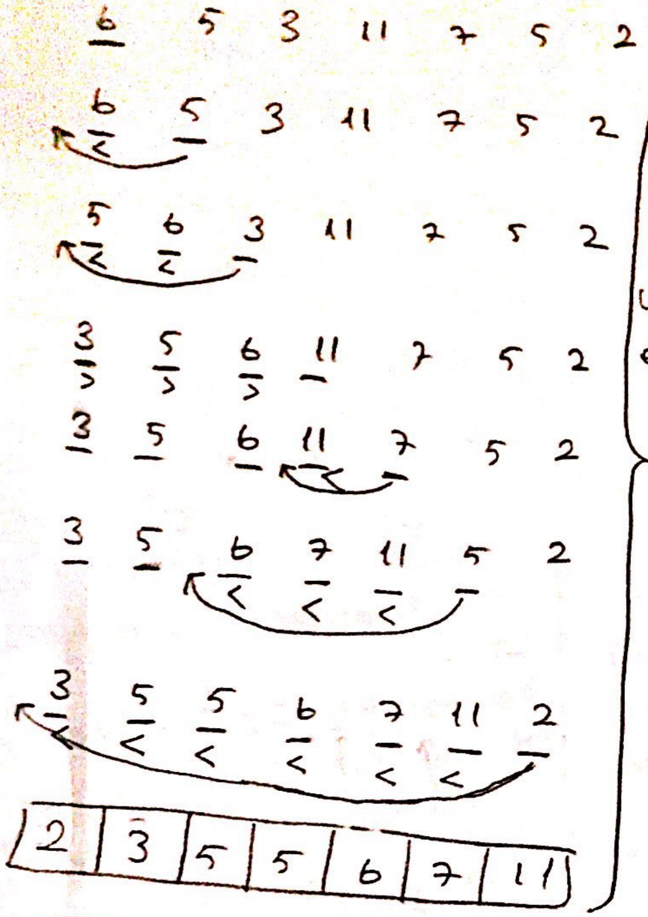
① 6, 5, 3, 11, 7, 5, 2



If the key elements is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

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;
   }
  }
}
```

Best case complexity

When the n is 1, function don't run loop just return. "return" is one process, so the function is $\Omega(1)$

Worst case Complexity

The second loop will run only once because there is "break" in second loop code scope. The first loop will run n times. Because they are nested, $O(1) \times O(n) = O(n)$

=> Best case $\Omega(1)$
worst case $O(n)$

2.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

The loop runs $(n-\frac{n}{3})$ times

Second Loop

The loop runs x times. $\frac{n}{3}$ is constant number because the loop takes this number a top loop.

$(n-\frac{n}{3})=x$

Third Loop

The loop runs $\log_3 n$ times.

Proof

Suppose that third loop runs x times.

1, 3, 9, 27 . . . n

$3^0 \quad 3^1 \quad 3^2 \quad 3^3 \quad . . . \quad 3^x$

$3^x = n \Rightarrow \boxed{x = \log_3 n}$

As these are all loops nested

$(n-\frac{n}{3})\times(n-\frac{n}{3})\times(\log_3 n) \Rightarrow$ function complexity

$(\frac{4n^2}{9}\times\log_3 n) = \frac{4n^2\log_3 n}{9} \Rightarrow \boxed{O(n^2\log n)}$

✱ Function runs $O(n^2\log n)$ stably. Cause of this situation function time complexity is $\boxed{\Theta(n^2\log n}$

```
③ define function merge(arr, l, m, r):
      set num1 to (m-l+1)
      set num2 to (r-m)
      set l-arr to [0]* (num1)
      set r-arr to [0]* (num2)

      for i in rage(0, num1):
         set l-arr[i] to arr[l+i]
     for i in rage(0, num2):
        set r-arr[i] to arr[m+1+i]

    set i to 0
    set j to 0
    set k to l

    while i < num1 and j < num2:

       if l_arr[i] <= r_arr[j]:
          set arr[k] to l_arr[i]
          i+=1

      else :
         set arr[k] to r_arr[j]
         j+=1

     k+=1

   while i < num1:

     set arr[k] to l_arr[i]

     i+=1
     k+=1
```

```
define function mergeSort (arr, l, r):

    if l < r
        set m to (l + (r - l))//2

    mergeSort (arr, l, m)
    mergeSort (arr, m+1, r)
    merge (arr, l, m, r)


define function productPairs_helper(arr, target, pairs):
    mergeSort(arr, 0, len(arr)-1)

    set beg to 0
    set end to len(arr) - 1

    while beg < end :

        set temp_t TO arr[beg]* arr[end]

        if temp_t equals target :
            Pairs.append(arr[beg], arr [end])

        if temp_t < target :
            beg+=1

        else :
            end -=1
```

```
define function productPairs (arr, target, pairs):
    productPairs-helper(arr, target, pairs)

    if len(pairs) equals 0:
        Output("No pairs...")
    else:
        Output("Pairs: " + str(pairs))


set arr to [1,2,3,6,5,4]
set target to 6
set pairs to []
productPairs (arr, target, pairs)
```

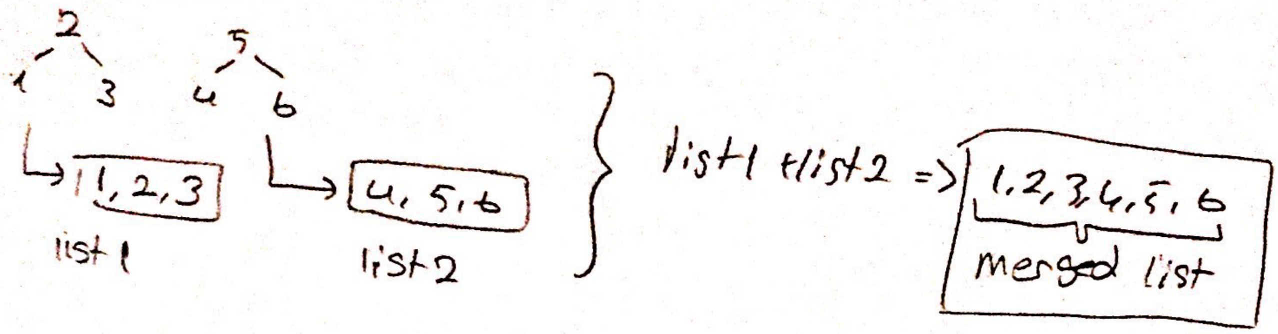
First we need to sort the array from smallest to largest. I used merge sort algorithm in this part because an n logn solution was requested. After sorting the array, we can find pairs in linear time by performing a binary search.

$\boxed{\text{Time Complexity} = \theta(n\log n)}$

4 Do inorder traversal of first and second tree and store on list The complexity is the same for two operations on two trees, becouse trees have n-nodes.

And then merge two list to one list.

For example



Construct a new balanced tree from the merged list. This step takes $O(2n) = O(n)$ times

5

```
define function isSubarray (arr1, arr2):

    initilaze hashset

    for i in range (0, length of arr1) :
        add hashset element of arr1[i]

    for i in range (0, length of arr2):
        if arr2[i] in hashset:
            continue
        else
            return False

    return True
```

* The algorithm runs => linear time becouse adding hashset O(1) time but if "collusion" if exist then it will runs O(n) time in worst case

The function generally runs $\boxed{O(n) \text{ linear time}}$