Muhammad Mazharul Islam
2334105 3
Section-04 (A-1)

Solutions of Question-01

(a)

For the above portion, we see a nested loop. For a large value of n, the outer loop executes (n-1) times and the inner loop executes (n-1) times as well. But, for each outer loop iteration the inner loop executes (n-1) times. Hence,

$$T_1(n) = (n-1) \times (n-1)$$
$$= (n-1)^2$$
$$= n^2 - 2n + 1$$
$$= O(n^2)$$

Now coming to the second portion wee see three loops — outer loop, middle loop, and inner loop. If we ~~take~~ notice carefully the inner loop starts at $k=0$ and runs as long as $k<=j$ and gets multiply by 3 after each iteration. As it's initialized at 0 that means after each iteration k will still remain 0 as $3*0 = 0$. Hence the condition $k<=j$ will always be true and the loop will never stop. So, it's an infinite loop which will take infinite amount of time. Hence, for the second portion :-

$$T_2(n) = O(\infty)$$

Therefore, the overall time complexity of the given code snippet —

$$T(n) = T_1(n) + T_2(n) = O(n^2) + O(\infty) = O(\infty)$$

(Answer)

Solutions of Question-2

(a)

```
Algorithm BoundSearch(A, key)
{
    n ← A.length
    left ← 0
    right ← n-1
    fIdx ← -1
    while (left <= right)
    {
        mid = (left + right)//2
        if(A[mid] >= key)
        {
            fIdx = mid
            right = mid - 1
        }
        else
        {
            left = mid + 1
        }
    }
    return fIdx
}
```

(b)

Let me define 2 lambda functions. I'm using python equivalent syntax but imagine as if I wrote an algorithm-

```
lower = lambda ele, x : ele >= x
upper = lambda ele, x : ele > x

Algorithm boundSearch(A, x, compare)
{
    n ← A.length
    left ← 0
    right ← 0
    bound ← -1      // Imagine, if you will that -1 is an
                       invalid index for the list.
    while (left <= right)
    {
        mid = (left + right) // 2
        if (compare(arr[mid], x))
        {
            bound ← mid
            right ← mid - 1
        }
        else
        { left ← mid + 1
        }
    }
    return bound
}

Algorithm Search (A, key)
{
    startIndex ← boundSearch(A, key, lower)
    ubound ← boundSearch(A, key, upper)
    frequency ← ubound - startIndex
    return (startIndex, frequency)
}
```

Solution of Question -03

First of all, we're basically using a binary search algorithm. Point to notice, the given list is an unsorted list and we know we have to pass a sorted list as argument if we want the algorithm to work properly for all test cases. Anayways, for the given test case $T=2$ the algorithm will be able to find the search value.

Now, let me show you the values of L, R, m in each step:-

# of steps	L	R	m
0	0	7	3
1	0	2	1

Solutions of Question -04

(a)

Actually, it depends on some factors such as the size of the dataset and the number of Queries you want to perform on the same dataset. And most of the time, we sort the dataset before performing search operations. Because, in practical scenario, it works efficiently though it ~~seems~~ might seem counterintuitive because.

$$O(N \log N) + O(\log N) > O(N)$$

But it's true ~~eh~~ only when the data set is small and we're doing less number of Queries. ~~For~~

For instance, $N = 10^6$ and you have 10^3 search queries. Then, the total time complexity for a linear search would be -

$$10^3 \times O(10^6) = O(10^9)$$

In the contrary, if you sort the array first and then do your queries through binary search -

$$O(10^6 \cdot \log 10^6) + O(\log 10^6) \times 10^3$$

$$= O(6 \times 10^6) + 6\,(10^4)$$

$$\approx O(10^7)$$

See, though it might amaze you, but the thing is that, sorting is a one time cost and after that ~~every~~ search becomes efficient for each subsequent search.

Therefore I could reach the conclusion that -

- For a single search over a small data set, a linear search may be better.
- For a large data set or for multiple queries I would sort the array first and then perform binary search instead of just performing a linear search.

(b)

Two things to notice:-

(i) I need to sort a very big data set

(ii) The memory available to me can barely accommodate the data.

Now i've two options – Merge Sort and Quick sort. I would chose the randomized version of the Quick sort. Why randomized version of the Quick sort? Because, with the randomized version of the Quick sort, hitting the worst case is almost 0. Hence, i will always hit the average case. The time complexity will be $O(\log n)$ in this case which is almost constant for large data set. So, i've constant space complexity with $O(n \log n)$ time complexity. On the other hand merge sort has a space complexity of $O(n)$ with worst-case time complexity $O(n \log n)$. So, yeah that is why I would choose Quick sort (Randomized version).

(c)

Quick sort will hit the worst case if we don't use a randomized version of it. I mean to answer the question, in any arry where unbalanced partitioning is happening in each step of the recursion then Quick sort will hit the worst case that is $O(n^2)$. For instance, in a sorted array the unrandomized version of Quick sort ~~fail~~ fail to work in $O(N \log N)$ time. Therefore,

arroy = [1, 3, 6, 9, 12, 15, 18, 21]

Solutions of Question -05

(b) $T(n) = 2T(\frac{n}{3}) + n$

If we compare with the form $T(n) = a \cdot T(\frac{n}{b}) + cn^k$ where $a \geq 1$ and $b > 1$ it matches. Hence, let me apply Master's Theorem –

$a = 2, \quad b = 3, \quad c = 1, \quad k = 1$

Here

$2 < 3^1$ means $a < b^k$.

Hence, $T(n) = O(n^k) = O(n)$.

(Answer)

(d)

$T(n) = 2T(n/4) + n^2$

As the given format matches with the format of Master's Theorem, let me apply it.

Here, $a = 2; \quad b = 4; \quad c = 1; \quad k = 2$

$2 < 4^2$ means $a < b^k$

$\therefore T(n) = O(n^k) = O(n^2)$

(Answer)

(c) $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{5}\right) + n$

$= T\left(\frac{n}{2}\right) + T_1(n) \quad \left[\because T_1(n) = T\left(\frac{n}{5}\right) + n\right]$

$T_1(n)$ matches with the form of Master's Theorem.

Here, $a = 1$; $b = 5$; $c = 1$; $k = 1$.

$1 < 5^1$ means $a < b^k$

$\therefore T_1(n) = O(n^k) = O(n^1) = O(n)$

Hence, $T(n) = T\left(\frac{n}{2}\right) + O(n)$

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

Now again we can apply masters theorem. Here,

$a = 1$; $b = 2$; $c = 1$; $k = 1$

$1 < 2^1$ means $a < b^k$.

Therefore, $T(n) = O(n)$

(Answer)

(a) Given, $T(n) = 2T(n/2) + \frac{1}{n}$

$= 2^2 T\left(\frac{n}{2^2}\right) + \frac{2^0}{n}$

$= 2\left[2T\left(\frac{n}{2^2}\right) + \frac{2}{n}\right] + \frac{2^0}{n}$

$= 2^2 T\left(\frac{n}{2^2}\right) + \frac{2^2}{n} + \frac{2^0}{n}$

$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{2^2}{n}\right] + \frac{2^2}{n} + \frac{2^0}{n}$

$= 2^3 T\left(\frac{n}{2^3}\right) + \frac{2^4}{n} + \frac{2^2}{n} + \frac{2^0}{n}$

⋮

$= 2^k \cdot T\left(\frac{n}{2^k}\right) + \frac{2^0}{n} + \frac{2^2}{n} + \frac{2^4}{n} + \cdots\cdots + \frac{2^m}{n}$

$= 2^k \cdot T\left(\frac{n}{2^k}\right) + \frac{1}{n}\left[2^0 + 2^2 + 2^4 + \cdots\cdots + 2^m\right]$

$= 2^k \cdot T\left(\frac{n}{2^k}\right) + \frac{1}{n} \times \frac{2^0 \cdot (2^{2k} - 1)}{2^2 - 1}$

Assuming,

$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$

$\therefore T(n) = n \cdot T(1) + \frac{1}{n} \times \frac{2^{2\log_2 n} - 1}{3}$

$= n \cdot 1 + \frac{1}{3n} \cdot (n^2 - 1)$ [Assuming $T(1) = 1$]

$= n + \frac{n}{3} - \frac{1}{3n}$

$= O(n)$

$\therefore T(n) = O(n)$ (Answer)

Solutions of Question - 06

(a)

For 2 reasons –

(i) The array is sorted hence using the first element as pivot it will do unbalanced partitioning in each level of recursion.

(ii) As a result, it will hit the worst case time complexity that is $O(n^2)$.

That's why the teacher has said the right thing that using Quick sort won't be efficient.

(b)

As I'm using Quick sort (Unrandomized Version) as my approach the recurrence relation I will get is:-

~~$T(n) = T(n-1) + n$~~
~~$= T(n-2) + \ldots$~~

$$T(n) = \begin{cases} T(n-1) + n & \text{if } n \neq 1 \\ 1 & \text{if } n = 1. \end{cases}$$

Let's solve this recurrence relation,

$$T(n) = T(n-1) + n$$
$$= T(n-2) + (n-1) + n$$
$$= T(n-3) + (n-2) + (n-1) + n$$
$$\vdots$$
$$= T(n-k) + n + (n-1) + (n-2) + \cdots + (n-(k-1))$$

Let, $n - k = 1 \Rightarrow k = n - 1$

$$\therefore T(n) = T(1) + n + (n-1) + (n-2) + \cdots + 1$$
$$= 1 + \frac{n(n-1)}{2} = 1 + \frac{n^2 - n}{2} = 1 + \frac{n^2}{2} - \frac{n}{2}$$

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

(Answer)