Análise do processo de ordenação dos algoritmos

Equipe:
Evellyn Rodrigues da Rocha;
Kauã Fellipe Pereira Bispo;
Lara Fernanda Amorim Alves Cavalcante.

- **Base de dados utilizada**
  A base de dados foi utilizada para ordenar os planetas a partir do ano de descoberta.

  Link:
  https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=PS

- **Componentes de Hardware**

  Dispositivo 1:

  Processador: Core i3 Intel 7th Gen
  Memória RAM: 4GB
  SO: Ubuntu 20.04 LTS
  Kernel: 5.15.0-71-generic

  Dispositivo 2:

  Processador: Core i5 intel 11th Gen
  Memória RAM: 8GB
  SO: Windows 11 22H2
  Kernel: Windows NT

  Dispositivo 3:

  Processador: Intel Pentium E5700 (2) @ 3.003GHz
  Memória RAM: 8GB DDR3
  SO: Pop_Os! 22.04 LTS
  Kernel: 6.2.6-76060206-generic
  SSD: Crucial Ballistix BX500

- **Tempo de execução**

Este foi o código utilizado para medir o tempo de execução de cada algoritmo de ordenação. Começamos importando primeiramente o time e logo em seguida temos o start_time que irá registrar o tempo inicial e o end_time registrará o tempo final, para que possamos ter a diferença entre esses tempos e por fim, imprimir o tempo de execução em segundos na tela.

```
import time

start_time = time.time()
end_time = time.time()
execution_time = end_time - start_time

print(f'O tempo de execução: {execution_time}s')
```

1. Merge Sort: No dispositivo 1, o tempo de execução variou de 0.6 a 1s, tendo uma frequência maior de 0.6s e 0.7s. No dispositivo 2, o tempo de execução do algoritmo variou de 0.9 a 1 segundo, tendo uma frequência maior de 1 segundo. No dispositivo 3, o tempo de execução do algoritmo variou de 0.8s a 1,2s, tendo uma frequência maior de 1s.
2. Bucket Sort: No dispositivo 1, o tempo de execução variou de 1 a 3s, tendo uma frequência maior de 1s a 2s. No dispositivo 2, o tempo de execução variou entre 1 a 1.5s, tendo uma frequência de 1.1s e 1.3s. No dispositivo 3, o tempo de execução variou de 0.04s a 0.07s tendo uma frequência de 0.06s
3. Quick Sort: No dispositivo 2, o tempo de execução variou de 33 a 79 segundos, tendo uma frequência maior de 39.5 segundos. No dispositivo 3, o tempo de execução do algoritmo variou de 1,9 minutos a 2,1 minutos, tendo uma frequência maior de 2 minutos.

- **Complexidade**

1. Merge Sort -

A ideia central do merge sort é dividir uma lista em sublistas, as quais são recursivamente divididas pela metade até que tenham apenas um elemento. Nesse sentido, ao finalizar essas divisões sucessivas e obter sublistas com apenas um elemento, é iniciado as comparações de maior e menor para que se possa ordenar a lista. Resumindo, a ideia é dividir uma lista pela metade, tendo um lado esquerdo e um lado direito, até que se tenha apenas sublistas de apenas um elemento, e então, é

feita a comparação de maior e menor entre os elementos do lado direito e do lado esquerdo, de modo a modificar a lista para que ela fique com os valores ordenados.

O merge sort possui duas chamadas recursivas, as quais têm como principal finalidade dividir o array pela metade. Logo, a complexidade passa a ser 2*T(n/2). Ademais, a função merge, que também foi utilizada, tem complexidade O(n). Portanto, a complexidade é 2 * T(n/2) + n, logo, fornece um custo total n * log n, sendo n o número de itens na lista de dados, sendo, nesse caso, 3422.

Diante disso, a complexidade do algoritmo é O(n log(n)), a qual é considerada bastante eficiente, sendo que ela cresce mais rapidamente que o O(n) e mais lentamente que o O(n^2), sendo assim, é uma das mais rápidas.

O merge sort é muito útil para ordenar grandes quantidades de dados, uma vez que a complexidade de tempo é relativamente independente do número de inversões existentes no conjunto de dados, ou seja, independente da desordenação do conjunto, visto que a forma como ele divide e mescla as sublistas ajuda a remover essas inversões em etapas. Portanto, o merge sort é uma boa escolha, pois mesmo pegando um conjunto extremamente desordenado, ele consegue ordenar sem muita dificuldade.

2. Bucket Sort -

A ideia principal do bucket sort é dividir o array em um determinado número de buckets, o que dependerá do exemplo em questão para que em seguida cada elemento seja colocado no seu devido bucket que será uma lista e a adição desses elementos nessas listas/buckets vai depender dos elementos, por exemplo se for valores de 1 a 100, cada bucket pode ter um determinado grupo de números, como: 1 a 9, 10 a 19 e assim por diante, depois deste processo, cada lista será ordenado de forma individual com as funções sorted ou sort ou até mesmo algum algoritmo de ordenação e no final, esses buckets individuais ordenados serão concatenados formando uma única lista.

A complexidade do algoritmo bucket sort vai depender dos números de buckets criados, o determinado tipo de ordenação que cada bucket individual vai ter e também, da separação dos elementos em cada bucket. Assim, normalmente a complexidade desse algoritmo é O(n), mas em alguns fatores podem mudar esse resultado, como: o aumento de buckets, a complexidade do algoritmo de ordenação escolhido for maior que o do bucket sort e entre outros.

No código do bucket sort vai ter 3 repetições, na qual a primeira está relacionada na divisão dos elementos em seus buckets individuais, a segunda está relacionada com a ordenação e a terceira está relacionada com a concatenação desses buckets. Assim, a complexidade passa a ser O(n + n + n + knlog n) que resulta em O(n + knlog n), em que n é o número de elementos total, k o número de baldes e o n do log o número de elementos de cada bucket individual, então se o valor k for bastante pequeno, a complexidade estará voltada a O(n), mas caso o valor k seja bem maior em relação a n, a complexidade será O(knlog n).

Dessa forma, como na base de dados que escolhemos temos ao todo 3422 elementos e no código foram criados 4 buckets, podemos perceber que o valor de n é bem maior que k e assim, a complexidade será O(n).

O bucket sort é bastante útil quando os elementos da entrada são dados uniformemente distribuídos e intervalos conhecidos, ou seja, estão bem distribuídos nos buckets e os intervalos podem ter uma lógica de classificação para que possam ser adicionados em seu respectivo bucket. Com isso, a eficiência deste algoritmo depende bastante da divisão dos elementos e da quantidade de buckets, se esses fatores estiverem de acordo, o bucket sort pode ser um algoritmo bem eficiente para grandes bases de dados.

3. Quick Sort - O(n log(n))

O quicksort ordena a lista utilizando como base um elemento pivô que pode ser estabelecido de inúmeras formas diferentes a depender da situação e do tamanho da base de dados, uma vez que toda a lista será ordenada utilizando o pivô como referências, da seguinte forma.

Estabelecido o pivô, a lista é separada em duas sublistas, uma a esquerda que serão os elementos menores que o pivô e outra à direita com os elementos maiores que o pivô, a partir disso as comparações serão feitas usando dois ponteiros, um que começa no ínicio da lista e outro no final da lista, eles vão se movimentando com a seguinte regra.

1. Move o ponteiro do início para a direita até encontrar um elemento maior que o pivô.
2. Move o ponteiro do final para a esquerda até encontrar um elemento menor que o pivô.
3. Se os ponteiros ainda não se cruzaram, troca os elementos apontados pelos ponteiros.
4. Repete os passos 1-3 até que os ponteiros se cruzem.

Ao final, separamos a lista em duas sublistas e aplicamos o quicksort recursivamente nelas, até que toda a lista esteja ordenada.

No geral, o quicksort é um bom algoritmo para uma complexidade O(n log(n)) porém pode decair para $O(n^2)$ em casos onde a lista possui todos os elementos iguais ou já esteja ordenada.

- Conclusão

Diante dos testes realizados com os três algoritmos de ordenação em três diferentes máquinas, foi possível concluir que o algoritmo mais eficiente para a base de dados utilizada foi o Merge Sort. O tempo de execução do Merge Sort foi menor do que os outros algoritmos em dois dispositivos, indicando sua eficiência. Além

disso, a complexidade do Merge Sort é O(n log(n)), o que é melhor do que a complexidade do Bucket Sort, que é O(n).

Apesar de a complexidade do Merge Sort e do Quick Sort ser a mesma, o Quick Sort não apresentou eficiência em comparação com os outros dois algoritmos na base de dados utilizada. Isso porque, o Quick Sort chegou a demorar minutos para executar.

No entanto, ao considerar os componentes físicos de cada máquina, o Bucket Sort mostrou-se mais eficiente no dispositivo 3, executando em menos de 1 segundo. Portanto, para essa máquina específica, o Bucket Sort é a melhor escolha de algoritmo. Logo, a partir disso é possível notar que a escolha do melhor algoritmo depende do dispositivo utilizado.

Conclui-se, portanto, que a partir das considerações e comparações levantadas, o Merge Sort é o algoritmo mais eficiente para a base de dados considerada, sendo seguido pelo Bucket Sort, que também apresentou uma boa eficiência. No entanto, é importante ressaltar que a escolha do algoritmo ideal pode variar dependendo das características do dispositivo que está sendo usado, como foi citado anteriormente no Bucket Sort, que apresentou excelente desempenho em um dos dispositivos.