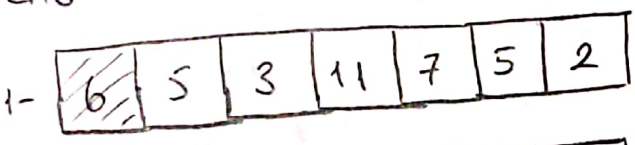
CSE - 321 Introduction to Algorithm Design Homework - 2

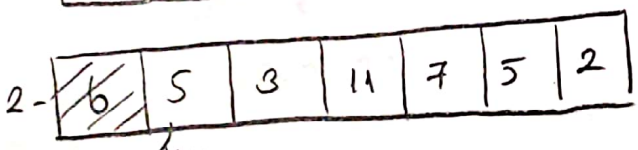
Türker Tercan 171044032

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

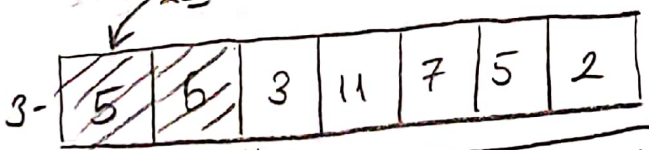
```
procedure InsertionSort (L[1:n])
 end for
   current = L[i]
   position = i-1
   while (position >= 1 and current < L[position] do
      L[position +1] = L[position]
      position =position -1
   end while
   L[position +1] = current
 end for
end
```

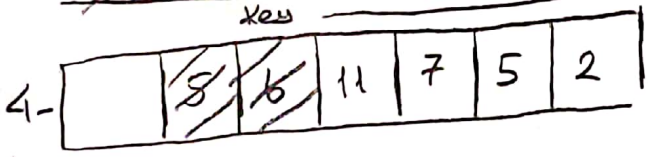


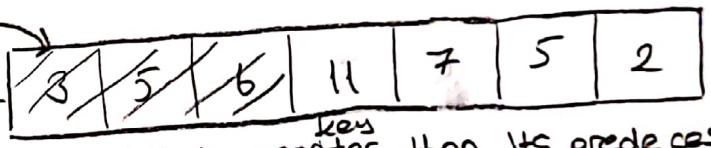
- first elemented counted as sorted.



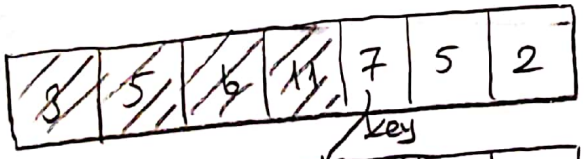
- if key is smaller than its predecessor, Move the grater elements one position up to make space for key elements

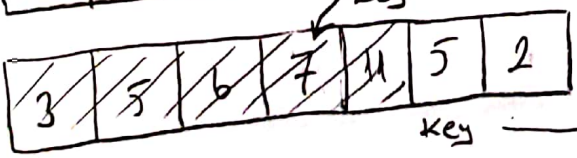


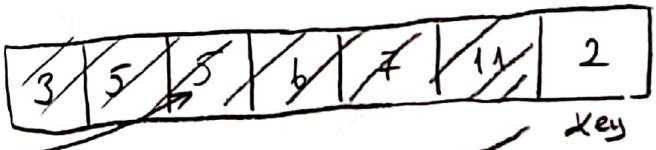


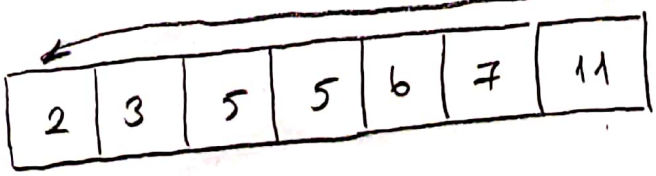


Since 11 is greater than its predecessor We can say that list is sorted up to element 11









List is now sorted

2-

a)

```
function (int n) {
    if (n==1)
        return;
    for (int i=1; i<=n; i++) {
        for (int j=1; j<=n; i++) {
            print ("*");
            break;
        }
    }
}
```

Outer for loop runs n times

Inner for loop runs n.1 times because of break statement

print ("*"); — n

$f(n) = n \in O(n)$

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++;
}
```

for (int i=n/3; i<=n; i++) — Approximately takes O(n)

for (int j=1; j+n/3<=n; j++) — Approximately takes O(n)

for (int k=1; k<=n; k=k*3) — $\log_3 n$

$f(n) \in O(n^2 \log n)$

3-

```
procedure QuickSort(L[low:high])
  if high > low
    call Rearrange(L[low:high], position)
    call QuickSort(L[low:position-1])
    call QuickSort(L[position+1:high])
  end if
end
```

QuickSort

$B(n) \in \Theta(n \log n)$ $W(n) \in \Theta(n^2)$ $A(n) \in (n \log n)$

```
procedure Rearrange(L[low, high], position)
  right = low
  left = high+1
  x = L[low]
  while right < left do
    repeat right++ until L[right] >= x
    repeat left-- until L[left] <= x
    if right < left
      Interchange(L[left], L[right])
    end if
  end while
  position = left
  L[low] = L[position]
  L[position] = x
end
```

```
procedure Binary Search(Sorted Array[low:high], target)
  if high >= low
    mid = (high + low) / 2
    if (arr[mid] == target)
      return mid
    end if
    else if (arr[mid] > target)
      return Binary_Search(Sorted Array[low:mid-1], target)
    end else if
    else
      return Binary_Search(Sorted Array[mid+1: high], target)
    end else
  end if
  else
    return -1
  end else
end
```

Binary Search

$B(n) \in \Theta(1)$

$W(n) \in \Theta(\log n)$

$A(n) \in \Theta(\log n)$

```
procedure pair_search(L[0:n], target)
  QuickSort (L[0:n])
  for i=0 to n do
    index = Binary Search(L[i+1:n-1], target / L[i])
    if (index != -1)
      print(L[i], L[index])
    end if
  end for
end
```

First, I sorted the array with QuickSort algorithm. After sorting we can use Binary Search Traverse the array for every element arr[i]. We need to find for pairs whose multiplication yields in desired numbers.

desired number = L[i] • L[x]

We do binary search for x / L[i] in the right sub-array

QuickSort's average case complexity is $\vartheta(n \log n)$

```
for i =0 to n do
    index = BinarySearch(L[i+1:n-1], target/ L[i]
```

Binary Search's average time complexity is $\vartheta(\log n)$ but it runs n times so its $\vartheta(n \log n)$

$F(n) = n \log n + n \log n \in \vartheta(n \log n)$ and also it is $O(n \log n)$

4-

First, do inorder traversal to small tree and store it to an array. Assume that it's size n. It takes $O(n)$ time. Do the same thing to bigger array and assume that its size is m. This takes also $O(m)$. Merge these two array together. The arrays are already sorted so it takes $O(n+m)$ time. Create another binary search tree, get the middle of the array and make it root and recursively do same for left and right half. And also this step takes $O(n+m)$ time All process is done in $O(n)$ time.

5-

```
procedure find_elements(arr1[], arr2[])
    n = arr1.length
    m = arr2.length
    HashSet<int> hash1 = new HashSet<>()
    HashSet<int> hash2 = new HashSet<>()

    for i=0 to n do
        if (!hash1.contains(arr1[i]))
            hash1.add(arr1[i])
        end if
    end for

    for i=0 to m do
        if (hash1.contains(arr2[i]))
            hash2.add(arr2[i])
        end if
    end for

    for-each i in hash2
        print(i)
    end for
end
```

Since I use HashSet average time complexity of adding and if contains a given element is O(1) time. So all this procedure $B(n) \in O(n)$ $A(n) \in O(n)$

But if too many elements were hashed same key it may take O(n) time so worst case of this algorithm is $O(n^2)$