Muhammed ÖZKAN 151044084

CSE-321 Introduction to Algorithm Design,

Fall 2020 Homework 2

1)

6	5	3	11	7	5	2

Starting Insertion Sort.

Highlighted red records to the left are always sorted. We begin with the record in position 0 in the sorted portion, and we will be moving the record in position 1 (in gray) to the left until it is sorted.

6	5	3	11	7	5	2

Processing record in position 1
Move the gray record to the left until it reaches the correct position.

5	6	3	11	7	5	2

5 < 6 Swap.

5	6	3	11	7	5	2

Processing record in position 2
Move the gray record to the left until it reaches the correct position.

5	3	6	11	7	5	2

3 < 6 Swap.

3	5	6	11	7	5	2

3 < 5 Swap.

3	5	6	11	7	5	2

Processing record in position 3
Move the gray record to the left until it reaches the correct position.

3	5	6	11	7	5	2

Processing record in position 4
Move the gray record to the left until it reaches the correct position.

3	5	6	7	11	5	2

7 < 11Swap.

3	5	6	7	11	5	2

Processing record in position 5
Move the gray record to the left until it reaches the correct position.

3	5	6	7	5	11	2

5 < 11 Swap.

3	5	6	5	7	11	2	5 < 7 Swap.
3	5	5	6	7	11	2	5 < 6 Swap.
3	5	5	6	7	11	2	Processing record in position 6 Move the gray record to the left until it reaches the correct position.
3	5	5	6	7	2	11	2 < 11 Swap.
3	5	5	6	2	7	11	2 < 7 Swap.
3	5	5	2	6	7	11	2 < 6 Swap.
3	5	2	5	6	7	11	2 < 5 Swap.
3	2	5	5	6	7	11	2 < 5 Swap.
2	3	5	5	6	7	11	2 < 3 Swap.
2	3	5	5	6	7	11	Array is finish. Done sorting.

2) a)

function(int n){
 	
  if (n==1)
  return;
 	
  for (int i=1; i<=n; i++){
  for (int j=1; j<=n; j++){
  printf("*");
  break;
  }
  }
}
	where the function n = 1 has O (1) = O (n) complexity.
	
	In this part, although there are 2 nested loops, the word "break" in the inner loop will cause the inner loop to run only 1 time with any n values. Therefore, the complexity of this part is O (n).
	
	Consequently, the time complexity of this function is **O (n).**

b)

void function(int n){
 	
int count = 0;
 	
for (int i=n/3; i<=n; i++)
  for (int j=1; j+n/3<=n; j++)
  for (int k=1; k<=n; k = k * 3)
  count++;
}
	First loop will run n/3 times. n/3 to n
	
	Second loop will run n/3 times as j+n/3<=n is the condition. So n/3 is already added to the iterator so only n/3 times loop will run.
	
	Third loop 3^k < n (iteration wise 3^1,3^2,…3^k)
	
	It will take on the series of values 1, 3, 9, 27, 81, 243, ..., 3k. Since k is tripling on each iteration, it takes on successive powers of three.
	
	The loop will stop when k > n. If we let t be some arbitrary iteration of the loop, the value of k on iteration t will be 3^t. The loop stops when 3^t > n, which happens when t > $\log_3$ n. Therefore, the number of iterations is only O(log n), so the total complexity is O(log n).
	
	So time complexity is(n/3)*(n/3)*(logn) so **O(n^2logn)** is the time complexity.

3)

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
       Call merge(arr, l, m, r)
```

The complexity of merge sort is O(nlogn)

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

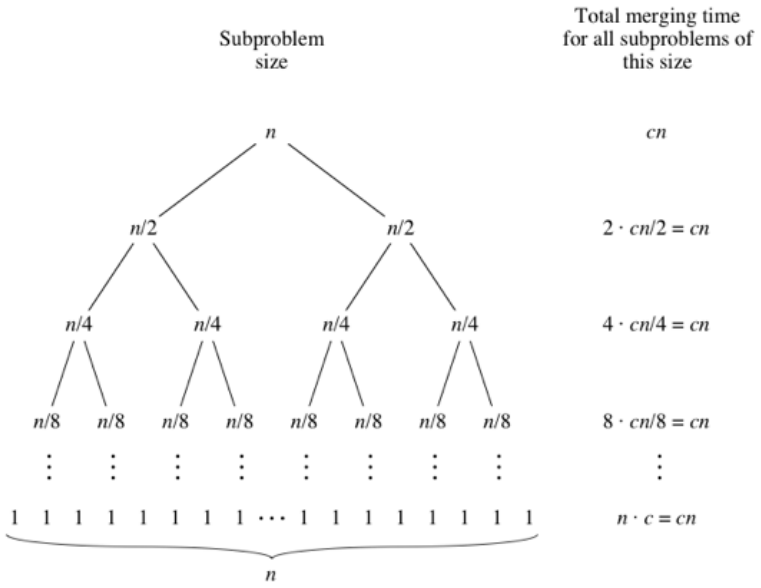
Think of it in terms of 3 steps

-The divide step computes the midpoint of each of the sub-arrays. Each of this step just takes O(1) time.
-The conquer step recursively sorts two subarrays of n/2 (for even n) elements each.
-The merge step merges n elements which takes O(n) time.
Now, for steps 1 and 3 i.e. between O(1) and O(n), O(n) is higher. Let's consider steps 1 and 3 take O(n) time in total. Say it is cn for some constant c.

How many times are these steps executed?

For this, look at the tree below - for each level from top to bottom Level 2 calls merge method on 2 sub-arrays of length n/2 each. The complexity here is 2 * (cn/2) = cn Level 3 calls merge method on 4 sub-arrays of length n/4 each. The complexity here is 4 * (cn/4) = cn and so on ...

Now, the height of this tree is (logn + 1) for a given n. Thus the overall complexity is (logn + 1)*(cn). That is O(nlogn) for the merge sort algorithm.




```
PairedElements(arr[], multiplication)
    low = 0
    high = arr lenght - 1

    loop(low < high)

        if (arr[low] *  arr[high] == multiplication)
            print(arr[low] , arr[high])
        end if

        if (arr[low] * arr[high] > multiplication)
            high = high - 1
        else
            low = low + 1
        end if

    end loop
```

The main working way of our code in this section is as follows. We select two indexes from the beginning and the end on the array we have and if their product is equal to the number we want, we print the numbers on the screen. If the result of multiplication is greater than the desired number, the last index is passed to the previous one. If the multiplication result is less than the desired number, the first index is advanced to the next. The up steps are repeated until there are no more elements in the array.

The time complexity of this part is O (n) since it is simpler than the sorting stage and consists of a single loop.

The time complexity of the two separate pieces of code required to do what is required in the question was explained above. Now it's time to combine them and calculate the total time complexity.

```
arr = [1,2,3,6,5,4]
MergeSort(arr)
 PairedElements(arr, 6)
```

Since we need to sort the given sequence first and then find the pairs, whichever of the two time complexities is greater, the time complexity of our program will be equal to it. Since Merge sort is O (nlogn) and the other algorithm is O (n), our algorithm has **O(nlogn)** time complexity since Merge sort has a greater complexity in this part.

4)

Two methods can be considered for this process. The first of these has a time complexity of **O (m + n)**, while the other can take as long as **O (m*log (m + n-1)).**

Method 1

1) Browse through the items of the first tree one by one and store the traversal in one temp array arr1. This step takes O(m) time.
2) Browse through the items of the second tree one by one and store the traversal in another temp array arr2. This step takes O(n) time.
3) The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.
4) Construct a balanced tree from the merged array. This step takes **O(m+n)** time.

Method 2

Take all the items of the smaller BST one by one and add them to the BST the bigger one. Adding an item to a self-balancing BST takes O(logn) time. Where n is the size of BST. The time complexity of this method is log (n) + log (n + 1)… log (m + n-1). The value of this expression will be between **O (m*logn) and O (m*log (m + n-1)).**

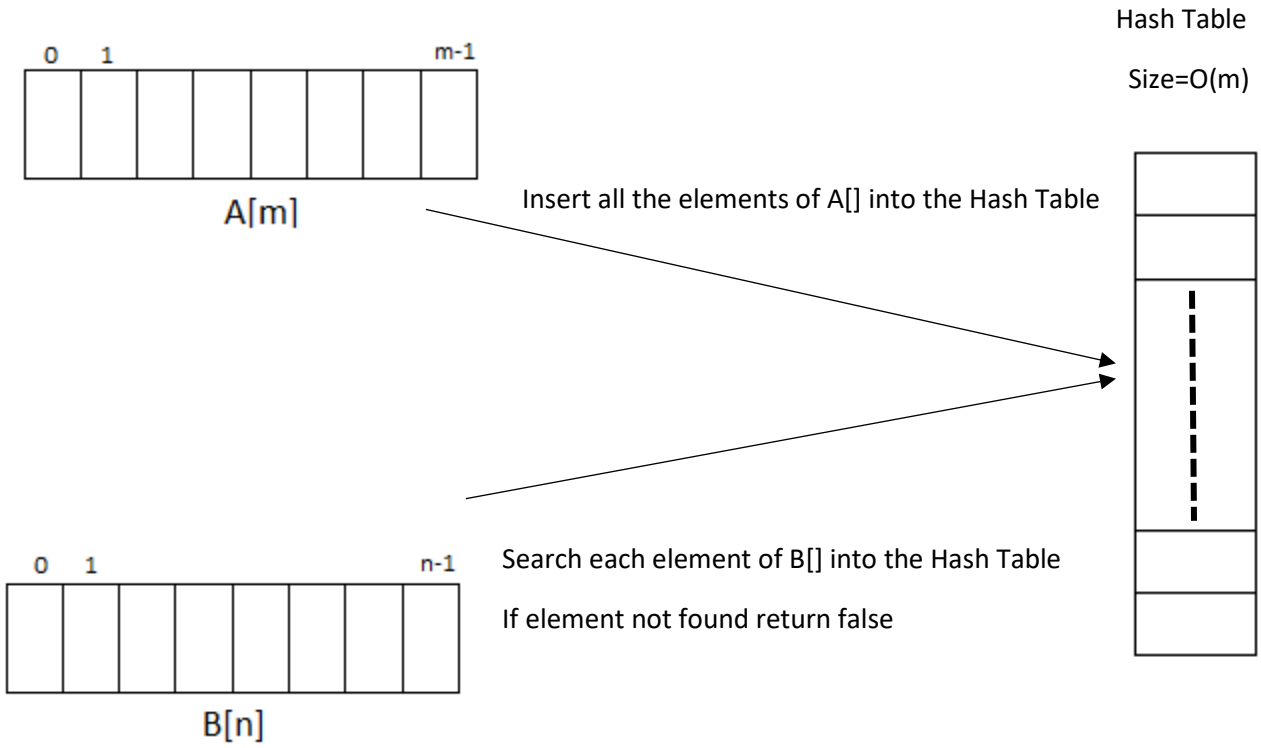
5)

Searching is an important operation in this problem because we need to look for every value of B [] in A []. Thus, we can improve the time complexity by using a less complex data structure for the search process. Hash Table is almost the best data structure that can search and insert efficiently at O (1) average. We will use this because we aim for a linear time complexity result.

```
      int Subset(A[], B[], m, n)
         {
            Take HashTable H of Size m
            for(i = 0 to m-1)
            {
               H.insert(A[i])
            }
            for( i = 0 to n-1)
            {
               if(H.search(B[i]) is false)
                 return -1
            }
         return 1
   }
```

Solution Step

- Take a Hash Table of size O(m)
- Insert all the elements of array A[] into the Hash Table
- Traverse B[] and search for each element of B[] in the Hash Table. If The element is not found then return 0.
- If all elements are found then return 1.



Time complexity of inserting m elements of A[] in hash table + Time complexity of searching n elements of B[] in hash table = m*O(1) + n* O(1) = O(m) + O(n) = **O(m+n)**

There is an important point in this solution. If there are repeating elements in the arrays given in this way, this solution approach will not give correct results. We can solve this problem by keeping the number of repetitions of the elements in the arrays given in addition to this system in a field in the hash table. By checking these repetition numbers in the comparison phase, we can reach a problem-free solution for arrays with repeating elements.