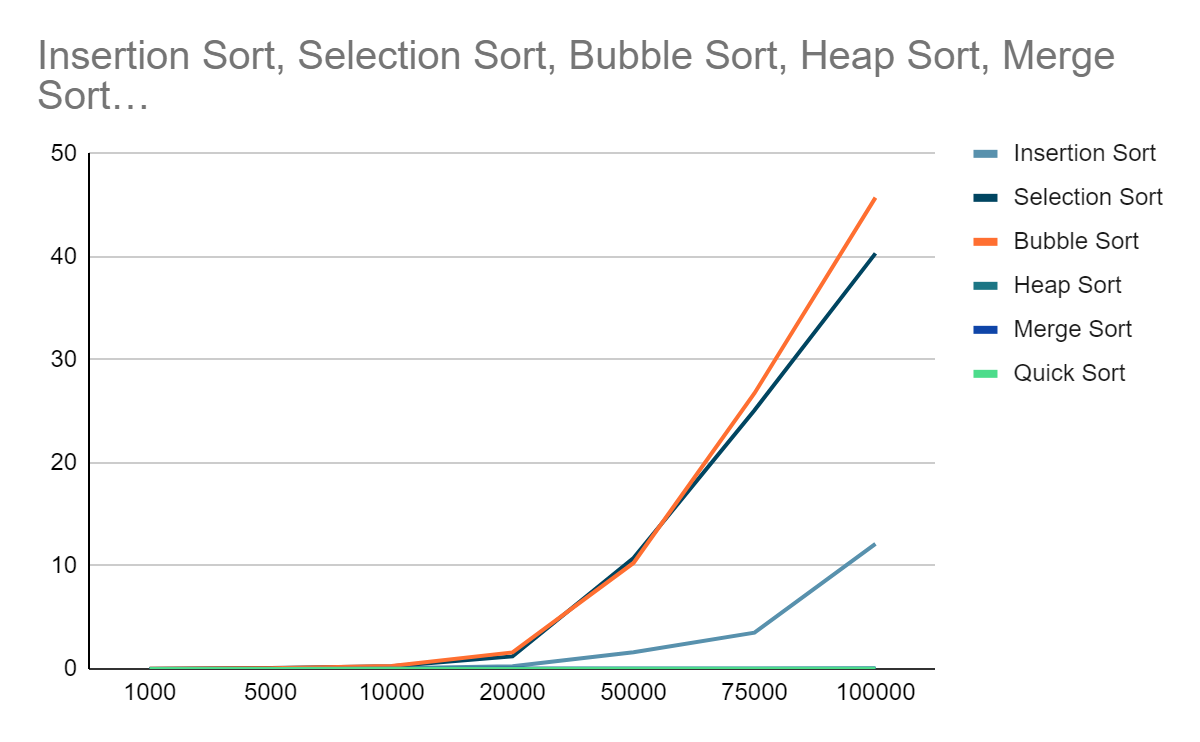
**Tabela de tempo de execução dos algoritmos de ordenação em segundos**

|  | **1000** | **5000** | **10000** | **20000** | **50000** | **75000** | **100000** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Insertion Sort** | 0.001926 | 0.037576 | 0.073472 | 0.255471 | 1.618404 | 3.512138 | 12.132465 |
| **Selection Sort** | 0.002612 | 0.077633 | 0.267643 | 1.212947 | 10.757487 | 25.100789 | 40.336916 |
| **Bubble Sort** | 0.002742 | 0.068202 | 0.273125 | 1.601178 | 10.284282 | 26.76092 | 45.753538 |
| **Heap Sort** | 0.000189 | 0.002351 | 0.001403 | 0.002687 | 0.008934 | 0.015204 | 0.028978 |
| **Merge Sort** | 0.000302 | 0.001769 | 0.003647 | 0.002759 | 0.010501 | 0.015072 | 0.02585 |
| **Quick Sort** | 0.000286 | 0.000539 | 0.003155 | 0.002273 | 0.010684 | 0.010454 | 0.016003 |

**Gráfico com os valores da tabela**



1)

**Insertion Sort**

Melhor caso: O(n)

Caso medio: O(n^2)

Pior caso: O(n^2)

Complexidade: O(1)

**Bubble Sort**

Melhor caso: O(n^2)

Caso medio: O(n^2)

Pior caso: O(n^2)

Complexidade: O(1)

**Selection Sort**

Melhor caso: O(n^2)

Caso medio: O(n^2)

Pior caso: O(n^2)

Complexidade: O(1)

**Heap Sort**

Melhor caso: O(nlog n)

Caso medio: O(nlog n)

Pior caso: O(nlog n)

Complexidade: O(1)

**Merge Sort**

Melhor caso: O(n\*log n)

Caso medio: O(n\*log n)

Pior caso: O(n\*log n)

Complexidade: O(n)

**Quick Sort**

Melhor caso: O(n\*log n)

Caso medio: O(n\*log n)

Pior caso: O(n^2)

Complexidade: O(log n)

2)

* **Insertion Sort**: O Insertion Sort, foi o algoritmo de ordenação mais eficiente até então. Com 100.000 números a serem ordenados, ele performou melhor que as outras 2 alternativas, que chegaram a ultrapassar 40 segundos, e o Insertion Sort fez a mesma ordenação não ultrapassando 15 segundos. Isso se dá pelo fato do Insertion Sort, percorrer a quantidade de números já ordenando eles.
* **Selection Sort**: Assim como o Bubble Sort, ao passar de 20 mil números ele começa a perder performance, isso se dá pelo fato de que naturalmente, o algoritmo verifica constantemente os números mínimos e ordenar eles antes, mesmo que esse número mínimo esteja no fim da lista de números e voltará a comparar a partir do elemento anterior + 1.
* **Bubble Sort**: Bem parecido com o Selection Sort, o Bubble compara todos pares colocando eles nas suas ordens, sempre voltando para o primeiro índice da lista de números para poder re-comparar e ver se todos estão nos seus lugares. Por esse motivo, comparar uma quantidade enorme de números com este não é tão eficiente.
* **Heap Sort**: Aparentando ser uma estrutura de árvore binária, o heap sort, cria uma árvore com todos elementos de um array, e vai jogar os números menores para baixo. Por sua estrutura, ele acaba sendo muito rápido para tratar números já desordenados, seu problema maior é quando precisamos de números ordenados, pois terá que efetuar mais um processo de ordenação (Crescente).
* **Merge Sort**: Com sua estratégia de dividir para conquistar, o array é dividido em dois e a partir daí inicia sua troca de posições, por ele utilizar mais um array auxiliar para montar todos os valores já ordenados, costuma ser mais rápido que outros algoritmos.
* **Quick Sort:** Este algoritmo aponta para dois valores nas extremidades e vem comparando eles, já os ordenando, por esse motivo é um dos algoritmos mais rápidos entre os que testamos, sua comparação é baseada em lançar ponteiros nos pivôs(alvos que serão comparados), tendo o algoritmo mais ordenado no fim das contas.