Análise QuickSort

Nicolas Beraldo

Maio 2019

Analisaremos o código de ordenação "QuickSort" desenvolvido pelo aluno Nicolas Beraldo que se baseia no pseudocódigo de Tony Hoare, desenvolvido em 1959.

1 Código

O código foi totalmente desenvolvido em python, possuiu duas funções, uma nomeada "quicksort" e outra "partition", a primeira é a que gera as chamadas recursivas da própria função enquanto a outra é a que seleciona o pivô, elemento utilizado para começar a ordenação, e separa de um lado do pivô os elemento menor a ele e do outro lado os maiores que ele. A seguir o código desenvolvido:

Listing 1: "QuickSort"

```
1  def quicksort(data, low, high):
2      if low < high:
3          pivot = partition(data, low, high)
4          quicksort(data, low, pivot)
5          quicksort(data, pivot + 1, high)
6
7  def partition(data, low, high):
8      pivot = data[low + math.floor((high - low)/2)]
9      start = low - 1
10     end = high + 1
11     while True:
12         while True:
13             start += 1
14             if data[start] >= pivot:
15                 break
16         while True:
17             end -= 1
18             if data[end] <= pivot:
19                 break
20         if start >= end:
21             return end
22         data[start], data[end] = data[end], data[start]
```

2 Análise de tempo

A primeira chamada da função "QuickSort" é feita após o recebimento da string do tipo JSON e os parâmetros são a lista com os elementos, a posição do elemento mais baixo (low = 0) e a posição do último elemento (high = n - 1). Iniciando a análise onde ocorre o caso base começaremos a avaliar a função "partition", nela podemos perceber que há três "while" esses são os loops feitos para percorrer a sub lista com finalidade de organizara-la. As linhas 13 e 17 são complementares uma a outra, assim como as linhas 14 e 18 e 12 e 16 também são e essas seis linhas são limitadas pela linha 20, como há um contator da

variável “start” e “end” que cresce a cada loop e há uma condição de parada “if“ quando “start” for maior que “end” a função se encerrará devido ao “return” ali posto. Com essa condição de parada e as condições de paradas dos loops de incremento das variáveis “start” e “end” é impossível que a soma das posições seja maior que $n + 1$(tempo de execução da linha 20) e devido a isso e que as condições de parada possuem uma condição de maior/menor e igual à soma dos tempos de execução das linhas 12, 13, 14, 16, 17 e 18 é $3n+6$ pois as linhas 13 com 17, 14 com 18 e 12 com 16 se complementam para formar a lista inteira e devido ao igual se sobrepõem. Para finalizar a análise da função “partition” falta analisar as linhas restantes, onde as linhas 8, 9, 10, 15, 19, 21 têm tempo constante e igual a 1, a linhas 11 não tem como passar de $n + 1$ e como a linha 22 pode ocorrer depois de um “return” o seu tempo é n. Assim a função “partition” tem um tempo de execução de $6n + 14$.

A seguir analisaremos o pior caso, onde irá ocorrer se o pivô a ser escolhido for “low + 1” ou “high - 1”, pois nesse caso serão necessários n–1 chamadas da recursividade para ordenar a lista. Este caso de quadrático($O(n^2)$) já que para a “partition” é $O(n)$ e será necessário $T(n - 1)$ chamadas para ordenar tudo, o que equivale a $O(n)$.

Para o caso médio ondem o a sub lista será sempre dividida em locais mais aleatórios e normalmente tendentes a se aproximarem a $n/2$ a necessidade da chamada da recursividade será em menor quantidade ($\log_2 n$), para outros casos próximos a $n/2$ e casos de divisão e conquista, como ”MergeSort” o seu tempo de execução pode ser assumido sendo $O(\log n)$. Assim se juntarmos a partição os tempos de execução das funções “partition” e “QuickSort” teremos um tempo $O(n \log n)$