

**Análise de Algoritmo**

Projeto de Análise Prática de Complexidade de Algoritmos

Angelo Salvatti

Caio Viana de Jesus

Mateus Gabriel Mendes de Paula

Victor Giordano Damiani

**Sumário**

[Característica da Máquina de Teste 3](#_Toc199274276)

[Característica da Linguagem Testada 3](#_Toc199274277)

[Comparação dos Algoritmos O(n²) 4](#_Toc199274278)

[Bubble Sort: 4](#_Toc199274279)

[Insertion Sort: 4](#_Toc199274280)

[Selection Sort: 4](#_Toc199274281)

[Comparação dos Algoritmos O(n log n) 6](#_Toc199274282)

[Merge Sort: 6](#_Toc199274283)

[Quick Sort: 7](#_Toc199274284)

[Heap Sort: 7](#_Toc199274285)

[Comparação dos Algoritmos O(n + k) / O (nk) 10](#_Toc199274286)

[Counting Sort: 10](#_Toc199274287)

[Radix Sort: 10](#_Toc199274288)

[Implicações Práticas 14](#_Toc199274289)

[Considerações Finais 15](#_Toc199274290)

[Sugestões de Melhorias do Projeto 15](#_Toc199274291)

# Característica da Máquina de Teste

A máquina que será utilizada para teste de algoritmo com os valores, tem as seguintes propriedades:

Texto

O conteúdo gerado por IA pode estar incorreto.

O sistema operacional é o Windows 11 e a IDE será o Visual Studio Code, utilizando as extensões para compilação de C++. Além disso, foi configurado para 0 otimizações (O0) para que tire valores mais reais.

# Característica da Linguagem Testada

Uma característica importante da linguagem C++ é a alocação dinâmica de valores, já que para arrays simples o processo de alocação de memória pode falhar ao executar diversos valores, assim, virando um problema quando a complexidade do algoritmo é maior ou igual que O(n²), já que a alocação de espaço acaba por travar no meio da ordenação dos valores. Também pode ocorrer a mesma coisa quando o algoritmo é recursivo.

Portanto, foi necessário utilizar a alocação dinâmica (sem bibliotecas) que permite que execute com valores mais altos, mas aumento um pouco os valores para algoritmos dessa complexidade.

Ademais, a única biblioteca que foi utilizada para ordenação foi a *chrono*, que serve para pegar o tempo de execução de determinadas sequências de comando, onde deve-se demarcar o COMEÇO e o FIM da contagem, para retornar o valor de tempo de execução. Conforme exemplo abaixo:

auto start = high\_resolution\_clock::now(); // Onde começa

bubbleSort(arr, N);

auto end = high\_resolution\_clock::now(); // Onde termina

unsigned long long value = duration\_cast<nanoseconds>(end - start).count();

cout << "Nanossegundos: " << value << endl;

cout << "Segundos: " << value / 1e+9 << endl; // Conversão de nano para segundos

# Comparação dos Algoritmos O(n²)

## Bubble Sort:

O **Bubble Sort**, ou ordenação por flutuação, é um algoritmo de ordenação simples que funciona comparando elementos adjacentes e trocando-os se estiverem fora de ordem, movendo o maior elemento para o final da lista em cada passagem. Este processo é repetido até que a lista esteja completamente ordenada.

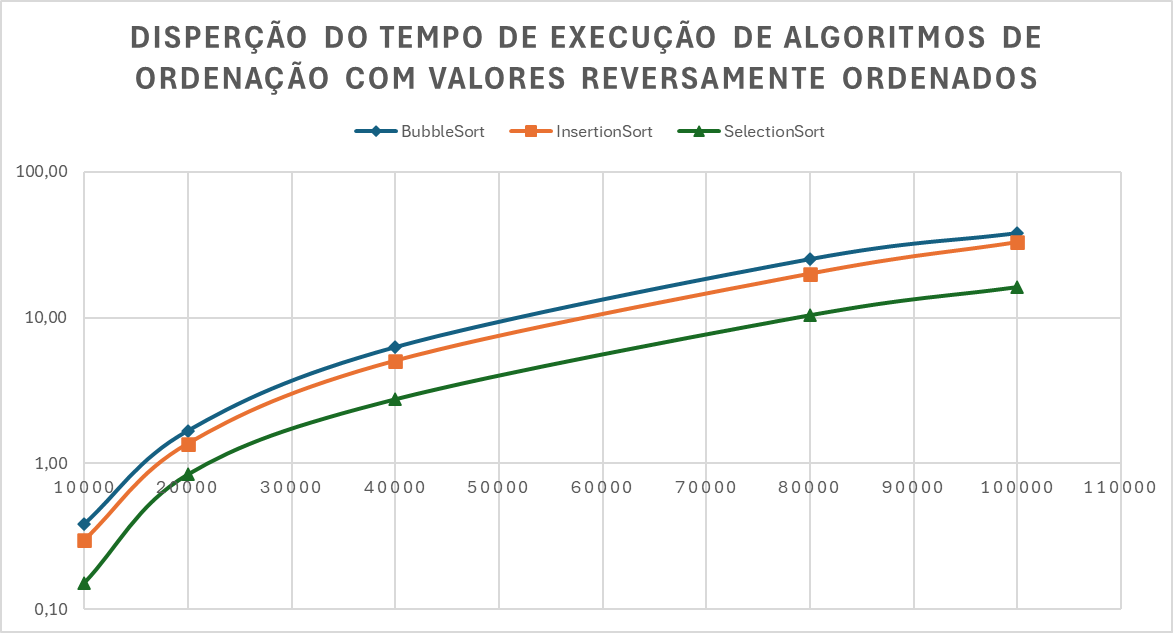
## Insertion Sort:

O **Insertion Sort**, ou ordenação por inserção, funciona de forma semelhante a ordenação de um baralho de cartas. O algoritmo itera sobre um array, considerando cada elemento como uma nova carta a ser inserida na posição correta dentro de um "baralho" já ordenado (a parte do array à esquerda do elemento atual).

## Selection Sort:

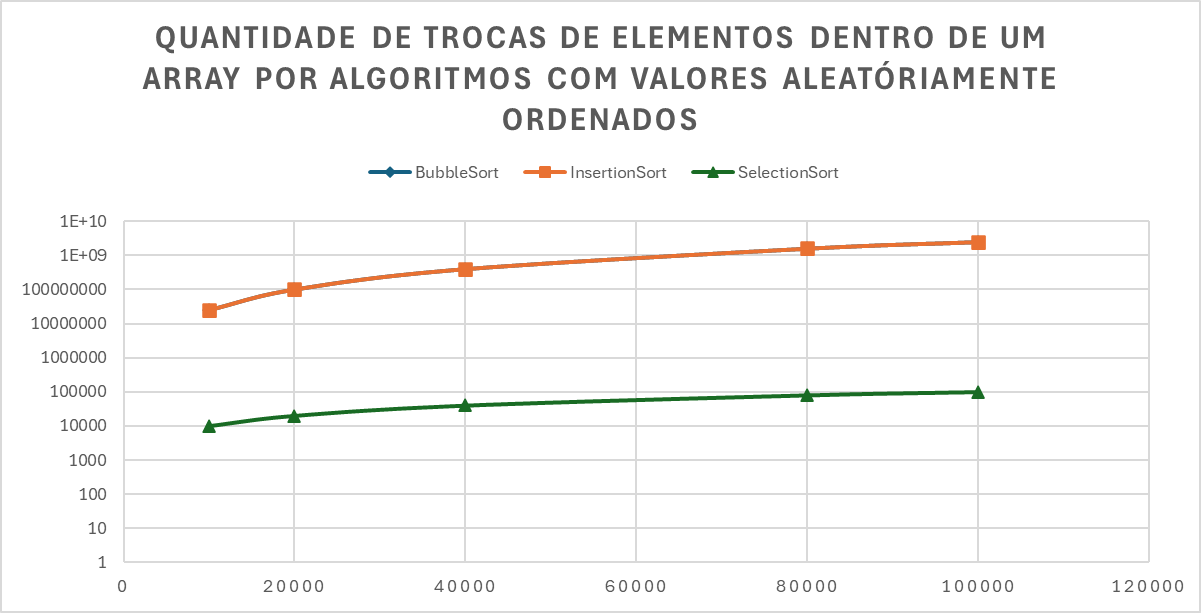
O **Selection Sort** é um algoritmo de ordenação simples que funciona selecionando o menor elemento de uma lista e movendo-o para o primeiro lugar. Este processo é repetido para os elementos restantes, resultando numa lista ordenada.

Durante testes para valores ordenados, todos apresentam um tempo quase tendendo a zero. Porém, quando olhamos quando eles estão inversamente ordenados conseguimos ver o pior desses casos:



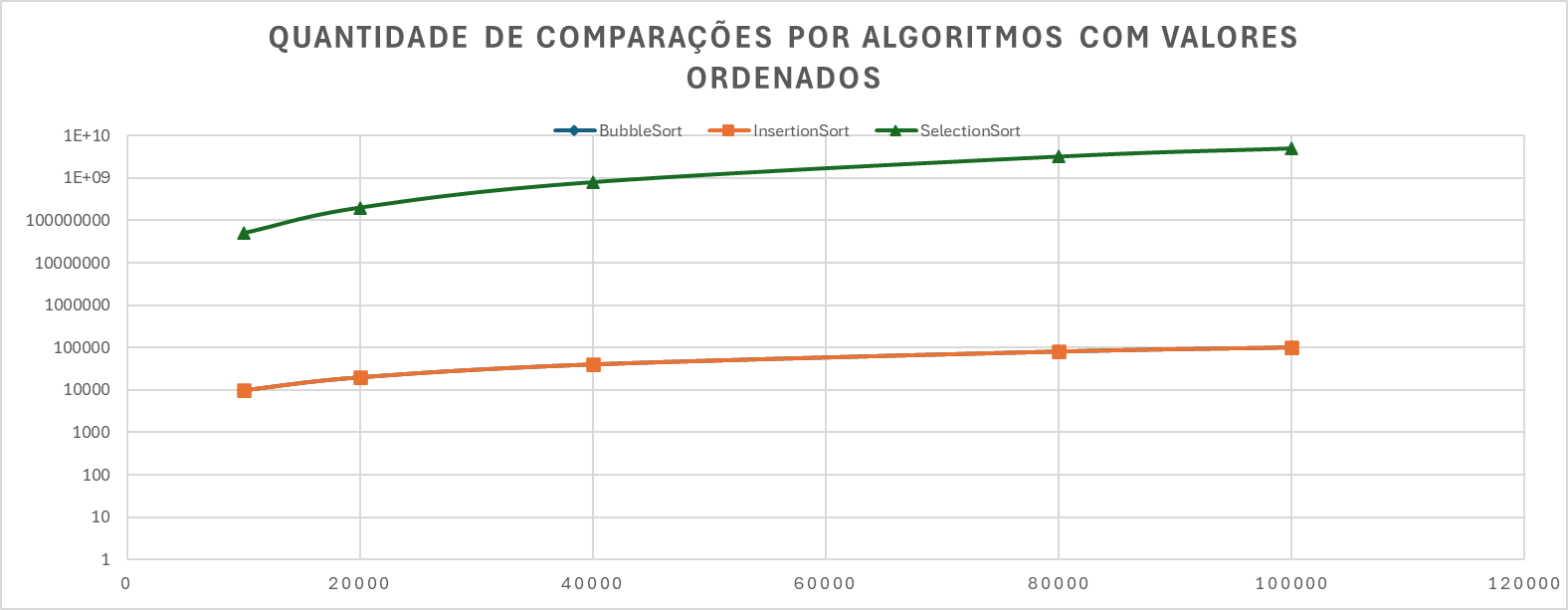
Onde o Bubble Sort lidera com o maior tempo de execução. Em contrapartida ao seu tempo de execução, todos eles têm uma complexidade espacial constante, de modo que armazenam apenas 3 (para o Insertion Sort) e 4 (para o Bubble e Selection Sort) words. O que faz deles uma opção viável caso seu armazenamento seja o problema em questão.

Outro fato interessante sobre os algoritmos O(n²) é quando quantos os valores estão ordenados nenhuma troca é feita, otimizando o tempo (motivo para o qual os valores ordenados de Insert e Bubble Sort tendem a zero), ao contrário do Selection Sort que tem uma constante de troca de valores. Porém, essa constância do “Selection Sort” é compensada de modo que quandos os valores estão Inversamente Ordenados ou Aleatoriamente Ordenados, já que ele tem substituições constantes, enquanto Bubble e Insertion Sort, acabam por fazer mais trocas e ordenações de costume. O gráfico abaixo ilustra a quantidade de trocas em casos aleatórios:



Note que o caso do Bubble Sort está abaixo da linha do Insertion, pois ambos tiveram valores semelhantes em quantidade de trocas, variando por poucos casos.

Além disso, na quantidade de comparações ordenadas, Selection lidera, mostrando que ele geralmente precisa fazer mais comparações quando os algoritmos estão ordenados, mas ambos têm números semelhantes quando aleatoriamente ordenados e inversamente ordenados.



Por fim, pode-se dizer que para casos de pouca memória, esses algoritmos são ideias, embora ele precise de um tempo superior para rodar e ordenar os valores dentro do array.

# Comparação dos Algoritmos O(n log n)

## Merge Sort:

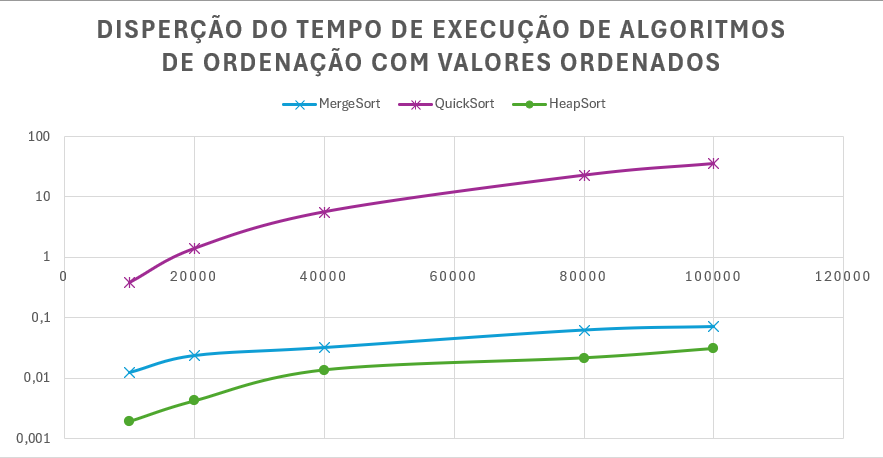
O **Merge Sort** é um algoritmo baseado na técnica dividir e conquistar. Neste caso, temos que ordenar uma lista de tamanho n. Dividir: Dividimos a lista de tamanho n em duas sub-listas de tamanho aproximadamente iguais (de tamanho n/2). Resolvemos o problema de ordenação de forma interativa para estas duas sub-listas.

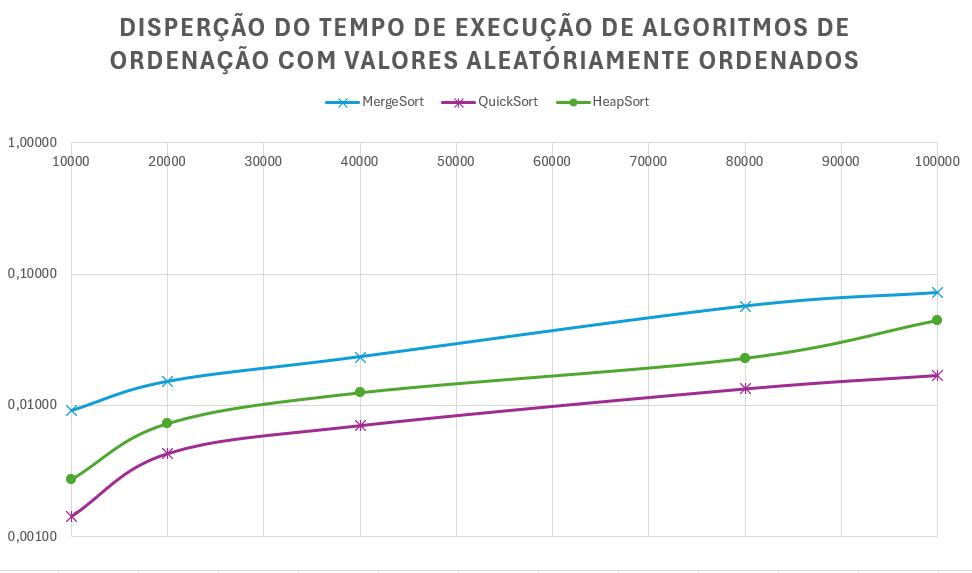
## Quick Sort:

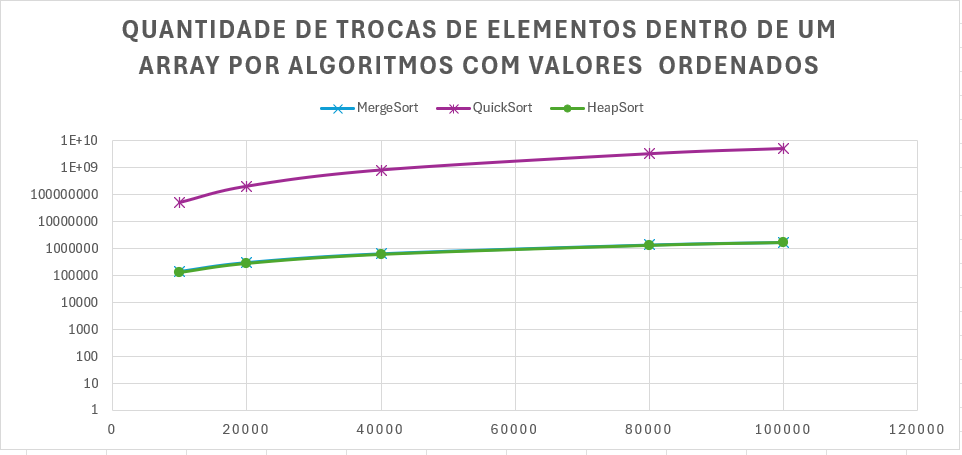
O **Quicksort** é um algoritmo de ordenação que utiliza a estratégia de "dividir para conquistar" para ordenar uma matriz ou array. Ele escolhe um elemento "pivô" e separa os outros elementos em dois subarrays, com valores menores à esquerda e maiores à direita do pivô. Essa etapa é repetida interativamente para os sub-arrays até que a matriz esteja totalmente ordenada.

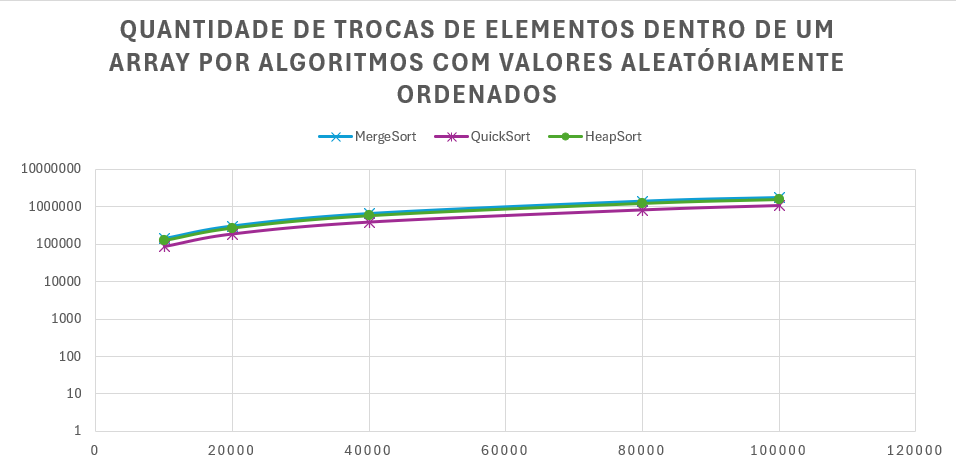
## Heap Sort:

O **Heap Sort** é um algoritmo de ordenação que utiliza uma estrutura de dados chamada heap (árvore binária) para classificar elementos. O algoritmo funciona construindo um heap a partir do array de entrada e, em seguida, removendo os elementos do heap na ordem correta, adicionando-os de volta ao array.

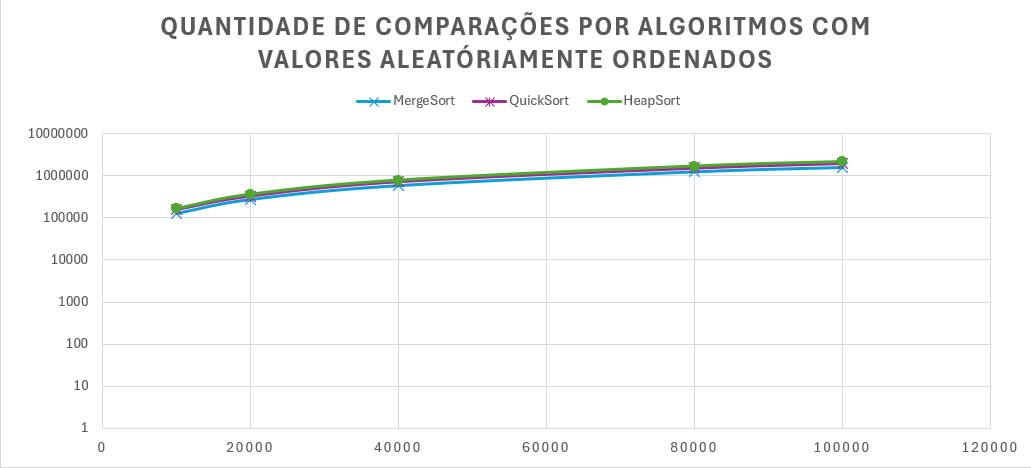
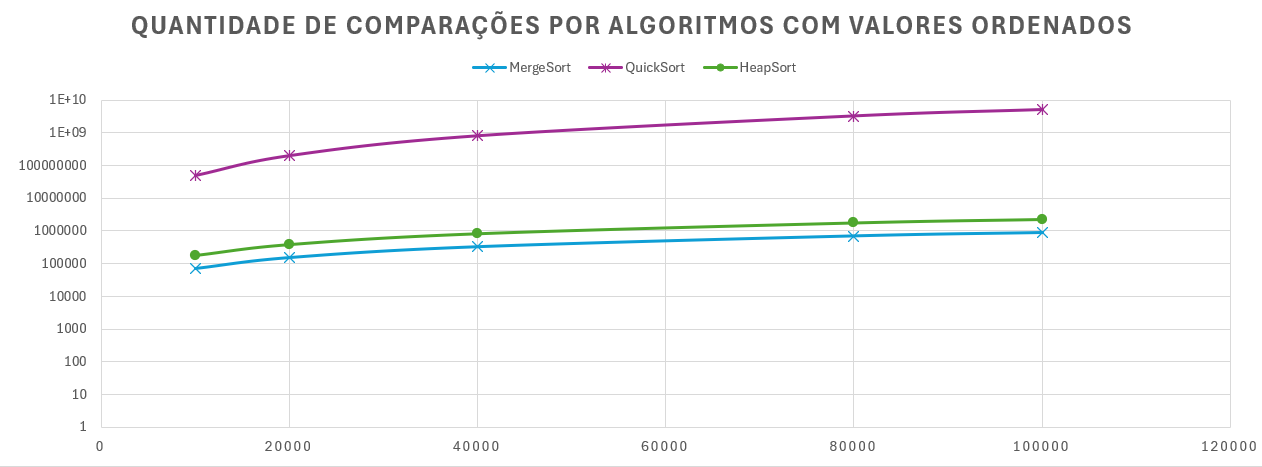


  
 Através dessa comparação é perceptível que o método quick sort é o melhor método quando se usa valores de forma aleatória, porém ao utilizar valores que estão em ordem seu rendimento é o pior entre eles.

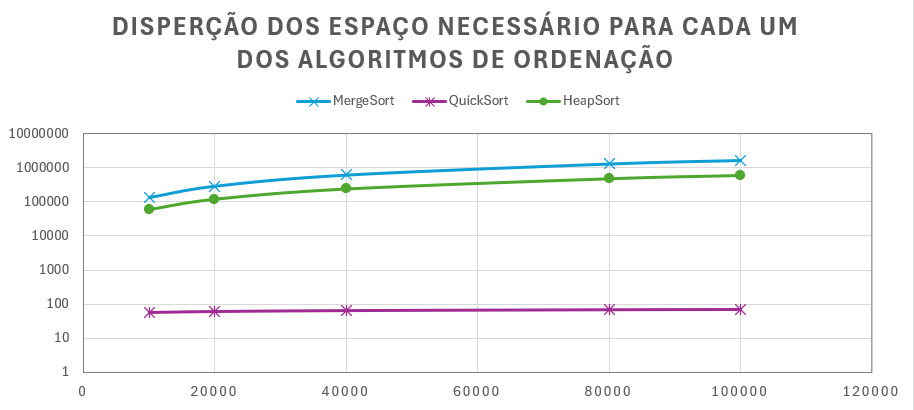




Agora analisando a quantidade de trocas realizadas entre esses 3 métodos percebe-se que em valores ordenados o Quicksort ainda continuou o pior, porém em valores aleatórios eles tiveram uma performance semelhante, sendo o quick sort por último.



Observando agora pela quantidade de comparações realizadas com valores ordenados o quick sort faz a maior quantidade de comparações, agora quando são valores aleatórios todos fazem praticamente a mesma quantidade de comparações.

  
 Ao analisar a comparação espacial é possível perceber que o algoritmo Quick Sort é o que ocupa a menor quantidade de memória do computador, enquanto Heap e merge possuem alocamento de memória semelhantes.

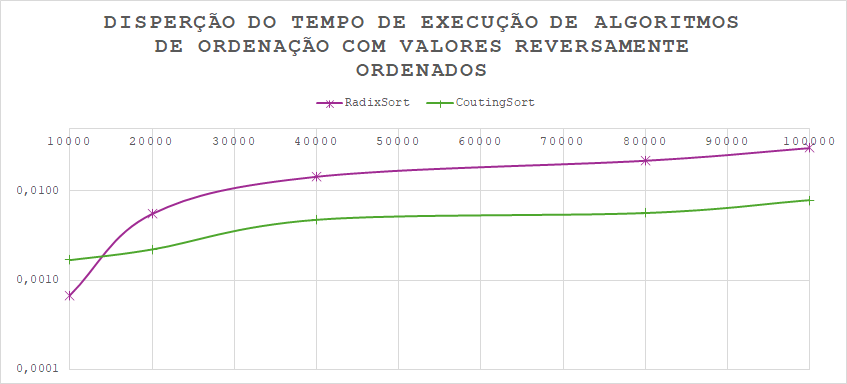
# Comparação dos Algoritmos O (n + k) / O (nk)

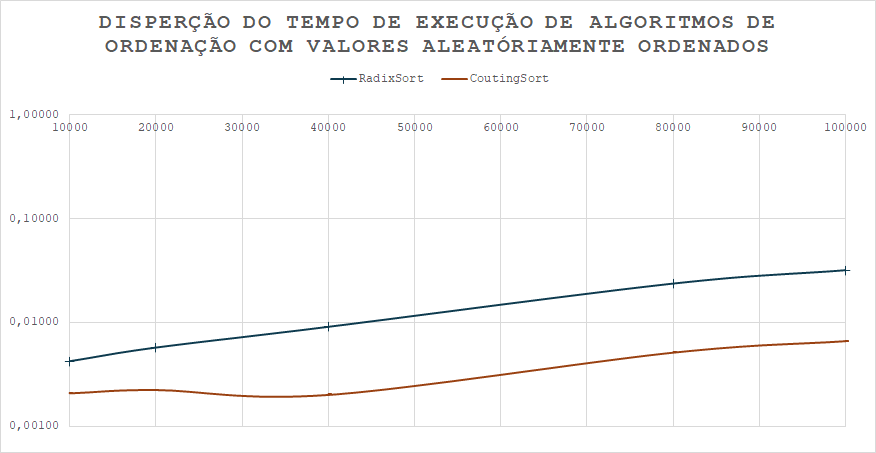
## Counting Sort:

O **Counting Sort** é um algoritmo de ordenação não comparativo, que funciona contando quantas vezes cada elemento aparece e usando essa contagem para posicionar os elementos diretamente no array final.

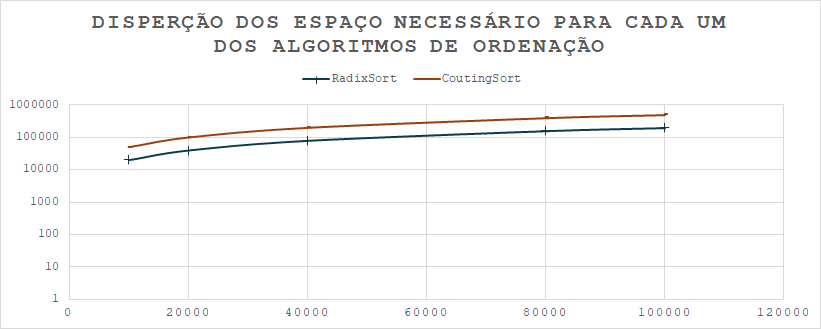
## Radix Sort:

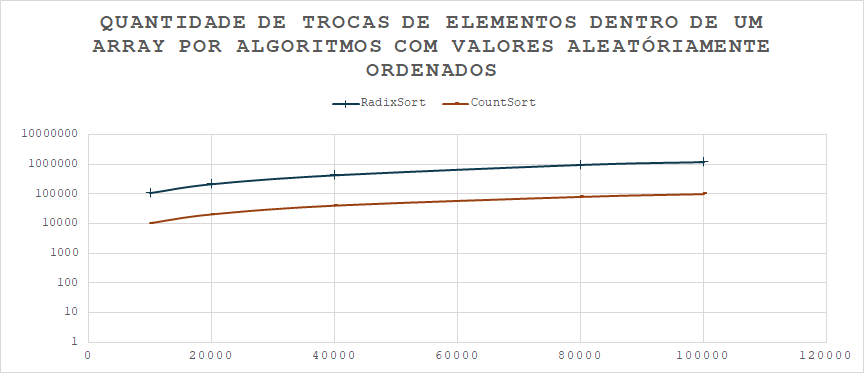
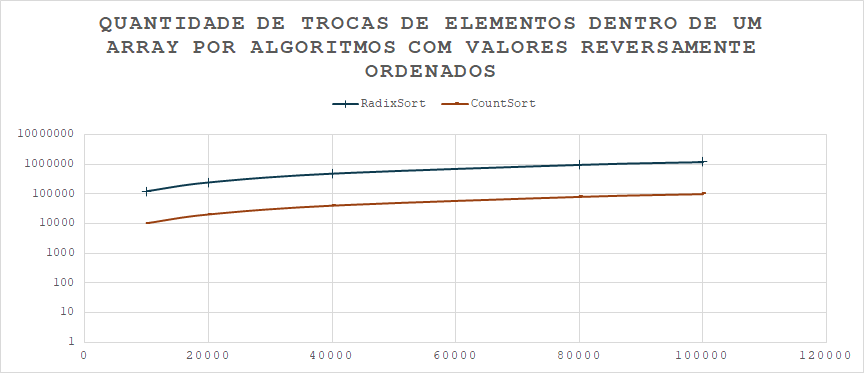
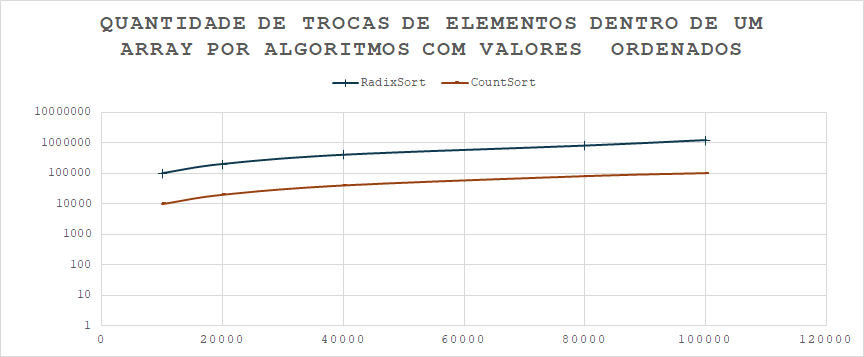
O **Radix Sort** é um algoritmo de ordenação que organiza números dígito a dígito (ou caractere a caractere), geralmente começando do menos significativo (LSD) ou mais significativo (MSD). Ele usa o Counting Sort como subrotina para ordenar por cada dígito. Gráfico

O conteúdo gerado por IA pode estar incorreto.

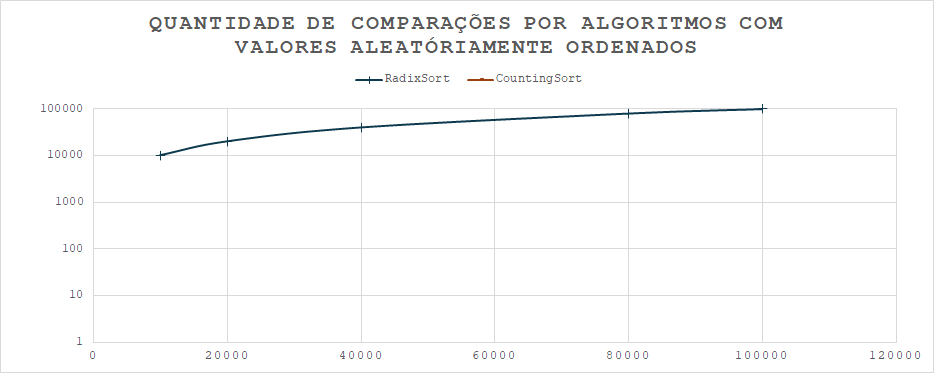
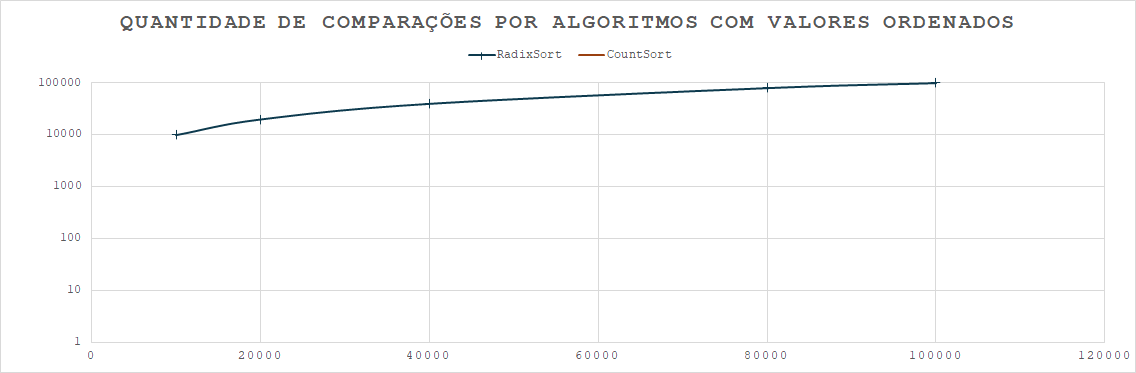
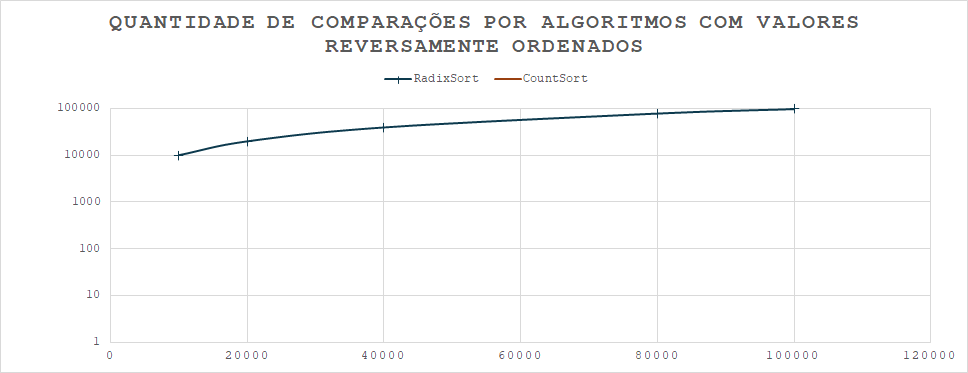


Analisando o tempo de execução dos métodos para as três situações: Com valores já ordenados, reversamente ordenado e ordenados aleatoriamente é perceptível que o tempo de execução do Radix cresceu bem mais que o do Count. Porém, não é indicativo final que o Count seja melhor que o Radix, para isso deve-se analisar outros dados coletados.

Agora analisando a quantidade de espaço que é necessário à medida que a entrada de dados aumenta, percebe-se que o Radix cresceu bem menos que o Count, no qual o Count precisa de mais que o dobro de espaço do Radix para realizar a ordenação.

A quantidade de trocas nas 3 situações propostas indica que o Radix realiza bem mais trocas que o Count, devido aos rearranjos que realiza, tendo que realizar novamente algumas comparações caso algum elemento não esteja ordenado, no fim, sendo necessário mais trocas.

Por fim, a partir dos gráficos de Comparações, pode-se concluir que o Count não compara elementos entre si para decidir quem vem antes, ele simplesmente conta posições com base em índices diretos (acessos ao array). Por isso que no gráfico ele está em 0.



# Implicações Práticas

Considerando a organização de dados de usuários em um contexto empresarial, a escolha do melhor algoritmo de ordenação depende da chave de ordenação.

Se o objetivo é ordenar os usuários por salário, o Quick Sort geralmente se destaca como a opção mais eficiente. Sua performance média é superior para grandes conjuntos de dados, tornando-o ideal para essa finalidade.

No entanto, se a ordenação for feita por ID do usuário, outros métodos podem ser mais vantajosos. Isso ocorre porque os IDs geralmente são sequenciais ou têm uma distribuição mais uniforme, permitindo que algoritmos como Merge Sort ou até mesmo Counting Sort (se os IDs tiverem um intervalo limitado) apresentem um desempenho melhor ou mais consistente, especialmente em cenários específicos.

# Considerações Finais

Pelo que foi analisado por meio dos gráficos, utilizando dados ordenados funções como bubble sort e insertion sort foram muito eficientes pois primeiro verificam o vetor para realizar as trocas, já para os dados invertidos a função counting sort teve melhor desempenho, devido sua baixa taxa de comparações e trocas, além de sua complexidade, algo que também foi visto para os dados aleatórios, com o counting se saindo melhor.

Radix Sort quebra números grandes em dígitos menores, e usa Counting Sort em cada dígito permitindo-o lidar com números grandes sem precisar um array gigante de contagem. Fazendo com que ele resolva problemas que Counting Sort puro não consegue, por causa do tamanho do range. Concluindo, se é possível usarCounting Sort puro, ele é mais rápido e simples, mas quando os números têm um range muito grande, Radix Sort é a solução prática porque ele quebra o problema em partes menores.

Para mais de 1 milhão de elementos, o Radix Sort é uma excelente escolha para ordenar números inteiros (IDs ou salários), pois sua complexidade de tempo é linear em relação ao número de elementos e ao número de dígitos do maior elemento, e ele supera as limitações de memória e alcance do Counting Sort para grandes intervalos de valores.

# Sugestões de Melhorias do Projeto

**Máquina isolada:** isolamento da máquina para evitar execuções paralelas, já que a máquina pode apresentar variações de tarefas executando em segundo plano, consumindo bastante tempo do projeto para compilação dos dados, já que teve que ter uma análise de cerca de 10 (dez) resultados obtidos para cada caso. Isso eliminaria inconsistência dos dados.

**Outros tipos de dados:** testar os mesmos algoritmos, mas com tipos de dados diferentes para ver a efetividade comparada, embora os resultados sejam semelhantes, eles poderiam refletir muitas possibilidades de execução

# Link para o GitHub

<https://github.com/wavering-ruby/SortingAlgorithms-TimeComplexity>