# CSE 321

# INTRODUCTION TO ALGORITHM DESIGN

# HOMEWORK 2

**BERKE BELGİN**

**171044065**

**1)**

**[6, 5, 3, 11, 7, 5, 2], index = 0;**

**[6, 5, 3, 11, 7, 5, 2], index = 1;**

(5 < 6) => set [6, 6, 3, 11, 7, 5, 2]

index == 0 => insert [5, 6, 3, 11, 7, 5, 2]

**[5, 6, 3, 11, 7, 5, 2] , index = 2;**

(3 < 6) => set [5, 6, 6, 11, 7, 5, 2]

(3 < 5) => set [5, 5, 6, 11, 7, 5, 2]

index == 0 => insert [3, 5, 6, 11, 7, 5, 2]

**[3, 5, 6, 11, 7, 5, 2] , index = 3;**

(11 => 6) => insert [3, 5, 6, 11, 7, 5, 2]

**[3, 5, 6, 11, 7, 5, 2] , index = 4;**

(7 < 11) => set [3, 5, 6, 11, 11, 5, 2]

(7 => 6) => insert [3, 5, 6, 7, 11, 5, 2]

**[3, 5, 6, 7, 11, 5, 2] , index = 5;**

(5 < 11) => set [3, 5, 6, 7, 11, 11, 2]

(5 < 7) => set [3, 5, 6, 7, 7, 11, 2]

(5 < 6) => set [3, 5, 6, 6, 7, 11, 2]

(5 >= 5) => insert [3, 5, 5, 6, 7, 11, 2]

**[3, 5, 5, 6, 7, 11, 2] , index = 6;**

(2 < 11) => set [3, 5, 5, 6, 7, 11, 11]

(2 < 7) => set [3, 5, 5, 6, 7, 7, 11]

(2 < 6) => set [3, 5, 5, 6, 6, 7, 11]

(2 < 5) => set [3, 5, 5, 5, 6, 7, 11]

(2 < 5) => set [3, 5, 5, 6, 7, 7, 11]

(2 < 3) => set [3, 3, 5, 6, 7, 7, 11]

index == 0 => insert [2, 3, 5, 6, 7, 7, 11]

**[2, 3, 5, 5, 6, 7, 11]**

**2)**

`function(int n){`

`if (n==1)` **-> Dependent on n. n time(s) when n = 1**

`return;`

`for (int i=1; i<=n; i++){` **-> n times**

`for (int j=1; j<=n; j++){` **-> 1 time since there is a break**

`printf("*");`

`break;`

`}`

`}`

`}`

**So, the time complexity of the above program is Ω(n) = Q(n) = O(n).**

`void function(int n){`

`int count = 0;`

`for (int i=n/3; i<=n; i++)` **-> (n – n*1/3) = n*2/3 times**

`for (int j=1; j+n/3<=n; j = j++)` **-> (n – n*1/3) = n*2/3 times**

`for (int k=1; k<=n; k = k * 3)` **-> x times where $3^x$ = n, $\log_3$n times**

`count++;`

`}`

**So, the time complexity of the above program is:**

**(n*2/3)*(n*2/3)* $\log_3$n -> Ω($n^2$logn) = Q($n^2$logn) = O($n^2$logn).**

**3)**

```
PROCEDURE heapify(arr, n, i)
  largest = i
  l = 2 * i + 1
  r = 2 * i + 2

  IF l < n AND arr[i] < arr[l] THEN
    largest = l
  ENDIF

  IF r < n AND arr[largest] < arr[r] THEN
    largest = r
  ENDIF

  IF largest != i THEN
    arr[i],arr[largest] = arr[largest],arr[i]
    heapify(arr, n, largest)
  ENDIF
END

PROCEDURE heapSort(arr)
  n = len(arr)

  FOR i IN range(n // 2 - 1, -1, -1) DO
    heapify(arr, n, i)
  ENDFOR

  FOR i IN range(n-1, 0, -1) DO
    arr[i], arr[0] = arr[0], arr[i]
    heapify(arr, i, 0)
  ENDFOR
END

PROCEDURE findMultipliers(arr, num)
  indxStart = 0
  indxEnd = len(arr) - 1

  WHILE indxStart != indxEnd DO
    IF arr[indxStart] * arr[indxEnd] == num THEN
      print(arr[indxStart],arr[indxEnd])
      indxStart += 1
    ELSE IF num % arr[indxStart] == 0 THEN
      indxEnd -= 1
    ELSE
      indxStart += 1
    ENDIF
  ENDWHILE
END

arr = [1,2,3,6,5,4]
heapSort(arr)
findMultipliers(arr, 6)
```

**Since Heap Sort has a complexity of nlogn and finding multipliers has a complexity of n because of a single while loop, the program has a complexity of O(nlogn + n) = O(nlogn).**

**4)**

Since regardless of the data structure, best case time complexity of traversing a list of elements is **n** and we can traverse the binary tree to be merged easily in n steps with a simple recursive function. Then for each element, we can insert them into destination BST in **logn** steps since the time complexity of finding an element in BST is logn and inserting a node to a node of a tree is **1.** So, we can say that merging two BSTs by traversing one tree, finding the appropriate position for the every element in the traversed BST in destination BST and then inserting the element has a time complexity of (n*logn*1) which is equal to **O(n*logn).**

**5)**

```
PROCEDURE getSubArray(bigArray[] , smallArray[])
        maxVal = 0
        minVal = 0
        FOR EACH i IN bigArray DO
                IF i > maxVal maxVal = i ENDIF
                IF i < minVal minVal = i ENDIF
        ENDFOR

        tempArraySize = maxVal - maxVal + 1
        tempArray[tempArraySize]

        FOR int i = 0 TO bigArray.length DO
                tempArrayIndex = bigArray[i] - minVal
                tempArray[tempArrayIndex] = bigArray[i]
        ENDFOR

        resultArraySize = smallArray.length
        resultArray[resultArraySize]
        resultArrayIndex = 0

        FOR EACH i IN smallArray DO
                tempArrayIndex = i - minVal
                IF tempArray[tempArrayIndex] is null THEN
                        resultArray[resultArrayIndex] = i
                        resultArrayIndex++
                ENDIF
        ENDFOR

        RETURN resultArray
END
```

Since there were no restrictions in terms of memory space, I assumed we have an infinite source of memory. So, I implemented a Hash like system where it's both best and worst case is Q(n) but may take up as much memory space as the value gap between the smallest and the greatest value of the array.

Since the function has no nested loops and there are 2 loops for big array and 1 loop for small array in a row, we can accept that this functions complexity is:

T(2a + b) => T(3n) => Ω(n) = Q(n) = O(n).