CSE224 - Assignment 01

2210593

1

a. $O(\log_6(n) \cdot \log_4(n) \cdot \log_3(n))$

(1st loop / outer loop) (2nd loop / middle loop) (3rd loop / innermost loop)

b. 1st Part

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

(outer loop) (inner loop)

2nd Part

$O(\log_6(n/2) * \log_4(n) * \log_3(n))$

∴ total time complexity = $O(n^2) + O(\log_6(n) * \log_4(n) * \log_3(n))$

C. "The running time of Algorithm A is at least $O(n^4)$" is meaningless. Because Big-O notation describes an upper bound on the time complexity of an algorithm. And we see that for algorithm A, we got the time complexity is $O(\log_6(n) * \log_4(n) * \log_3(n))$. It mean for algorith A, time complexity will be atmost $\log_6(n) * \log_4(n) * \log_3$ But ~~or~~ the statement ~~they~~ tries to say that its time & complexity "at least $O(n^4)$". & That's why, the statement is meaningless in that case.

$O(\log_6(n) * \log_4(n) * \log_3(n))$.

(2)

(Ans)

a. for id in range (0, totalNumOfUsers):
    if users[id].username == searchUsername:
        return users[id]

return "User not found".

⇒ Linear Search Algorithm.

b. Here I will use Count sort algorithm for getting linear time complexity.

* Find the maximum value from the posts list..
* Create an array with length = maximumValue+1.
  (Count array) (Initially all the elements of this array will be 0.
* iterate on the posts list.
  → for each posts list value, i will increment by 1 on Count array of "each posts list value" index.
* iterate the Count array & make a new list.
  → adding their adjacent value.
* Create a sorted list.
  → Iterate on the new list. For each non-zero value, add the index to the sorted list as many times as the count.

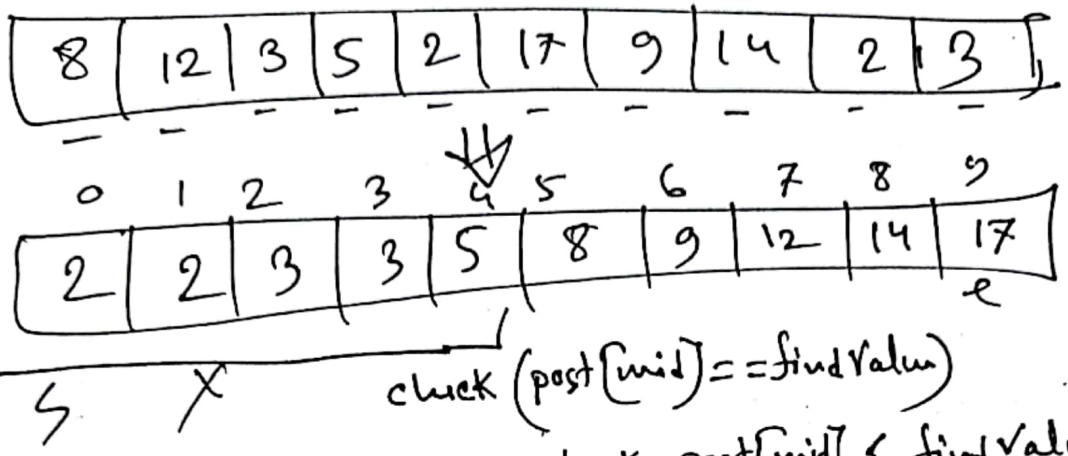
c. Yes, the entire server crashed after running out of memory. Because in Count Sort algorithm, we create a lot of local variables. So, the number of local variable creation in the code increases, as the number of user increases.

d. For nlogn time complexity, I will use Merge Sort algorithm. And the best part of using this algorithm is, its time complexity is always same in the best, worst and average case. Because it follows Divide and Conquer approach.

* find mid
* create 1st part of index (0, mid) & 2nd part of index (mid, last). } do it recursively untill we go the base case (means, one element).
* Then compare them. And Sort it.

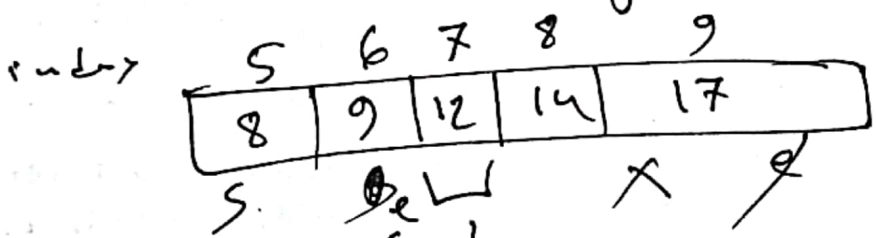
e. (*) Firstly I will use Merge Sor for sorting Post count list with their name & username.

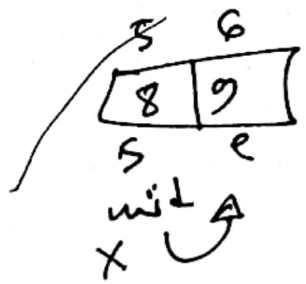
(*) then, use Binary Search algorithm.

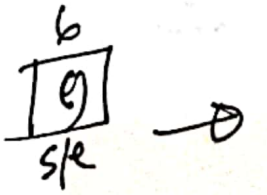


Then check post[mid] < findValue:
go on the right side.

Then check post[mid] > findValue:
go on the left side.







→ found.

(3)

a.

```
while (s <= e):
    mid = (s + e) // 2
    check findValue == arr[mid]
        return mid
    check findValue > arr[mid]:
        s = mid + 1
    check findValue < arr[mid]:
        e = mid - 1
return -1.
```

b. ~~same as a~~.

```
position = -1
while low <= high:
    mid = (high + low) // 2
    if arr[mid] < findValue:
        low = mid + 1
    elif arr[mid] > findValue:
        high = mid - 1
    else:
        position = mid
        high = mid - 1
return position
```

c. s, e, firstIndex = 0, length-1, -1.

```
while (s<=e):
        mid = (s+e)//2
        check findValue == arr[mid]
                firstIndex = mid
                e = mid - 1
        check findValue < arr[mid]
                e = mid - 1
        check findValue > arr[mid]
                s = mid+1
If firstIndex == -1:
        return (-1, 0)
count = 1
i = firstIndex + 1
while i < len(arr) and arr[i] == findValue:
        count += 1
        i += 1
return (firstIndex, count).
```

Time Complexity: O(log

d) low = 0
high = length - 1

while low < high:
    mid = (low + high) // 2
    if arr[mid] > arr[mid+1] then
        low = mid + 1
    else:
        high = mid
return arr[low].

(4)

No, the algorithm will not able to find the search value T = 2.

Reasons :-

i) First of all, the given list is not sorted. And for performing Binary Search algorithm, it is necessary to sort the list.

ii) We see that on the Binary Search algorithm, there mid value (m = floor ((L+R) / 2)) is floor. But we know that mid value must be in the Integer form.

(5)

i) use divide & Conquere approach.

```
low = 0
high = length - 1
while low < high :
        mid = (low + high) // 2
        if (arr[mid] < arr[mid+1]) then
                low = mid+1
                # low will go after the mid.
        else :
                high = mid
return arr[low].
```

ii) Time Complexity : $O(\log_2 n)$.

6

a) Yes, I would sort the array first, then perform binary search. Because in this way, the time complexity will be $O(n \log n + K \log n)$

K = num of searches.

which can be significantly faster than performing K linear searches, and each linear search performs, it take $O(n)$ time complexity.

b) The main idea is to find the minimum element in the array, and shift all elements by the absolute ~~val~~ value of this minimum element.

c) In the given list, we see that there is some floating numbers. And we know that, count sort is not directly applicable ~~to~~ to floating point numbers or negative numbers. That's why at first I will multiply 10^m (m = num of fractional part) with each element of the array. Then for negative number, I will find minimum value first, And the shift all elements by the absolute value. Then I will perform Count sort

d. For limited memory, QuickSort would be a better choice than Merge Sort.

→ Space Complexity :- QuickSort is an in-place sorting algorithm, which means it doesn't require any extra storage, whereas Merge Sort requires extra space for the merging process, hence not suitable for large datasets if the system has limited memory.

→ Worst case Scenario :- The worst case scenario of QuickSort ($O(n^2)$) can be avoided by using a technique called "randomized QuickSort", which ensures that the worst case doesn't occure for all cases of data.

e. The QuickSort algorithm performs poorly ($O(n^2)$ time complexity) when the input array is already sorted or in reverse order.

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

arr = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]