CSE 321 Introduction to Algorithm Design
HW2

1) 6-5-3-11-7-5-2

first step, start with L[2]

→ $\underbrace{5-6}_{\text{sorted}}$-3-11-7-5-2 ⇒ Since 5 is smaller than 6, we shift 6 to right and put 5 to old place of 6.

→ $\underbrace{3-5-6}_{\text{sorted}}$-11-7-5-2 ⇒ In this step, our current element is 3. We started to compare right to left. first we compare with 6, 3<6 so put 6 to one place right, compare with 5, 3<5 so put 5 to one right. Put 3 to current position.

→ $\underbrace{3-5-6-11}_{\text{sorted}}$-7-5-2 ⇒ We apply the same process to 11. In this time 11>6 so we didn't change anything.

→ $\underbrace{3-5-6-7-11}_{\text{sorted}}$-5-2 ⇒ Current element is 7. So, 7>11, put 11 one index right. 7>6, put 7 in place of current position+1 (old place of 11)

→ $\underbrace{3-5-5-6-7-11}_{\text{sorted}}$-2 ⇒ Current element is 5. 5<11, shift 11 to right. 5<7, shift 7 to right, 5<6 shift 6 to right. 5=5 place 5 to current position+1 (old place of 6)

→ 2-3-5-5-6-7-11 ⇒ Current element is 2. 2<11, so shift 11 to right. 2<7, so shift 7 to right. 2<6, so shift 6 to right. 2<5, so shift 5 to right. 2<5, so shift 5 to right. 2<3 so shift 3 to right. Place 2 to current index.

2) a) In this part the basic operation is printf. There are 2 nested for loop and there is no recursive part. When we look at outer loop it iterates n times but in inner loop we have a "break;" statement under the printf;. Because of that break statement it iterates only 1 times. So complexity is O(n).

Scanned with CamScanner

b) In this part the basic operation is increment operation. There are 3 nested loop and no recursive call. When we look at first loop (outer loop);

→ for(int i=n/3; i≤n; i++)

$n/3+1, n/3+2 \dots n \Rightarrow O(n-\frac{n}{3}) = O(\frac{2n}{3}) = O(n)$

→ for second loop;

for(int j=1; j+n/3≤n; j++)

Nothing is different from first loop

$n/3+1, n/3+2 \dots n \Rightarrow O(n-n/3) = O(\frac{2n}{3}) = O(n)$

→ For third loop

for(int k=1; k≤n; k=k*3) $\Rightarrow O(\log(n))$

$1, 3, 9 \dots n \Rightarrow$ increases with 3

so $O(n^2 \log n)$

3) Real python code is provided in the homework file.

PSEUDOCODE

```
procedure mergeSort(L[0:n]):
        if n<1:
            return 0
    left = L[0:n/2]
    right = L[n/2+1 : n]
    mergeSort (left)
    mergeSort (right)
    merge(left, right)
end
```

```
procedure find_pairs (my_array, number):
    mergeSort (my_array)
    i=0
    j=len(my_arr)-1
    while i<j:
        result = my_arr[i]*my_arr[j]
        if result < number: #increase result
            i=i+1
        elif result > number: #decrease result
            j=j-1
        else
            print ("numbers are: "+str(my_arr[i])+
                                  str(my_arr[j]))
            i=i+1
            j=j+1
        endif
    endwhile
end
```

⇒ Complexity analysis for merge sort;

$$T(n) = \underbrace{2T(n/2)}_{\text{dividing part}} + \underbrace{n}_{\text{merging part}}$$

From master theorem;

$a=2$
$b=2$ if $a = b^d$ $\Theta(n^d \lg n)$
$d=1$ $2 = 2^1$

so complexity of mergesort is $\Theta(n\lg n)$

⇒ Complexity analysis for find-pairs procedure;

There is one mergesort, so ⇒ $\Theta(n\lg n)$

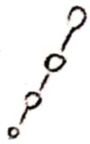
Below that we have a loop that has 2 iterating variable, one is from start and other is from end. Since they stop when they reach each others value they iterates only one time. So the complexity is $\Theta(n)$

$$\Theta(n + n\lg n) \Rightarrow \Theta(n\lg n)$$

4) We can solve this problem in three way;

→ We can add the whole tree from root of tree 1 to tree 2.
→ We can add one by one from tree 1 to tree 2.
→ We can store trees into 2 different arrays (as sorted), and we can merge these two array. Finally we create new BST from merged array

First case;

If tree is very unbalanced (all the elements are added to the left child of the all nodes like this ), in this case we have single linked list.

In this case finding a proper position in tree2 is $O(n)$. Because of linked list shape.

Adding tree 1 to tree 2 is $O(1)$. So the complexity is $O(n)$.

But in normal condition, finding proper position in tree2 for root of tree 1 is $O(\log n)$. Adding tree is $O(1)$. so complexity is $O(\log n)$.

Second case;

If tree2 is in list shape as 

and when we take elements one by one;

$O(n) \rightarrow$ taking every element one by one from tree 1;

$O(n) \rightarrow$ Adding new element to tree2.

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

But in normal condition we will take elements from tree2 one by one ($O(n)$) and add to the tree 1 (which has $O(\log n)$)

$$O(\log n) \rightarrow O(\log n+1) \rightarrow O(\log n+2) \dots \rightarrow \dots O(\log 2n) \Rightarrow O(\log n)$$

After adding first elem. After adding third elem. ......

$$O(\log(n * \log n)) \Rightarrow O(\log(n \log n))$$

Third case;

We can store two tree in a 2 different array. If we apply inorder traverse in both trees while adding to array, we obtain sorted arrays. Applying this procedure is $O(n)$ for each tree. After that we merge this two array into a 1 array. This procedure takes $O(n+n)$ complexity.

So total complexity is $O(n)$.

5) procedure check-subarr (arr, subarr):

```
    for i in arr: #add arr in a dictionary
        my dict [i] = 1
    endfor
    for i in subarr:
        if my-dict.get(i) == None:
            return -1
        endif
    endfor
    return 0
end
```

Basic operation is assignment.

Adding array 1 (big array) to dictionary is $O(n)$ complexity. Checking array2 in dictionary is $O(m)$ complexity. (n = size of array 1, m = size of array 2)

(We know python dictionary get function or hashtable get functions has $O(1)$ complexity)

Best case → First element of small array is not in the dictionary.

$O(n+1) = O(n)$

Worst case ⇒ Array 1 contains Array 2 or array 1 contains all elements of array 2 except last one;

$O(n+m)$

Yusuf Can Kan

161044007