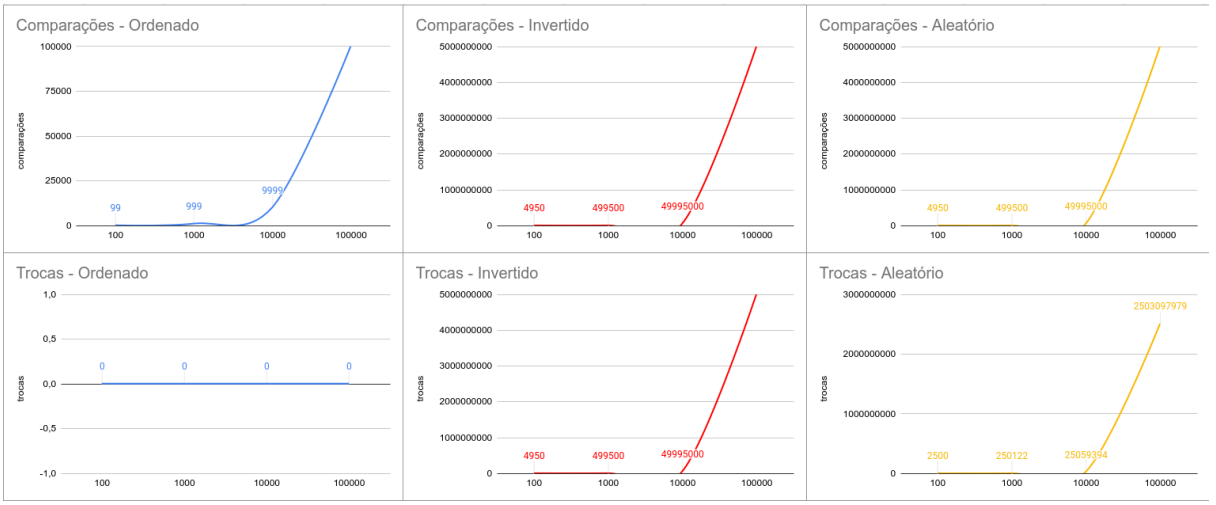
Na execução, cada algoritmo passa por um laço que percorre todos os arquivos de entrada, executando cada teste e coletando os dados de comparações, trocas e tempo para cada tipo de vetor. Os resultados são armazenados em um arquivo “.csv” referente a cada algoritmo de ordenação.

Com os dados organizados, os gráficos foram gerados usando as ferramentas do Google Planilhas e, analisando os resultados e as representações visuais, foi possível comparar a performance dos algoritmos estudados.

Algoritmos

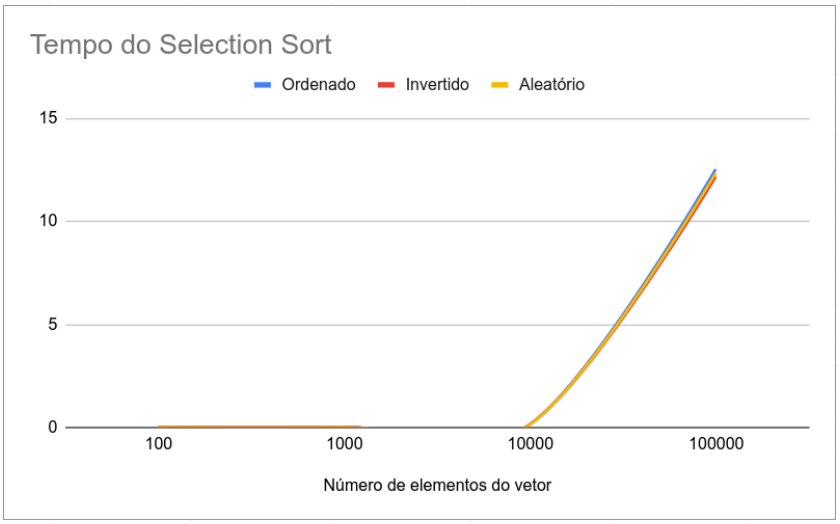
- **Bubble Sort**
  - Esse método compara dois elementos vizinhos e os troca de posição se estão na ordem incorreta.
  - Complexidade (pior caso): $O(n^2)$
  - Complexidade (melhor caso da implementação aprimorada): $O(n)$, o vetor está ordenado.

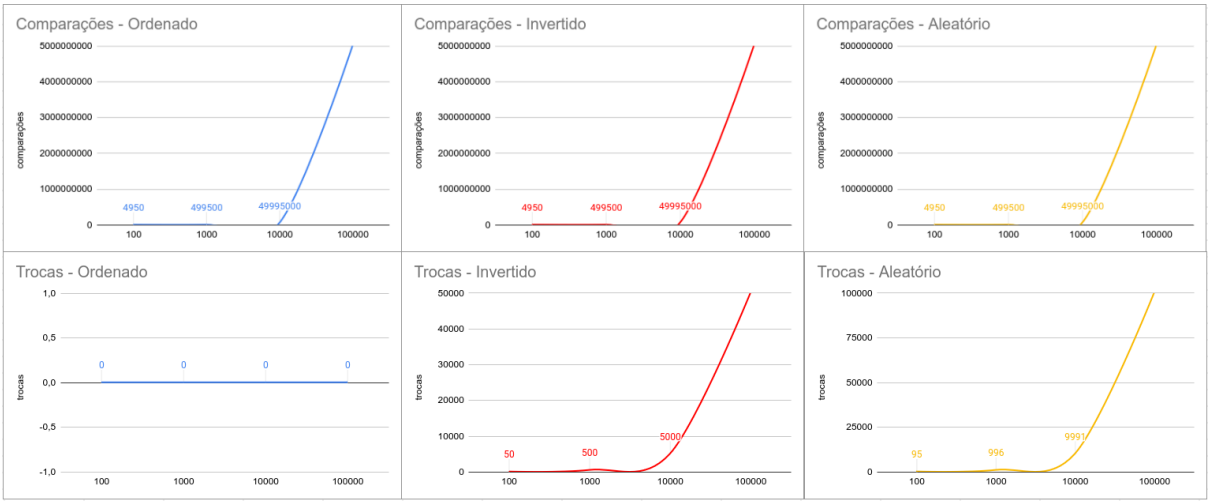




Quando o vetor está ordenado, não há nenhuma troca. Temos uma variável que mantém controle dessa quantidade, de forma que quando não acontece nenhuma troca o loop é encerrado, pois o algoritmo identifica que o vetor está ordenado. Nesse caso, a complexidade temporal é próxima de $O(n)$, como visto teoricamente. Nas outras duas situações é possível notar como o bubble sort é um algoritmo de ordenação pouco eficiente, esses tempos de execução foram os maiores entre todos os outros algoritmos. Note que isso é reflexo da quantidade elevada de comparações e troca de elementos que esse método realiza.

- **Selection Sort**
  - Esse método seleciona o menor elemento e o troca com o primeiro elemento do vetor naquela iteração.
  - Complexidade: $O(n^2)$

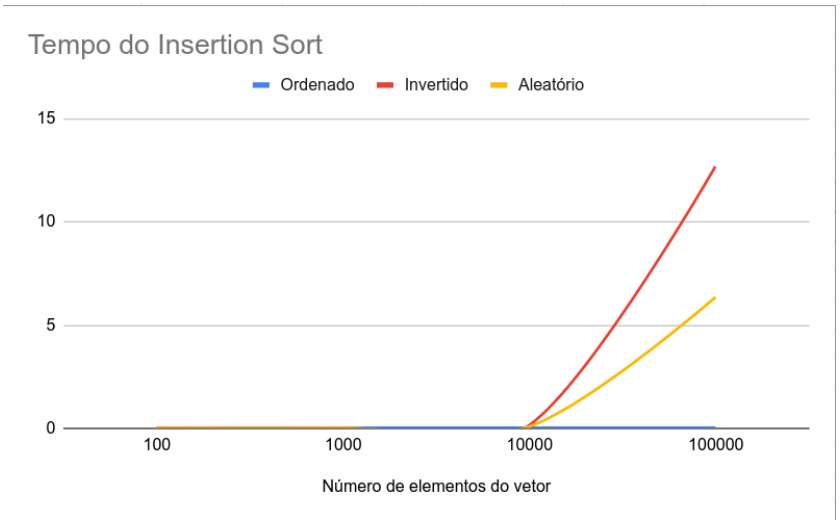


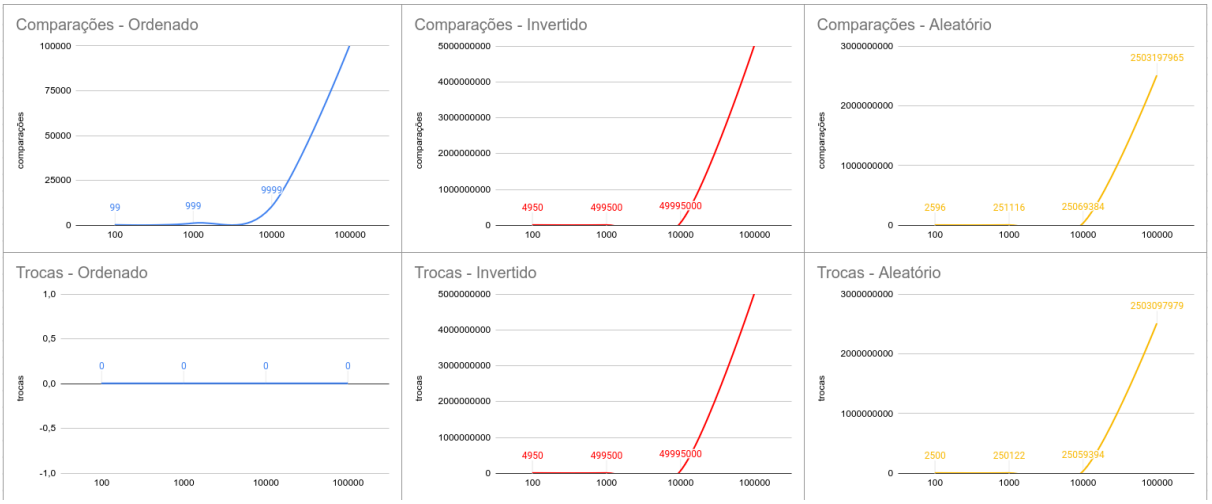


Para todos os vetores de mesmo tamanho, a quantidade de comparações é a mesma, pois sempre é necessário percorrer o vetor para encontrar o menor elemento. Dessa forma, os tempos de execução, como mostra o gráfico, são semelhantes, visto que independente da disposição inicial do vetor, é necessário percorrer todos os elementos para encontrar o menor. Por esse mesmo motivo, a quantidade de comparações dos três vetores também é a mesma.

Note que quando o vetor está inversamente ordenado, basta trocar metade dos elementos de suas posições que já teremos o restante do vetor em sua posição correta. No vetor aleatório, a quantidade de trocas é próxima da quantidade de elementos do vetor.

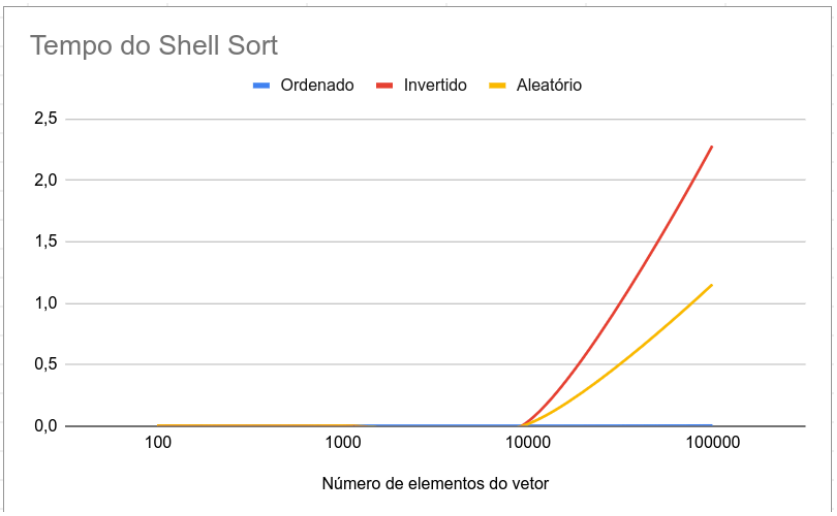
- **Insertion Sort**
  - Esse método insere o elemento em sua posição correta em um subconjunto já ordenado.
  - Complexidade: $O(n^2)$, pior caso é quando os elementos estão inversamente ordenados.

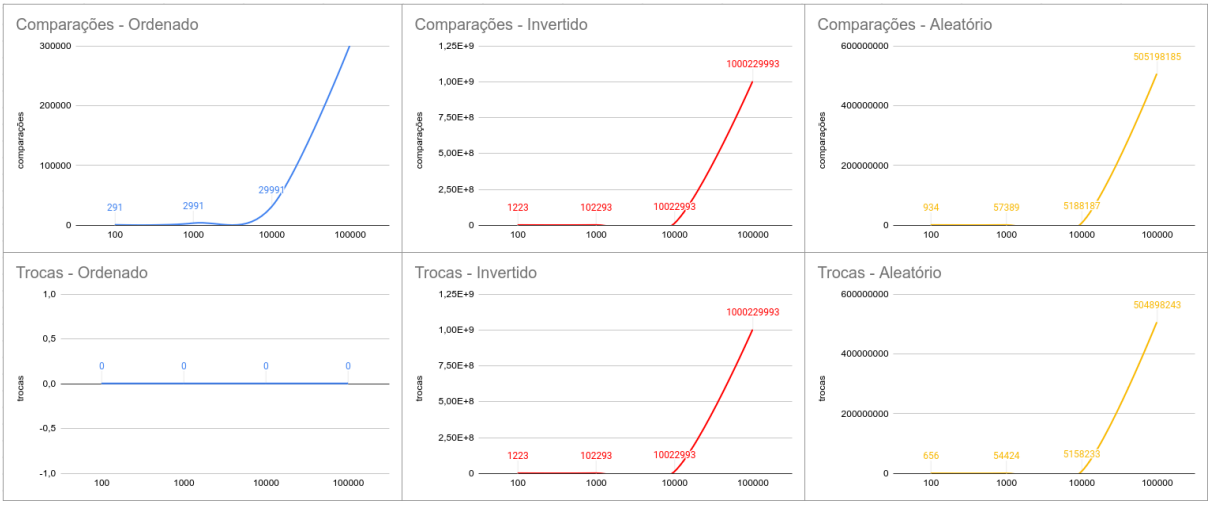




Quando o vetor está ordenado, o algoritmo verifica que o elemento está na posição correta, isso é feito em complexidade linear $O(n)$, muito pequeno em relação ao resto, por isso, é quase imperceptível no gráfico. O caso médio (elementos dispostos aleatoriamente) possui uma complexidade semelhante a teórica, o que era esperado. Além disso, como previsto, o maior tempo de execução é quando o vetor está inversamente ordenado, pois sempre será necessário mover o elemento para a primeira posição e deslocar todos os outros uma posição adiante. É possível notar esse fato na quantidade de trocas e comparações em vetores inversamente ordenados. As quantias são as mesmas, ou seja, cada comparação exigiu uma troca/deslocamento.

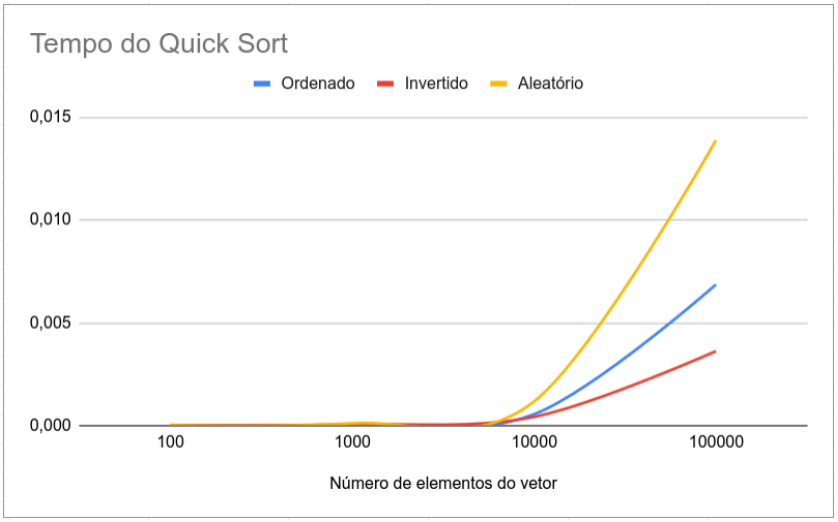
- **Shell Sort**
  - Esse método funciona como o insertion sort, mas ao invés de ordenar elementos vizinhos, o shell sort ordena elementos com um intervalo de distância.
  - Complexidade: $O(n\,log^{2}\,n)$ em geral, mas é dependente da sequência de incrementos.

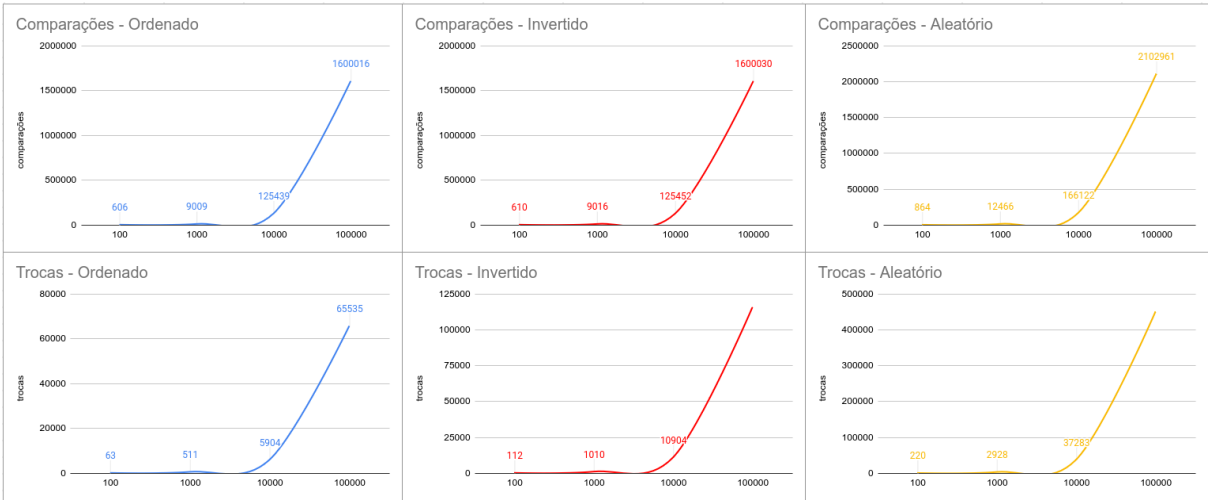




A eficiência do Shell Sort foi melhor que a do insertion sort, o que é esperado, pois as ordenações feitas em intervalos pré-determinados tornam o vetor mais ordenado, isso torna a inserção simples mais eficiente. O pior caso continua sendo quando o vetor está inversamente ordenado. Para o vetor ordenado inversamente, todas as comparações necessitam de trocas. No vetor aleatório, as quantidades são semelhantes.

- **Quick Sort**
  - Esse método utiliza um pivô, de forma que todos os elementos à sua direita são maiores que o pivô, e os à esquerda, menores. Chamamos a função para cada metade.
  - Complexidade: $O(nlogn)$, se o pivô divide o vetor na metade.
  - Complexidade: $O(n^2)$, se o pivô for o menor ou maior elemento ou vetor ordenado com pivô em um dos extremos

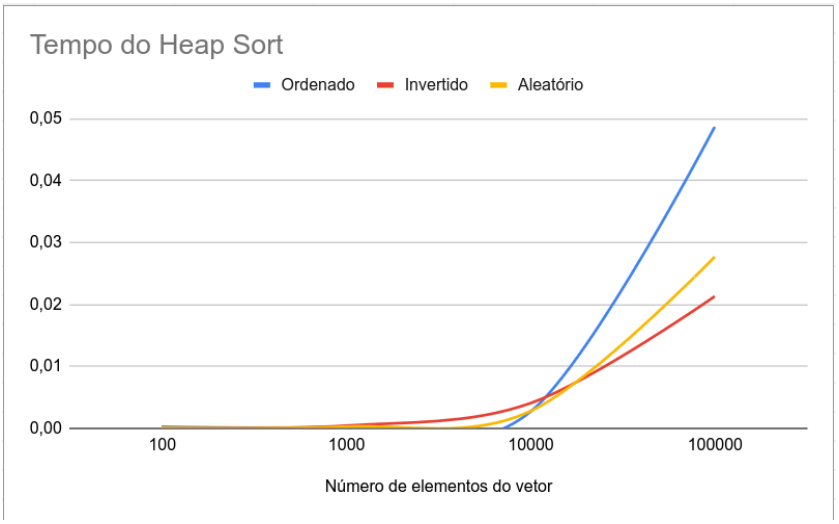


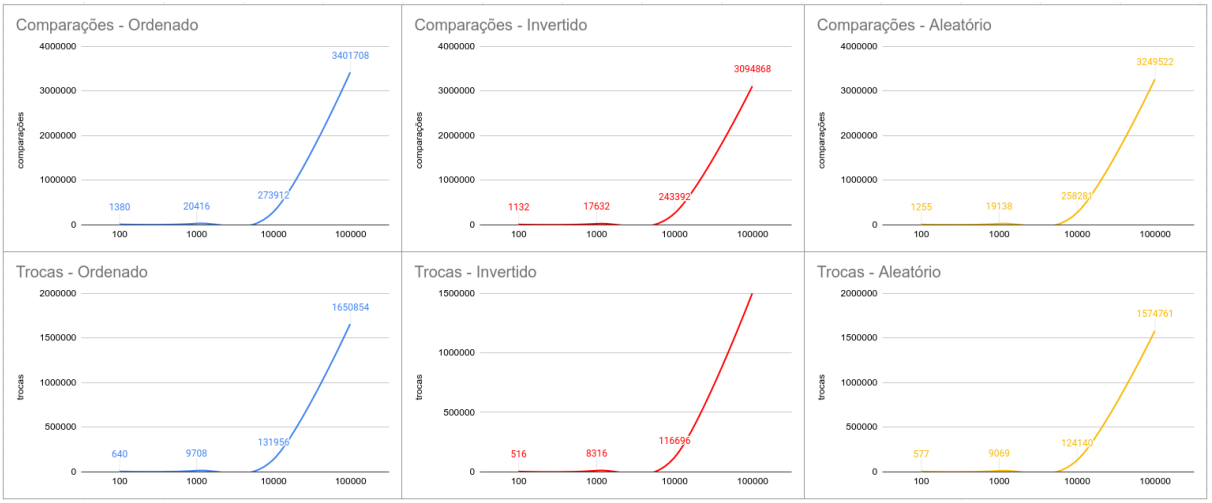


Os tempos de execução dos três tipos de vetores são semelhantes e próximos de $O(nlogn)$. Entretanto, quando o vetor é inversamente ordenado ou ordenado o tempo é um pouco menor, pois a escolha do pivô como a mediana entre os elementos do início, meio e fim, garante que o vetor será dividido exatamente na metade, garantindo a eficiência. De forma geral, a complexidade não chega perto do pior caso, devido a uma boa escolha de pivô.

As comparações contabilizadas são aquelas entre os elementos e o pivô, que garantem que os elementos à esquerda são menores e os à direita, maiores.

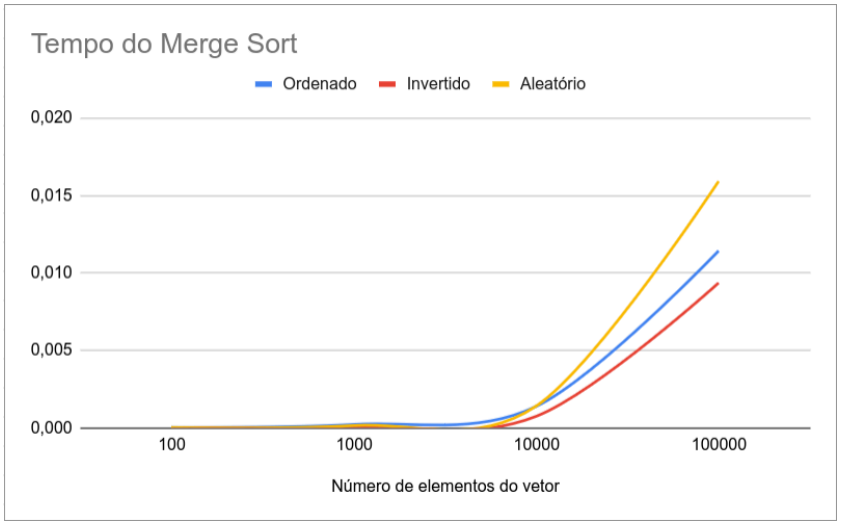
- **Heap Sort**
  - Esse método utiliza uma heap máxima para a ordenação. Troca-se o primeiro elemento (maior) com o último e rearranja-se a heap.
  - Complexidade: $O(nlogn)$

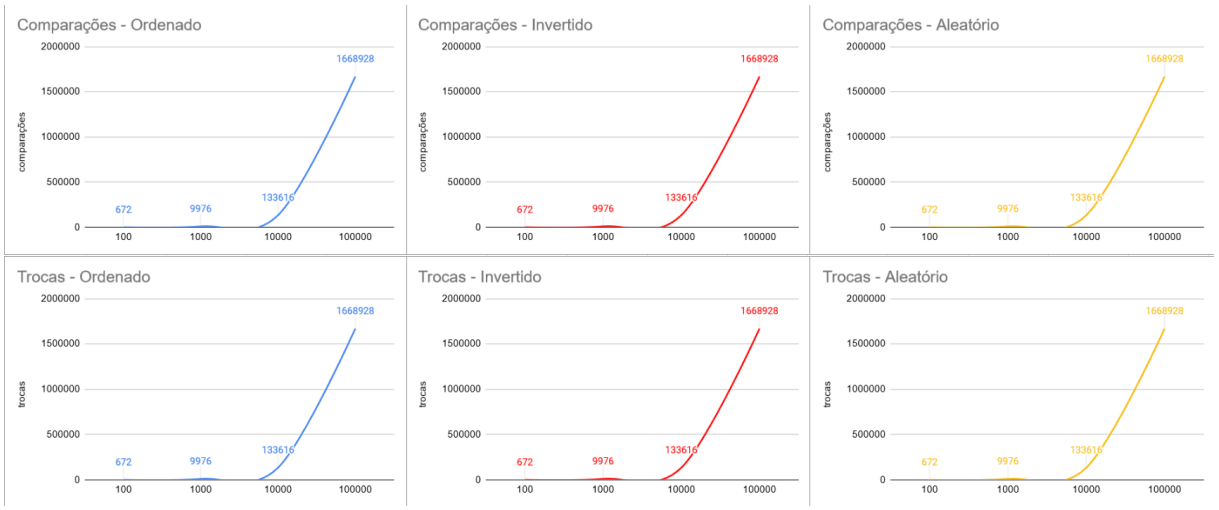




O maior tempo de execução foi do vetor ordenado, pois é o vetor que precisa da maior quantidade de trocas para rearranjar uma heap máxima. Os outros dois vetores tiveram tempos semelhantes. Mesmo quando o vetor já está ordenado, são realizadas diversas trocas, pois para a ordenação é necessário construir uma heap máxima e mantê-la.

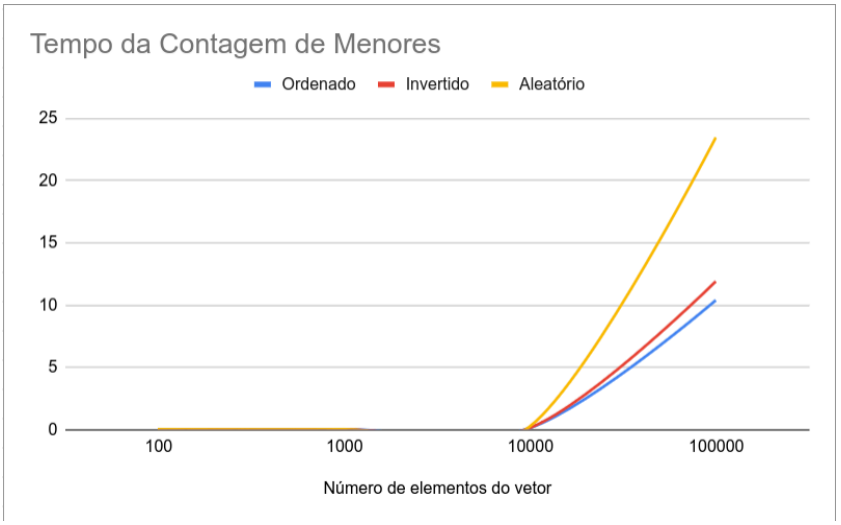
- **Merge Sort**
  - Esse método utiliza a técnica de divisão e conquista. O vetor é sucessivamente dividido em vetores menores, que depois são reunidos para formar o vetor ordenado.
  - Complexidade: $O(nlogn)$

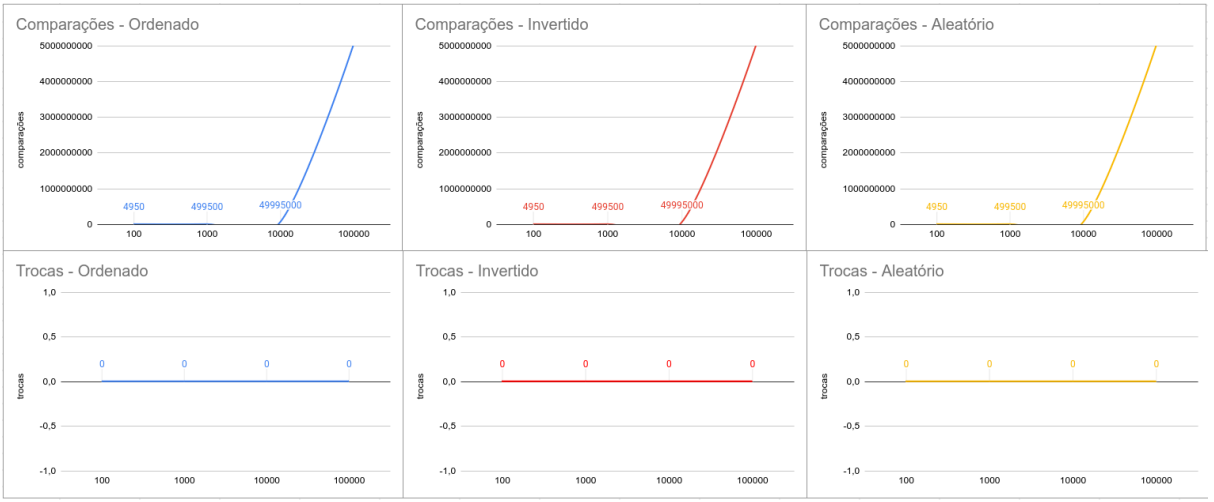




O tempo de execução dos três tipos de vetores foi bem semelhante e próximo da complexidade teórica, o que era esperado, já que o algoritmo do merge sort divide os vetores, recursivamente, na metade independentemente da sua disposição inicial. As quantidades de comparações são iguais para os três vetores, pois são dependentes da quantidade de registros, na etapa de *merge* em que os elementos são colocados em suas posições corretas. As movimentações dos registros dos vetores menores para o maior são contabilizadas como trocas, e para cada comparação entre os elementos dos dois sub-arranjos existe uma movimentação, o que explica os valores iguais de trocas e comparações.

- **Contagem de menores**
  - Esse método conta quantos elementos são menores que o registro sendo analisado. Um vetor auxiliar armazena essas quantidades e, assim, temos a posição correta que elemento deve estar no vetor ordenado.
  - Complexidade de tempo: $O(n^2)$
  - Complexidade de espaço: $O(3n)$



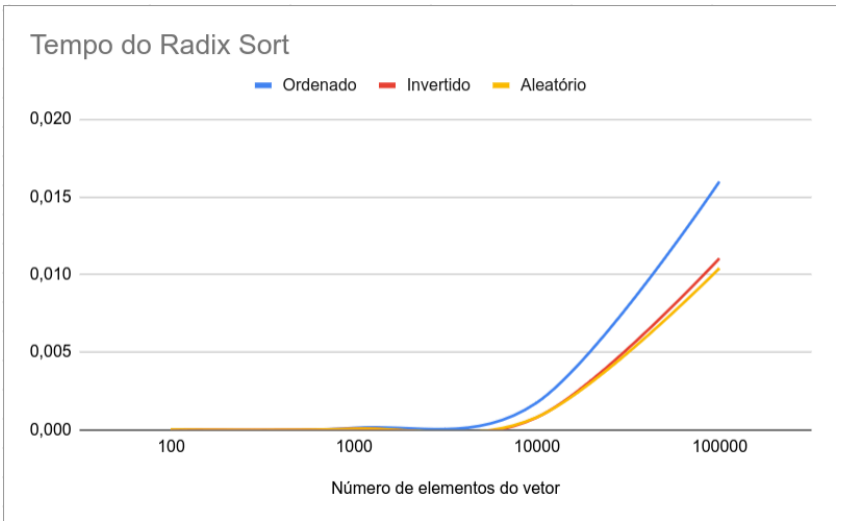


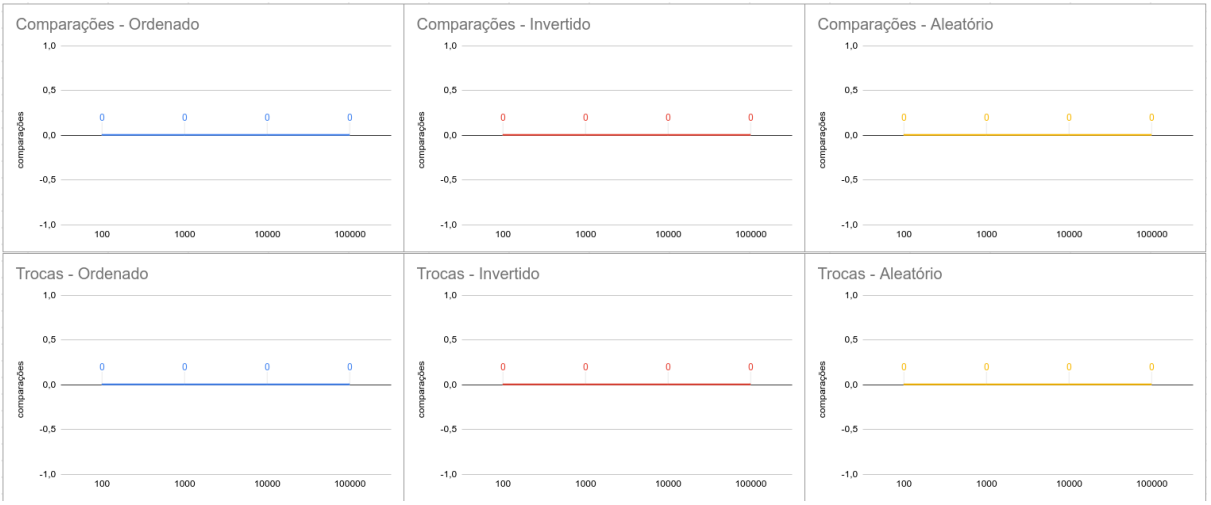
O algoritmo de contagem de menores foi um dos algoritmos com maior tempo de execução, o que era esperado considerando sua complexidade, a qual também independe da organização prévia do vetor.

O método de contagem de menores não realiza trocas, pois determina a posição correta do registro considerando a quantidade de elementos que são menores que ele. Para realizar a contagem de quantos registros são menores, realizamos comparações entre os elementos, a quantidade de comparações é a mesma para os três tipos de vetor.

Outra desvantagem é a necessidade de utilização de mais espaço de memória para um vetor que armazena a quantidade de menores e outro em que serão colocados os registros em suas posições corretas.

- **Radix sort**
  - Esse método ordena os elementos a partir dos dígitos, do menos significativo para o mais significativo.
  - Complexidade de tempo: $O(m * n)$, m é a quantidade de dígitos do maior número, ou seja, se m é pequeno, a complexidade é praticamente $O(n)$.





Os tempos de execução dos vetores são semelhantes, e o radix sort foi um dos algoritmos mais eficientes. Quando os vetores possuem números com poucos dígitos, a complexidade do algoritmo é, aproximadamente, $O(n)$ e, portanto, muito eficiente.

Não há comparações nem trocas no algoritmo radix sort, o que é esperado considerando que a ordenação dos algoritmos foi feita utilizando o método contagem de tipos, que conta quantos registros são de determinado tipo e assim determina a posição correta do dígito.

Conclusões

O algoritmo com o melhor desempenho é o Quick Sort. Todos os três tipos diferentes de vetores tiveram um bom desempenho considerando o tempo de execução, mas o melhor resultado foi obtido com o vetor inversamente ordenado. Uma das maiores desvantagens do Quick Sort é a escolha do pivô, que pode tornar a complexidade do algoritmo quadrática, entretanto, utilizando métodos adequados, como a escolha da mediana de três elementos, é possível contornar os piores casos (maior ou menor elemento ou o vetor ordenado com o pivô nas extremidades). Dessa maneira, garantimos que o algoritmo terá uma complexidade aproximada de $O(nlogn)$ na maioria dos casos. Além disso, é um algoritmo in-place, que não exige estruturas auxiliares de memória como a contagem de menores ou o merge sort, por exemplo.

O algoritmo com o pior desempenho foi o Bubble Sort. O tempo de execução foi o maior entre as implementações para quando o vetor é aleatório. No vetor inversamente ordenado, o tempo de execução é bem elevado também. No caso do vetor ordenado, o algoritmo é eficiente, pois em $O(n)$, consegue determinar que o vetor já está ordenado. Apesar disso, de modo geral, ele é o algoritmo de pior performance, pois possui um tempo de execução muito elevado, realiza muitas comparações entre os elementos e muitas trocas.