**Trabalho 1 – Classificação e Pesquisa de Dados**

**Dupla:** Rhuan Lucas Barbosa Fernandes e Davi Santos Ferrarez

Link para o github: <https://github.com/RhuanLucass/cpd-trabalho01>

1. **Introdução:**

**Bubble sort:**

Sua complexidade é O(n²), por este motivo o bubble sort não é recomendado para uma entrada muito grande de dados.

Este algoritmo de ordenação consiste em percorrer todo o vetor de elementos comparando o valor da posição atual com seu próximo, caso a condição seja verdadeira, os dois são trocados de posição. O vetor é percorrido em loop até que todos os elementos estejam ordenados da forma correta.

**Selection sort:**

Não é considerado um algoritmo estável. A complexidade deste algoritmo de ordenação é O(n²) no melhor e no pior caso. A busca pelo menor elemento custa n-1 passos na primeira ordenação, n-2 passos na segunda ordenação, n-3 passos na terceira, e assim por diante. O Selection sort funciona de forma bem simples e direta: Consiste em encontrar o menor elemento de um array e colocá-lo na primeira posição. Para ordenar o vetor, deve-se aplicar essa rotina repetidas vezes para o restante do array.

**Insertion sort:**

O insertion sort possui complexidade O(n²) e é o mais eficiente entre os algoritmos para solução de pequenas entradas.

Sua ordenação é realizada de forma que o elemento atual seja comparado com seus antecessores, um por vez, até que a condição não seja mais satisfeita, e após isto seja posicionado no seu devido local. Para ordenação crescente ele será posicionado a frente do primeiro elemento menor que ele, já para decrescente, será posicionado a frente do primeiro maior.

**Quicksort:**

O Quicksort é um algoritmo eficiente de ordenação por divisão e conquista, porém não é estável. Sua complexidade no melhor caso e no caso médio é O(n log n) e, no pior caso, O(n²). O funcionamento do Quicksort baseia-se em uma rotina considerada fundamental chamada **particionamento**. Esta rotina significa a escolha de um número qualquer presente no array, chamado de **pivô**, e colocá-lo em uma posição onde todos os elementos à esquerda são menores ou iguais e todos os elementos à direita são maiores.

**Merge sort:**

Sua complexidade é O(n log n), a mesma em comparação a outros algoritmos de divisão e conquista, mas é muito mais eficiente para entradas muito grandes em relação aos algoritmos de comparação e troca.

Seu funcionamento se resume em dividir o vetor ao meio gerando dois subvetores e, recursivamente, fazer o mesmo nestes subvetores. Após toda divisão, é feita, também recursivamente, a comparação e junção dos subvetores em vetores maiores até se tornar apenas um vetor com todos os elementos ordenados.

**Heapsort:**

É considerado um algoritmo instável. O comportamento do Heapsort é sempre O(n log n), independentemente de qualquer entrada, ou seja, em todos os casos. Este algoritmo utiliza uma estrutura de dados denominada heap binário, que é uma árvore binária mantida na forma de um vetor. O heap binário é usado para ordenar os elementos à medida que são inseridos por ele na estrutura. Então, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada. O heap é gerado e mantido no próprio vetor a ser ordenado. Para uma ordenação crescente, deve ser construído um heap máximo, onde o maior elemento fica na raiz. Já para uma ordenação decrescente, deve ser construído um heap mínimo (o menor elemento fica na raiz).

1. **Tabela:**

Foram utilizados 4 vetores de 25 mil elementos devido às limitações computacionais para totalizar a entrada sugerida de 100 mil.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Vetores  (Tam = 100000) | Bubble  sort | Selection  sort | Insertion  sort | Quick  sort | Merge  sort | Heap  sort |
| Aleatório | 7.328 s | 2.930 s | 1.668 s | 0.023 s | 0.035 s | 0.021 s |
| Crescente | 3.150 s | 2.928 s | 0.001 s | 3.166 s | 0.023 s | 0.014 s |
| Decrescente | 6.043 s | 2.972 s | 3.292 s | 3.157 s | 0.016 s | 0.014 s |

1. **Conclusão:**

O vetor preenchido com números aleatórios teve seu pior desempenho no bubble sort, pois este algoritmo realiza comparações entre dois elementos até que todo o vetor esteja ordenado. O fato de ser um vetor de entradas aleatórias consome mais tempo já que o loop é realizado diversas vezes. Já o melhor desempenho aconteceu no heap sort devido ao seu algoritmo ter uma árvore binária que ordena os elementos à medida que os insere na estrutura.

No vetor em ordem crescente temos o melhor desempenho sendo do insertion sort, pois ele apenas compara os elementos e, como já estão ordenados, não é necessário realizar trocas. O pior é no quick sort, devido a rotina de particionamento que gasta muito tempo separando os números menores que o pivô a sua esquerda e os maiores a sua direita para depois ordená-los.

Para o vetor decrescente o pior desempenho foi do bubble sort pelo fato de estar na ordem inversa, além de realizar várias comparações, realiza a troca desses elementos até que estejam devidamente ordenados. O melhor foi do heap sort devido a sua estrutura de árvore binária que compara e posiciona os elementos em um tempo menor que os outros algoritmos implementados.