**Introdução**

**Quicksort**

**- Tamanho de listas:** Para listas muito grandes, o Quicksort pode não ser a melhor opção, pois a recursão profunda pode levar a um consumo excessivo de memória e tempo de execução. Para listas pequenas, o Quicksort geralmente é uma boa escolha, pois possui uma complexidade de tempo médio de O(n log n), o que o torna eficiente.

**- Distribuição de dados**: O Quicksort funciona melhor quando a distribuição dos dados é aleatória ou próxima de uma distribuição normal. Se a lista estiver ordenada ou quase ordenada, o Quicksort pode ter um desempenho ruim, pois a escolha do pivô pode levar a uma partição desequilibrada, resultando em uma complexidade de tempo de O(n^2). Listas com muitos elementos repetidos também podem afetar negativamente o desempenho do Quicksort, pois a partição pode ficar desequilibrada.

**Mergesort**

Possui uma complexidade de tempo de O(n log n).

**- Tamanho de listas:**

O Mergesort é particularmente eficiente para listas grandes, pois sua complexidade de tempo não depende da distribuição inicial dos dados.

Para listas pequenas, o Mergesort pode não ser a melhor escolha, pois o overhead de divisão e mesclagem das sublistas pode superar o benefício da complexidade de tempo.

**- Distribuição de dados:**

O Mergesort funciona bem independentemente da distribuição inicial dos dados, pois sua estratégia de divisão e mesclagem garante uma ordenação eficiente.

**Heapsort**

O Heapsort é um algoritmo de ordenação eficiente, com complexidade de tempo de O(n log n), e é particularmente útil para lidar com grandes volumes de dados.

**- Tamanho de listas:**

O Heapsort é uma ótima escolha para listas grandes, pois sua complexidade de tempo não depende da distribuição inicial dos dados.

Para listas pequenas, o Heapsort pode não ser a melhor opção, pois o overhead de construção e manutenção da heap pode superar o benefício da complexidade de tempo.

**- Distribuição de dados:**

O Heapsort funciona bem independentemente da distribuição inicial dos dados, pois sua estratégia de construção e extração da heap garante uma ordenação eficiente.

Mesmo para listas ordenadas ou quase ordenadas, o Heapsort mantém sua eficiência, pois não depende da escolha de um pivô como o Quicksort.

Listas com muitos elementos repetidos também não afetam significativamente o desempenho do Heapsort.

Portanto, o Heapsort é uma ótima escolha para listas grandes, independentemente da distribuição inicial dos dados. Para listas pequenas, outros algoritmos, como o Insertion Sort ou o Quicksort, podem ser mais eficientes.

**Selection Sort**

O Selection Sort é um algoritmo de ordenação simples e fácil de implementar, mas possui uma complexidade de tempo de O(n^2), o que o torna menos eficiente para grandes volumes de dados.

**- Tamanho de listas:**

Para listas pequenas, o Selection Sort pode ser uma opção razoável, pois sua simplicidade de implementação e baixo overhead podem compensar sua complexidade de tempo.

No entanto, para listas grandes, o Selection Sort não é a melhor escolha, pois sua complexidade quadrática pode levar a tempos de execução muito longos.

**- Distribuição de dados:**

O desempenho do Selection Sort não é significativamente afetado pela distribuição inicial dos dados, pois sua estratégia de encontrar o menor elemento e trocá-lo com o primeiro elemento da lista não depende da ordem dos elementos.

Portanto, o Selection Sort se comporta de maneira semelhante independentemente da distribuição dos dados.

**Insertion Sort**

O Insertion Sort é um algoritmo de ordenação simples e eficiente para pequenas listas, especialmente quando os dados estão quase ordenados.

**- Tamanho da lista:**

Para listas pequenas, o Insertion Sort é uma ótima escolha, pois possui uma complexidade de tempo de O(n^2) no pior caso, mas é muito eficiente para listas com poucos elementos.

No entanto, para listas grandes, o Insertion Sort não é a melhor opção, pois sua complexidade quadrática pode levar a tempos de execução muito longos.

**- Distribuição dos dados:**

O Insertion Sort se comporta melhor quando a lista está quase ordenada ou possui poucos elementos fora de ordem.

Nesse caso, o algoritmo precisa realizar poucas trocas para ordenar a lista, o que torna sua execução muito rápida.

Por outro lado, se a lista estiver muito desordenada, o Insertion Sort precisará realizar muitas trocas, o que prejudica seu desempenho.

**Bubble Sort**

O Bubble Sort é um algoritmo de ordenação simples, com uma complexidade de tempo de O(n^2) no pior caso, o que o torna menos eficiente para grandes volumes de dados.

**- Tamanho da lista:**

Para listas pequenas, o Bubble Sort pode ser uma opção razoável, pois sua simplicidade de implementação e baixo overhead podem compensar sua complexidade de tempo.

No entanto, para listas grandes, o Bubble Sort não é a melhor escolha, pois sua complexidade quadrática pode levar a tempos de execução muito longos.

**- Distribuição dos dados:**

O desempenho do Bubble Sort é influenciado pela distribuição inicial dos dados na lista.

Se a lista estiver quase ordenada, o Bubble Sort se comporta de maneira eficiente, pois requer poucas trocas para ordenar a lista.

Por outro lado, se a lista estiver muito desordenada, o Bubble Sort precisará realizar muitas trocas, o que prejudica seu desempenho.

**Testes**

Os testes de benchmark estão no arquivo ‘sort-final.py’. Os testes foram realizados utilizando o Python v3.12.7, Visual Studio Code - Insiders versão 1.96.0 90 (rev 1730315900) no CachyOS 64 bits (baseado em Arch Linux), utilizando o Kernel 6.11.5\_tkg\_bore-1.x86\_64. CPU: Ryzen 5 5500U (6/12), 20GB de RAM DDR4 em 3200mHz.

Ao executar os testes, o programa irá criar uma única lista de números com 2.000 números aleatórios, sendo eles entre 0 e 200, e então, executa cada algoritmo de ordenação sequencialmente copiando a lista original (sem modificá-la). Ao final da cada execução de cada algoritmo, o programa guarda o tempo decorrido para a realização da operação num dicionário chave-valor com chave sendo o nome do algoritmo e o valor sendo o tempo decorrido em milissegundos. Ao final da execução de todos os algoritmos, o programa irá mostrar os dados coletados.

Desde o Python 3.11, o algoritmo padrão utilizado para a ordenação de listas é o Powersort, que é uma melhoria do algoritmo Tim Sort, usando em versões prévias.