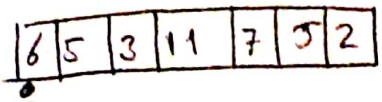
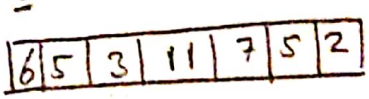
GEBZE TECHNICAL UNIVERSITY

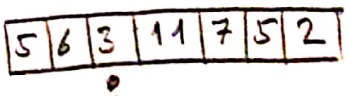
1) Array = {6, 5, 3, 11, 7, 5, 2}

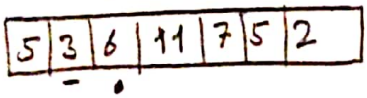
we can think number as a temp and each compalision we move one step backwaird or forword

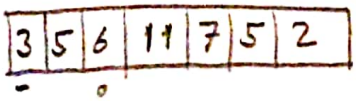

starts from first index and left side of it is empty so it moves next index


5 is smaller than 6 so it needs to swap

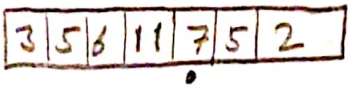

left side of 5 is empty so it stops

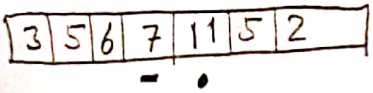

it moves next index of array

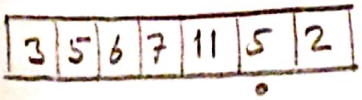

3 is small al than 6 so they swap


3 < 5 so they swap and it stops left side of 3 is empty

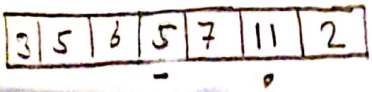

it moves next index and it's bigger than left side


it moves next index

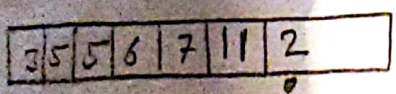

7 < 11 they swap ant 7 > 6 so it moves next index

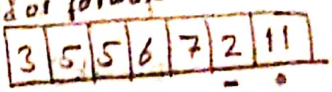

in the next index

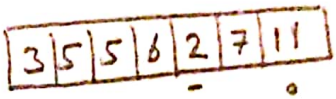

11 > 5 so they swap

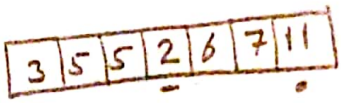

5 < 7 they swap


5 < 6 they swap


5 = 5 so it's tops here and moves the next index


11 > 2 so they swap


7 > 2 they swap


6 > 2 they swap


5 > 2 they swap


5 > 2 they swap


3 > 2 they swap and stops here because left side of it is empty.

The insertion sort starts from first element and each moving the next index in array it's purpose sort left handside of array from start that index and it swaps the element place until it is bigger than left index of it

- -> the index which will be sorted left handside of it

- -> the replacing number it moves until find its place

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;
        }
    }
}
```

n time — This loops works total regaidles number n

→ for the best case we can say that it works $O(n)$ time because even we count it at for loop it iterates 1 times which is n

for the worst case

it runs n.(constant + n)

so that runs $\Theta(n^2)$ time complexity for the worstcase.

we can say that algorithm work: $\Theta(n)$ in general

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++;
```

$n-n/3$ ↓ $\frac{2n}{3}$ time

$j+n/3=n$ ↓ $\frac{2n}{3}$ time

runs logn time

1 time according to outer loop

Since first loop start from $n/3$ until n it iterates $\frac{2n}{3}$ time

Second loop is also like first loop because each time it adds $n/3$ so to become n it iterates $\frac{2n}{3}$

third loop increases 3 by tree so it iterates 3^n more each time

so the total complexity = $\frac{4n^2}{9} \cdot \log_3 n$

it iterates $\frac{4n^2}{9}\log n$ times, and there is not anything to stop loops so the time complexity is $\Theta(n^2 \log n)$

1 → first
9 → second
27 → third
⋮ ⋮

3)

```
def pairs(arr, target):
    sorted(arr)
    i = 0
    J = len(arr) - 1
    while i < J:
        if arr[i] * arr[J] == target:
            print("(" + str(arr[i]) + ", " + str(arr[J]) + ")")
        if arr[i] * arr[J] < target:
            i += 1
        else
            J -= 1
```

n — while takes n times iteration max

Since we are using python first I sorted array by using sorted algorithm then I take first and last elements of array and according to target value I go on array backward or forward. so its easy with sorted arr but still is takes O(n) time complexity but we wrode in python and it uses timsort as sort algorithm which is combination of insertion and merge sort and average case performance of it takes O(nlogn) and while loop also takes O(n) so totally O(n) + O(nlogn) gives O(nlogn) for average and worst case time complexity. That algorithm calls as Timsort.

4) we can do that by placing all tree elements in two array then we can merge them so we get an ordered array and we can create new tree.

1- Place tree elements into array

If we make in order traverse on tree and place them into array we get sorted array and it takes O(n) time. (We do that for each tree)

2- We merge two array into one array.

merging two sorted arrays take O(n) time.

3- Creating a tree from ordered array

It takes O(n) time

Totally we can say O(n) + O(n) + O(n) it gives O(n) time complexity which is the efficient way to merge two bst.

5)

```
isSubset ( shortArr, longArr)

hash = set()
 for  i  to (longArr).length
        hash.insert(longArr[i])
for  i to shortArr.length
     if (shortArr in hash)
          return 0

return 1;
```

Adding elements in to hash table takes constant time if we assume long array has m elements it take O(m) time.

Also searching elements in hashtable takes constant time if we assume short array has n elements it takes O(n) time.

Totally it takes O(m+n) time for the worst case.