UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA DE GRADUAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FELIPE DE OLIVEIRA BRENNER

RELATÓRIO ALGORITMO DE ORDENAÇÃO QUICKSORT

Algoritmos e Programação: Árvores e Ordenação

São Leopoldo
2020

1. INTRODUÇÃO

O referente relatório apresenta uma análise do conceito do algoritmo de ordenação quicksort, sendo constituído por uma descrição de suas características, por gráficos do tempo de execução e de trocas realizadas e cálculo da complexidade de um algoritmo que foi escrito e executado em python. Este referente algoritmo de particionamento é um dos mais eficientes, um dos mais utilizados, e sua ideia em geral é basicamente dividir uma lista de itens em duas partes menores, ordená-las independentemente e combinar os dois resultados para se ter a ordenação completa.

2. CARACTERÍSTICAS

O quicksort, como já introduzido, é o algoritmo mais eficiente na ordenação por comparação, e entrando mais em detalhe, nele se escolhe um elemento chamado de pivô e a partir dele é organizada a lista para que todos os números anteriores a ele sejam menores que ele, e todos os números posteriores a ele sejam maiores que ele. Ao final desse processo o número pivô já está em sua posição final. Os dois grupos desordenados recursivamente sofreram o mesmo processo até que a lista esteja ordenada**.**

Comparando com o algoritmo mergesort, pois é um outro algoritmo que eu já conheço melhor, ambos usam o método de divisão e conquista e recursividade. No mergesort a etapa da divisão não faz muita coisa, deixando toda a lógica e processamento das comparações na etapa de combinação. Já no quicksort ocorre o oposto, na etapa de divisão é onde ocorre todo o trabalho, e a etapa combinação não faz praticamente nada. Outra diferença entre estes dois algoritmos é que o quicksort tem seu tempo de execução no pior caso tão ruim quanto o dos algoritmos de ordenação por seleção e inserção $O(n^2)$, mas seu tempo de execução médio é tão bom quanto o mergesort O($n \log n$).

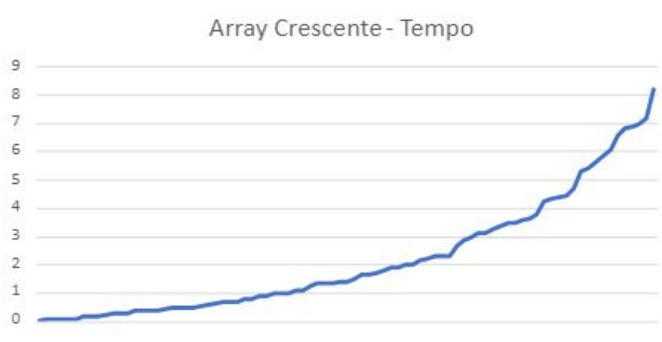
É válido pensar no quicksort ao invés do mergesort, apesar de ambos serem igualmente bons, devido ao fator constante oculto na notação Big O do quicksort ser muito bom. Na prática o quicksort tem um desempenho melhor que o mergesort, e é significativamente melhor que os algoritmos de seleção e inserção.

3. GRÁFICOS

Os gráficos ordenados foram executados partindo de uma entrada de tamanho 100 e aumentando de 100 em 100, e os gráficos com números aleatórios partindo de uma entrada de tamanho 1500 e aumentando de 150 em 150, pois este algoritmo é mais rápido para números aleatórios do que quando está ordenado, devido a sua lógica de particionamento explicada.

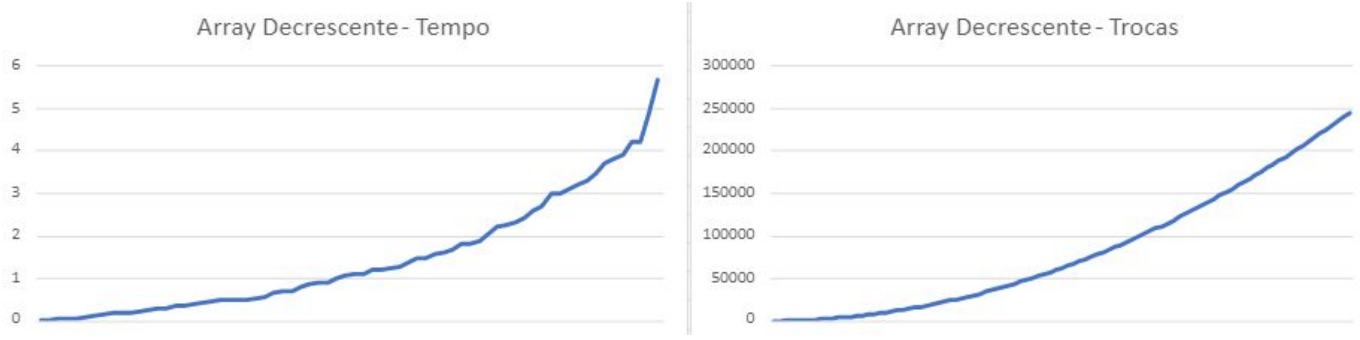
As curvas resultantes foram estas a seguir, no qual o eixo y dos gráficos a esquerda está na escala de segundos, e o eixo y dos gráficos a direita é o número de trocas realizadas. Vale ressaltar que os dados dos gráficos não foram coletados na mesma execução de código, nem mantidos executando durante o mesmo período de tempo, o que impossibilita a comparação dos gráficos nos exatos mesmos pontos do eixo x, mas a curva dos mesmos seguiram o conceito de cada caso.

3.1 Array já ordenado (crescente)



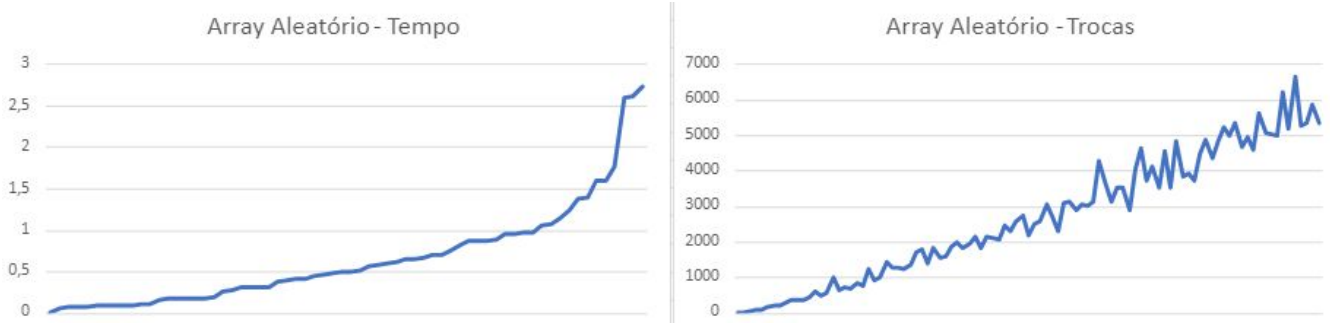
Nota-se que os gráficos seguiram a curva da complexidade para um array ordenado, que é o mais demorado, que é $O(n \log n)$, complexidade que será explicada e calculada no próximo tópico.

3.2 Array ordenado (decrescente)



Assim como o ordenado de forma crescente, o decrescente também possui o maior tempo de execução $O(n\ log\ n)$. Ressaltando novamente que os dados não foram adquiridos na mesma execução de código e nem no mesmo período de tempo, o que explica o tempo máximo não ter sido o mesmo entre o crescente e decrescente, nem o número de trocas máximas ter sido as mesmas.

3.3 Array com números aleatórios



Uma lista aleatória é quando se temos o melhor caso, que possui complexidade $O(n^2)$, que é quando o array é dividido exatamente no meio sempre. É válido explicar que o gráfico de trocas possuiu estas variações devido a ordenação dos arrays de entrada serem aleatórios, variando então a quantidade de trocas. Mas conforme o número foi aumentando, logicamente foi se aumentando a média e tornando o gráfico crescente sempre, e com uma execução muito mais rápida do que os arrays ordenados.

4. COMPLEXIDADE

```
1   from collections import deque
2
3   def iterativeQuickSort(a):                          #   Melhor Caso        Pior caso
4       stack = deque()                                 #       1                   1
5       start = 0                                       #       1                   1
6       end = len(a) - 1                                #       1                   1
7       stack.append((start, end))                      #       1                   1
8       while stack:                                    #       n                   n
9           start, end = stack.pop()                    #       n-1                 n-1
10          pivot = a[end]                              #       n-1                 n-1
11          pIndex = start                              #       n-1                 n-1
12          for i in range(start, end):                 #       (n-1)log(n)         (n-1)(n/2)
13              if a[i] <= pivot:                       #       (n-1)log(n)         (n-1)(n/2)
14                  temp = a[i]                         #       (n-1)log(n)         (n-1)(n/2)
15                  a[i] = a[pIndex]                    #       (n-1)log(n)         (n-1)(n/2)
16                  a[pIndex] = temp                    #       (n-1)log(n)         (n-1)(n/2)
17                  pIndex = pIndex + 1                 #       (n-1)log(n)         (n-1)(n/2)
18          temp = a[pIndex]                            #       n-1                 n-1
19          a[pIndex] = a[end]                          #       n-1                 n-1
20          a[end] = temp                               #       n-1                 n-1
21          pivot = pIndex                              #       n-1                 n-1
22          if pivot - 1 > start:                       #       n-1                 n-1
23              stack.append((start, pivot - 1))        #       n-2                 n-2
24          if pivot + 1 < end:                         #       n-1                 n-1
25              stack.append((pivot + 1, end))          #       0                   0
26
27  if __name__ == '__main__':
28      a = [7,6,5,4,3,2,1]
29      iterativeQuickSort(a)
30      print(a)
```

4.1 Complexidade no melhor caso

Ocorre quando o array é dividido exatamente no meio sempre

$$T(n) = 4 + n + 9(n-1) + 6(n-1)\log n + (n-2)$$

$$T(n) = (6n - 6)\log n + 11n - 7$$

$$T(n) = 6n \log n - 6 \log n + 11n - 7$$

$$T(n) = O(n \log n)$$

4.2 Complexidade no pior caso

Ocorre quando o array está ordenado em ordem crescente ou decrescente

$$T(n) = 4 + n + 9(n-1) + 6(n-1)(n/2) + (n-2)$$

$$T(n) = (6n - 6)(n/2) + 11n - 7$$

$$T(n) = 3n^2 + 8n - 7$$

$$T(n) = O(n^2)$$