Analysis of Algorithms – HW1

Li Hen Chen

Department of Computer Science & Engineering, Texas A&M University

1. Answer the following questions, and give a brief explanation for each of your answers.

First of all, big O may represent worst-case time complexity or average-case time complexity.

We have to know that time taken by QuickSort in general can be written as following.

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

And there are three cases for time complexity of QuickSort

Worst Case : The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

$$T(n) = T(n-1) + \Theta(\mathrm{n})$$

The solution of above recurrence is $\Theta(\mathrm{n}^2)$

Best Case : The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(\mathrm{n})$$

The solution of above recurrence is $\Theta(\mathrm{n}\log n)$

Average Case : We can get an idea of average case by considering the case when partition puts $\Theta(\frac{9n}{9})$ elements in one set and $\Theta(\frac{9n}{10})$ elements in other set. Following is recurrence for this case.

$$T(n) = T\left(\frac{9}{10}n\right) + T(\frac{9}{9}n) + \Theta(\mathrm{n})$$

The solution of above recurrence is $\Theta(\mathrm{n}\log n)$

a) True or False: Quicksort takes time O($n\log n$);

According the above, the answer is True for the average case.

b) True or False: Quicksort takes time O(n^2);

According the above, the answer is True for the worst case.

Merge Sort

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \Theta(n)$$

The above recurrence can be solved either using Recurrence Tree method or Master Theorem. It falls in case II of Master Theorem and solution of the recurrence is $\Theta(n\log n)$. Time complexity of Merge Sort is $\Theta(n\log n)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

c) True or False: Mergesort takes time O($n\log n$);

According the above, the answer is True for all cases.

b) True or False: Mergesort takes time O(n^2);

According the above, the answer is False

2. Solve the following recurrence relations:

The runtime of an algorithm such as $T(n)$ can be expressed by the recurrence relation.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where $f(n)$ is the time to create the subproblems and combine their results in the above procedure. This equation can be successively substituted into itself and expanded to obtain an expression for the total amount of work done. The master Theorem allows many recurrence relations of this form to be converted to Θ-notation directly, without doing an expansion of the recursive relation.

Master Theorem case II:

if $f(n) = \Theta(n^{\log_b a} \log^k n)$ then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$ $where\ k \geq 0$

Master Theorem case III:

If $f(n) = \Omega(\mathrm{n}^{\mathrm{c}})$ then $T(n) = \Omega(\mathrm{f(n)})$

a) $T(1) = O(1)$, and $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$

$f(n) = \Omega(\mathrm{n}^{\mathrm{c}})$ At this case where a=2,b=2, $f(n) = n^2$ Thus according to Master Theorem case III we can obtain.

$$T(n) = \Theta(\mathrm{n}^2)$$

b) $T(1) = O(1)$, and $T(n) = 2T(n-2) + O(n)$

So we can calculate the recurrence relation.

$$\begin{aligned} T(n) &= 2T(n-2) + O(n) \\ &= 4T(n-4) + 2O(n) + O(n) \\ &= O(n) + 2O(n) + 4O(n) + \cdots + nO(n) \end{aligned}$$

According to the above equation and the geometric sequence $\sum_{k=1}^{n} ar^{k-1} = \frac{a(1-r^n)}{1-r}$

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

3. Consider the following operation on a set S: Neighbors(S, x): find the two elements y1 and y2 in the set S, where y1 is the largest element in S that is strictly smaller than x, while y2 is the smallest element in S that is strictly larger than x. Develop an O(log n)-time algorithm for this operation, assuming that the set S is stored in a 2-3 tree. Hint: the element x can be either in or not in the set S.

For the s y1 (1 is the largest element in S that is strictly smaller than x) and y2 (the smallest element in S that is strictly larger than x)

1. Like the insertion find the leaf position s should be insert.
2. if there are suitable values for y1 and y2,which means at the same leaf y1<x<y2, return y1 and y2.
3. if there is no suitable value, recursively find the suitable values.
    for y1
        find parent node that has a child larger (value nearest s) than s
        then for this parent node find the smallest element in this subtree.
    for y2
        find parent node that has a child smaller (value nearest s) than s
        then for this parent node find the largest element in this subtree.

For this algorithm the time complexity is obvious O(log n), because for the step 1 the time complexity is O(log n), and the step 3's time complexity is also O(log n) for the worst case. Step1 and step 3 are not rely on each other so the time complexity of whole algorithm is O(log n).

4. Consider the following problem: given a 2-3 tree T of n leaves, and an integer k such that log n ≤ k ≤ n, find the k smallest elements in the tree T. Develop an O(k)-time algorithm for the problem. Give a detailed analysis to explain why your algorithm runs in time O(k).

In order to meet the requirement that the time complexity of the algorithm should be O(k) rather than $O(\log n)$, I decide to implement a recursive function to meet the requirement. The main idea of the algorithm is by recursively calling the function on every subtree of the node, consecutively, from the left-most to the right most and with decreasing parameter k. The procedure is as the following.

1. Let the original/ current tree to be R, starts recursion by calling the function on R's left-most sub-tree with the same k as R receives.

2. If calling the function on a subtree of R successfully return k-th key, then that's the answer and return it.
3. If calling the function on some subtree T of R couldn't find the k-th smallest key, but instead returns the size of T, say n (< k), then:
   - If T is the right-most subtree, then R has fewer than k items, returns the size of R (which would have been found by summing the sizes of all its subtrees and the number of keys in R's root).
   - If n == k-1, then the k-th smallest key is the key immediately to the right of T
   - If n < k-1, then recurse on the subtree S immediately to the right of T with argument k-n-1 using the same function (i.e., to find the (k-n-1)-th key in S)

Analysis : As the question mentioned, $\log n \leq k \leq n$. At the step 1, we have travel from the root to the leaves where the time complexity is $O(\log n)$ obviously. Then we analyze step 3. At the step 3, the time complexity depends on k, because we start our algorithm from the left-most leaves. If the answer is not in the current leaves, then recursively search the right tree of the subtree(S) until find the kth key. We can tell from that the time complexity is determined by the leaves searched by the algorithm. For the recurrence computation, it has to take maximum $k + \log n$ $(tree's\ height)$ steps to find the result. As a result, step 3's time complexity is O(k) which leads to the time complexity of the whole algorithm is also O(k).