CSCE 629: Analysis of Algorithms
Home Work - 1

Shabarish Kumar Rajendra Prasad
UIN - 228000166

September 2019

1 Question 1

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

The statement is **False**. Quicksort, on the worst case, takes a time of αn^2, where α is a constant and n is the number of elements in the array. And hence, the general statement that Quicksort takes a time of $O(n \log n)$ is false.

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

The statement is **True**. As mentioned in the previous section, the worst case time complexity of Quicksort is $\Omega(n^2)$. Hence the statement is true.

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

The statement is **True**. The worst case time complextity of Mergesort is $\Omega(n \log n)$. Therefore, the statement that Mergesort takes time $O(n \log n)$ is true.

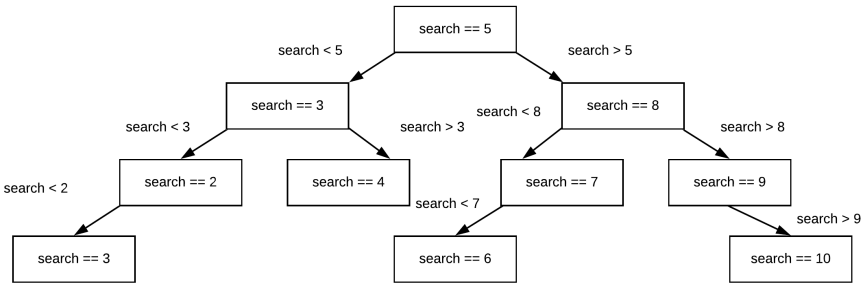
d) True or False: Mergesort takes time $O(n^2)$

The statement is **True**. As mentioned above, the worst case time complextity of Mergesort is $\Omega(n \log n)$. In other words, the worst case time taken by Mergesort is bounded by $\alpha n \log n$, where α is a constant. And thus we know, $\Omega(n \log n) \leq \alpha n \log n \leq \alpha n^2$. Now, by definition of the big O-notation, $O(n \log n) = O(n^2)$. And hence, this statement is true.

2 Question 2

Question: Prove: any comparison-based searching algorithm on a set of n elements takes time $\Omega(\log n)$ in the worst case.

Proof: any comparison based searching algorithm organises the set of elements in a decision-tree structure while searching. For example consider the binary search algorithm on the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. This algorithm generates the following decision-tree.



Decision-tree for the taken set

Consider that the tree generated for each of any comparison-based searching algorithm on a set of a total of n elements has the height h. By the property of binary-trees, we know that the total number of elements in the tree is bounded by the value 2^h. i.e.,

$$n \leq 2^h$$

Taking logarithm on both sides,

$$\log n \leq h$$

And since the log function is monotonically incresing, we can say,

$$\log n = h$$

On a comparison-based searching algorithm, the worst case time taken will be equal to the height of the binary-tree, since we have to go until the leaf node on the worst case, to find the element or to assert that the element is not found. In other words, the worst case time complexity of any comparison-based searching algorithm is $\Omega(h)$. Since we proved $h = \log n$ above, we can describe the worst case time complexity of the problem as $\Omega(\log n)$.

3 Question 3

The algorithm is given below:

Algorithm: Neighbors(S, x)

```
function Neighbors(r,x):

    y1 = findy1(r,x)
    y2 = findy2(r,x)
    return [y1,y2]


function findy1(r,x):

    if r == Null:
        return Null

    if (r is leaf):
        if r.k1 >= x:
            return Null
        else:
            return r.k1

    if r.k1 >= x:
        return findy1(r.p1)
    else if r.k2 >= x:
        temp = findy1(r.p2)
        if temp == Null:
            return r.k1
        else:
            return temp
    else if (r has a third child):
        temp = findy1(r.p3)
        if temp == Null:
            return r.k2
        else:
            return temp
    else if (r does not have a third child):
        if (r.k2 < x):
            return r.k2
        else:
            return Null
```

```
function findy2(r,x):

    if r == Null:
        return Null

    if (r is leaf):
        if r.k1 <= x:
    else:
        return r.k1

    if r.k1 > x:
        return findy2(r,x)
    else if r.k2 > x:
        return findy2(r,x)

    else if (r has a third child) and (r.k3 > x):
        return findy2(r,x)
    else:
        return Null
```

Both the functions, findy1(r,x) and findy2(r,x) executes h times, where h is the height of the tree, and takes constant time in each node.Since the height of 2-3 tree is bounded by $\log n$, the time complexity of the entire program is bounded by $O(\log n)$.

4 Question 4

The algorithm is given below:

Algorithm: SmallElements(r)

```
def SmallElements(r):

    if r == Null:
        return

    if (r is leaf):
        Add r.k1 to output_list
        k = k-1
        return

    if k>0:
        SmallElemnets(r.p1)
    if k>0:
        SmallElemnets(r.p2)
```

```
if (r has a third child) and (k>0):
    SmallElemnets(r.p3)

return
```

Explanation

This algorithm traverses the 2-3 tree from the root, navigates to the left child from each node and reaches leftmost leaf node, if $\log n \leq k \leq n$ and $n > 0$. After reaching the leaf node, the algorithm adds the value in the leaf to a global output list and decrements the global variable k. Here k is maintained as a global variable, since the decremented value in one recursive call should be reflected in all the other recursive calls. After adding the first element to the list, the algorithm goes up by one level and gets the middle and right children as they will contain the next smallest elements respectively. This process is repeated until k becomes zero. Here, the case where $k > n$ is not handled as it is given in the question that $k \leq n$.

Analysis

Consider the node n 2-3 tree and suppose the subtree rooteed at n has m leaf nodes, then the total number of nodes to be visited in the subtree is bounded by $(h_n - 1) + 2m$, where h_n is the height of the subtree rooted at n. Now the total number of nodes in the subtree including the node n will be bounded by $(h_n - 1) + 2m + 1$ or $h_n + 2m$.

By property of 2-3 trees, $h_n \leq \log n$ and thus the total number of nodes visited becomes bounded by $\log n + 2m$. If we are visiting k nodes which are inside the subtree of n, and since we are spending a constant time on each node, then the time complexity of the program becomes bounded by $O(\log n + k)$. Now since we know, $k \geq \log n$, the time complexity of the program can be described as $O(k)$.