**UNIVERSIDADE FEDERAL DA FRONTEIRA SUL**

**CAMPUS Chapecó**

**CURSO DE Ciência da Computação**

**relatório sobre comportamento de métodos de ordenação de vetores:**

Trabalho TP2 - pod

**Jonathan Gotz Correa**

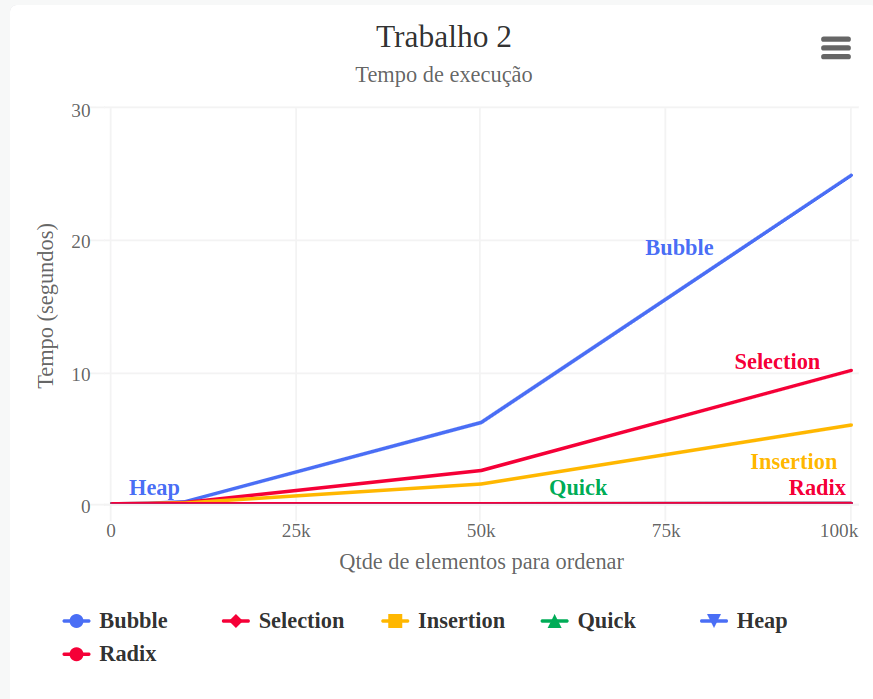
**Chapecó**

**2023**

Foi realizado uma pesquisa em aula sobre os diferentes métodos de ordenação de vetores/arrays utilizando da linguagem compilada C, destacando suas vantagens, desvantagens e lógicas de funcionamento, sendo esses métodos: Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Heap Sort, Counting Sort e por fim o Radix Sort.

Após criar e desenvolver em um arquivo tipo .c todos os métodos separadamente adicionando contadores de tempo de execução e também contadores de trocas realizadas para até finalizar a ordenação, que se encontra em meu [Github atualmente](https://github.com/Jonagotz/POD_sorts_and_studies), foi possível gerar os seguintes gráficos:

Gráfico de tempo de execução



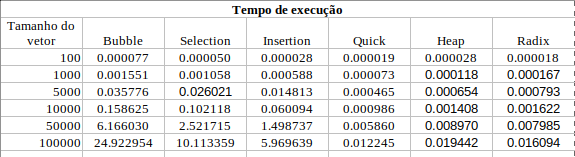
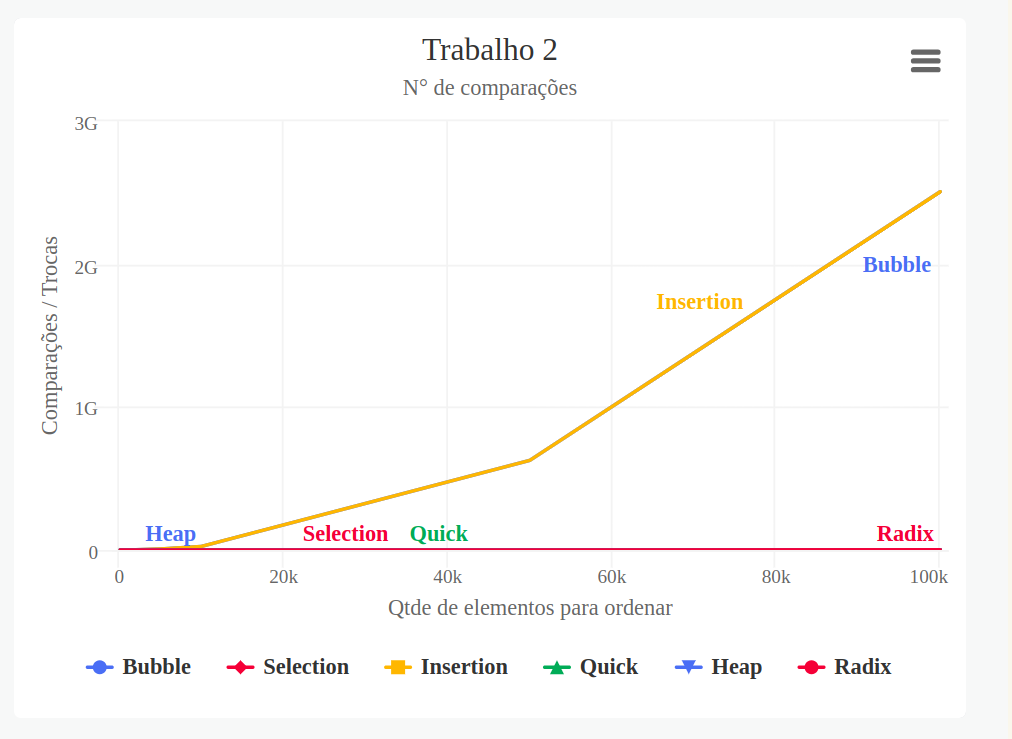
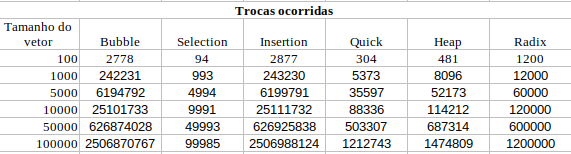


Gráfico de comparações





Baseado nos gráficos e nas planilhas usadas para produzi-los podemos perceber que em termos de eficiência o método de ordenação de vetores mais demorado e com mais comparações é o Bubble Sort, visto que ele é um algoritmo simples e somente útil para pequenos conjuntos de vetores, não sendo bom para conjunto de dados muito grandes, por ter uma complexidade de tempo quadrática onde o tempo médio com ou sem flag é sempre, além de comparar sempre o elemento atual no loop com o elemento adjacente, levando então a ser considerado o menos recomendável para uso por ter uma quantidade extremamente alta de comparações e trocas.

Em seguida o método Selecition Sort é o próximo no gráfico a ter uma quantia de tempo mais alta para realizar a ordenação dos arrays atribuidos a ele por mais que tenha uma quantidade de comparações perceptivelmente menor, ele possui funcionalidades recursivas melhores do que o Bubble, sendo elas primeiramente a procura pelo menor elemento não ordenado do vetor e em seguida adicionar na primeira posição e nas subsequentes do vetor, então, por mais que ambos possuam complexidade de tempo constante quadrática para serem executados, o Selection ao invés de comparar o elemento atual no loop com o elemento adjacente ele basicamente pega o menor elemento do array e o coloca na posição correta, o que gera menor trabalho de execução e menos comparações.

O Insertion se encontra com maior prestígio que o selection em seguida, tendo um desempenho melhor no tempo de execução, este sendo considerado mais eficiente quando o array já está parcialmente ordenado já que só será realizada alguma alteração no array enquanto ele for percorrido da esquerda para a direita e compara-o com os elementos à sua esquerda até encontrar o local correto para inseri-lo, e à medida que encontra o local correto, desloca os elementos maiores para a direita para abrir espaço para o elemento atual, aumentando a quantidade de trocas realizadas para a geração do segundo gráfico, enquanto o Selection realiza a mesma quantidade de comparações independentemente do estado inicial do array, mesmo que com menos trocas, de outras formas eles são consideravelmente bem parecidos, já que ambos buscam um menor valor e o colocam em seguida na posição correta.

A partir disso é mais complicado de decifrar qual é muito superior aos demais olhando somente os resultados do gráfico em relação ao desempenho de tempo de execução, precisamos verificar então os piores cenários e como são executados os decorrentes métodos, assim, podemos levar em consideração o Quick Sort em seguida, ele é considerado rápido e eficiente já que em casos médios sua complexidade de tempo constante é logarítimica ()) em que o pivô se divide em duas partes exatamente iguais ou quase, e isso é precisamente um ótimo tempo.

Dentre os processos que ocorrem dentro do método podemos explicar a quantidade de trocas existentes que levam a existênca do segundo gráfico, primeiramente o Quick Sort recursivamente realiza a escolhe do pivô, o qual seria um elemento do array a escolha do desenvolvedor, sendo o mais recomendado este ser um valor aleatório, em seguida é realizada a partição, onde o array é particionado em duas partes - uma contendo elementos menores que o pivô e outra contendo elementos maiores que o pivô e assim recursivamente aplicando partições para as partições subsequentes. Portanto, a razão pela qual o Quick Sort pode realizar um número significativo de comparações e trocas em alguns casos está relacionada à escolha do pivô e à distribuição dos dados.

Porém para o Quick, no pior caso, que é quando é passado como pivô o menor valor do array este terá complexidade quadrática tal qual Insertion Sort (), para remediar tal problema basta passar o pivô randômico porém sempre terá a possibilidade de tal ocorrência mesmo sendo considerado um dos ordenamentos mais eficientes.

O Heap Sort porém não possuí tal complicação já que tem complexidade de tempo constante no pior, melhor e no médio caso além de não ser tão afetado por conjuntos de dados pré-ordenados ou quase ordenados quanto alguns outros algoritmos. O Quick Sort normalmente se sai melhor na maioria dos testes de desempenho mesmo se for implementado corretamente utilizando grande conjuntos de dados, porém, em casos específicos, como quando a estabilidade é crucial ou quando os requisitos de espaço são restritos, o Heap Sort pode ser preferível.

O Heap Sort realiza comparações e trocas devido à sua natureza de construir e manter uma estrutura de heap, que é uma árvore binária completa ou quase completa. Possui alguns passos que explicam o motivo de ter uma certa quantidade maior de comparações e trocas comparado com o Quick Sort por exemplo, mas ainda garantindo desempenho de no pior caso:

Primeiramente a construção do Heap (Max Heap), que seria considerado uma árvore binária completa) a partir do array, o qual envolve percorrer o array da esquerda para a direita, convertendo-o em árvore binária, ao qual são realizadas trocas entre os elementos do array e seus pais para garantir que o Max Heap seja obedecido.

Em seguida a extração do máximo e reconstrução do Heap, ao qual, o maior elemento (na raiz) é trocado com o último elemento do array e o último elemento (que agora é o maior) é removido temporariamente do heap para então o heap ser reconstruído com o objetivo de manter a propriedade Max Heap, posteriormente, esses passos são repetidos até que todo o array esteja ordenado.

Por fim podemos detalhar sobre a eficiência do Radix Sort, o método estável do qual possui uma complexidade de tempo constante , particularmente muito eficiente para casos em que o array para ordenação contiver inteiros e o padrão dos dígitos for importante, sendo neste caso mais eficiente que o Heap Sort, porém somente para estes casos, visto que se o array contiver tipos de dados mais complexos ou se não houver um padrão claro nos dígitos, o Heap Sort pode ser uma escolha mais versátil.

Este método de ordenação utiliza da ideia de verificar e ordenar os valores pela quantidade de dígitos, definindo quantas vezes deverá ser passado recursivamente pelo método até ser finalizada a operação, não realizando comparações entre elementos, diferente dos demais algoritmos, ele explora a estrutura de dados. Sua principal operação é a distribuição dos elementos em baldes ou filas com base nos dígitos ou caracteres em posições específicas. Em seguida, os elementos são coletados de volta na ordem dos baldes. Este processo é repetido para cada posição do dígito ou caractere até que todo o array esteja ordenado.

Por exemplo, vamos pegar um vetor que tenha noventa e nove (99) números inteiros de somente dois (2) dígitos e um (1) número inteiro de três (3) dígitos; por causa deste valor de três dígitos o método será executado mais uma vez recursivamente, por isso é fortemente indicado para casos específicos em que o padrão dos dígitos for importante.