Esra Eryılmaz
171044046

- CSE 321 -

Homework 2

23.11.2020

(1) Apply insertion sort to the following sample array:

Array = {6, 5, 3, 11, 7, 5, 2}

Show each step of the algorithm. At each step, you are required to not only create the array with its new order, but also explain why you created that sequence.

0	1	2	3	4	5	6
6	5	3	11	7	5	2

already sorted

A[0] is already sorted.
Take A[1] which is '5' and compare with already sorted elements 5<6 so swap

5	6	3	11	7	5	2

already sorted

Take first unsorted element which is '3'
Compare with 6; 3<6 so swap.

5	3	6	11	7	5	2

But not enough, compare 3 with another already sorted '5'. 3<5 so swap

3	5	6	11	7	5	2

already sorted

3 is at the beginning of the array. So go to the unsorted element '11'.
Compare with 6 ; 11 > 6 so not swap

3	5	6	11	7	5	2

already sorted

Take first unsorted element which is '7'.
Compare with 11; 7 < 11 so swap.

3	5	6	7	11	5	2

already sorted

Compare 7 with 6. 7>6 so not swap.
We are done with 7.

3	5	6	7	11	5	2

already sorted

Take first unsorted element which is '5'.
Compare with 11 ; 5 < 11 so swap

3	5	6	7	5	11	2

Compare 5 with 7.
5<7 so swap.

3	5	6	5	7	11	2

compare 5 with 6.
5<6 so swap.

3	5	5	6	7	11	2

already sorted

Compare 5 with other 5.
$5 \not< 5$ so not swap.

3	5	5	6	7	11	2

already sorted

Take first unsorted element which is 2.
Compare 2 with already sorted elements.
Compare 2 with 11. 2<11 so swap.

3	5	5	6	7	2	11

compare 2 with 7.
2<7 so swap.

3	5	5	6	2	7	11

Compare 2 with 6.
2<6 so swap.

3	5	5	2	6	7	11

Compare 2 with 5.
2<5 so swap.

3	5	2	5	6	7	11

Compare 2 with other 5.
2<5 so swap

3	2	5	5	6	7	11

Compare 2 with 3.
2<3 so swap.

2	3	5	5	6	7	11

already sorted.

2 is at the beginning of the array.

Array is sorted!

Steps of the insertion sort algorithm:

1) Take array of unsorted elements.
2) mark the first element as a sorted and rest of unsorted.
3) Repeat step 4-5-6 until the unsorted elements finished.
4) Take first unsorted element
5) Compare that element with sorted elements. Swap this element until it arrives at the correct sorted position.
6) move the marker one position ahead.
7) Stop.

---

(2) What is the time complexity of the following programs? Explain in detail by providing your reasoning.

a)
```
function (int n){
    if (n==1)
        return;                      ] O(1) → Best case (constent time)
    for(int i=1; i<=n; i++){
        for (int j=1 ; j<=n ; j++){        O(1)
            printf("*");                  This for loop works
            break;                        only 1 times in each
        }                                 time. Because there
    }                                     is a break statement
}                                         inside. It breaks the cyle.
```

O(n)

In above code "*" will print n times. Because inner for loop works only 1 time when its called than break it. Therefore, the existence of inner loop does not change anything.
So, time complexity of above code is O(n).

b)
```
void function (int n) {
    int count = 0;
    for (int i = n/3 ; i <= n ; i++)              ] n times
        for (int j = 1 ; j+n/3 <= n ; j++)        ] n times
            for (int k = 1 ; k <= n ; k = k*3)    ] log3 n times
                count ++; ) O(1)
}
```

\# The first for loop is running n times.

$i = 1, 2, 3, \ldots \frac{n}{3} \Longrightarrow \frac{n}{3}$ times

$O(\frac{n}{3}) \in O(n) \rightarrow$ first loop.

\# The second for loop is running n times. $j + \frac{n}{3} <= n$

$j = 1, 2, \ldots \frac{2n}{3} \Longrightarrow \frac{2n}{3}$ times $\quad j <= \frac{3n-n}{3}$

$O(\frac{2n}{3}) \in O(n) \rightarrow$ second loop

\# The third for loop is running $\log_3 n$ times.
Loop body runs $O(1)$ (constant) time.

k values increases 3^t

1
$1 \times 3 = 3$
$3 \times 3 = 3^2$
$3^2 \times 3 = 3^3$
$\vdots$
3^t

$O(\log_3 n) \in O(\log n)$] third loop

$3^t = n$

$\boxed{t = \log_3 n}$

So, time complexity of above code is $O(n \cdot n \cdot \log_3 n)$

$= O(n^2 \log n)$ //

(3) We have an unordered array in which we are looking for pairs whose multiplication yields the desired numbers. For example, let our array be {1, 2, 3, 6, 5, 4} and let the desired number be 6. In this case, our pairs will be (1,6) and (2,3). Provide an algorithm that solves this problem with time complexity O(n(logn)). Express your algorithm as pseudocode (write actual code using Python) and prove that its complexity is O(n(logn)).

Pseudocode

```
QuickSort(A, low, high)
    if (low < high)
        pivot = Partition(A, low, high)
        QuickSort(A, low, pivot-1)
        QuickSort(A, pivot+1, high)

Partition(A, low, high)
    pivot = A[low]
    left = low
    for i=low+1 to high
        if (A[i] < pivot) then
            swap(A[i], A[left])
            left = left + 1
    swap(pivot, A[left])
    return(left)
```

This is the pseudocode for QuickSort.
We know that QuickSort Algorithm average time complexity is O(nlogn)

part3.py → python code

```
Find Product Pair (A, size, numb)
    begin = 0
    end = size-1
    while begin < end
        product = A[begin]* A[end]
        if (product == numb)
            print (A[begin], A[end])
O(n)    if (product < numb)
            begin++
        else
            end--
```

After sorting, this function takes O(n) time to execute.

↓

- First of all; we have an unordered array, we should sort this array increasingly. We can choose any sorting algorithm which time complexity is $O(n\log n)$
- I choose QuickSort also I can choose MergeSort or HeapSort as well.
- After sorting I write FindProductPair() function to find pairs whose multiplication yields the desired number.

In FindProductPair():

Using two integers "begin" and "end" to index the current pair of numbers. Initialize begin = 0 and end = size - 1.
Then compare the product of A[begin] and A[end] with desired number:

If it is equal, we found the pairs.
If it is lower, we must find a bigger product, so increase begin with 1
If it is higher, we must find a smaller product, so decrease end with 1.

- FindProductPair() function time complexity is $O(n)$.
Because it has one while loop inside. Statements inside the while loop have constant time complexity. It just increments, decrements ...

So one loop $\rightarrow O(n)$

- We have two parts of the algorithm:

1) Sorting $\longrightarrow O(n\log n)$
2) Finding pairs in the sorted array $\longrightarrow O(n)$

$$+$$

$$O(n\log n) + O(n)$$

(nlogn have control on n) $= O(n\log n)$

4) You are given a binary search tree (BST) with n nodes. You are required to merge this tree with another n-node BST. What is the time complexity of this process? Explain in detail by providing your reasoning.

- We have 2 BST with n nodes and we want to merge them into one BST.

1) First of all, perform the inorder traversal for both BST; it creates two sorted arrays.

→ In order to analyse the time complexity of inorder traversal, we have to think how many nodes visited. If tree has n nodes, then each node is visited only once. Because of that the complexity is $O(n)$

2) Merge two arrays in one array.

→ We traverse both arrays and choose the smaller element. In the end, we copy the rest of the elements from two arrays. So the time complexity becomes $O(n)$

3) Convert sorted array to BST

→ Every node will be created so there will be n calls. Each call have $O(1)$ runtime. Time complexity will be $O(n)$.

- So all three steps takes linear time. At the beginning we have 2 BST with n nodes. So we can say that every steps take $\underbrace{O(n+n)}_{O(2n)}$ time to execute.

$$O(2n) + O(2n) + O(2n) = 3O(2n) \in O(n)$$

5) Suppose that you are given two arrays, where one of them is larger than the other one. Propose a linear time algorithm that finds, if exists, the elements of the small array in the big array. Express your algorithm using pseudocode and calculate the worst case complexity of the algorithm.

Pseudocode

larger array

```
procedure FindSubset(A[], B[], sizeA, sizeB)
    hashset = set()
    for(i=0 to sizeA-1)
        hashset.add(A[i]))        O(1)    O(n)
    for(i=0 to sizeB-1)                   O(n)
        if B[i] in hashset:       O(1)
            continue
        else:
            return False          O(1)
    return True
```

$O(n) + O(n) = 2O(n) \in O(n)$

Hash Set	
add() →	O(1)
search() →	O(1)

- To reach linear time complexity, I used HashSet.
- First of all, add all elements of A[] (which is larger array) to the hashset. Adding takes constant time in hashset but we iterate all elements of the A[] so it takes O(n) time.
- After that we iterate over B[] and search for each element of B[] in the hashset. If the element is found than continue, if not than return false. Search takes constant time, so it takes O(n).
- At the end, if there is no returning false than it means that larger array contains the element of the small array, so return true.

$O(n) + O(n) = 2O(n) \in O(n)$

↓

worst case complexity of the algorithm:

- We can write the algorithm that takes quadratic time. $O(n^2)$
  we can use two loops.
  The outer loop takes all the elements of the smaller array.
  The inner loop searches if larger array contains smaller array elements linearly.

$O(\text{sizeA} \cdot \text{sizeB})$ ←

```
for i=0 to sizeB
    for J=0 to sizeA
        ------ ) constant
```

So we have two loops, one of them traverses sizeB times, other one traverses sizeA times.

$$O(\text{sizeA} \cdot \text{sizeB}) \in O(n^2)$$