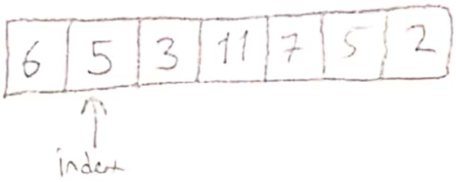
CSE-321 Homework #2

Muhammed Yasir Fidan
161044056

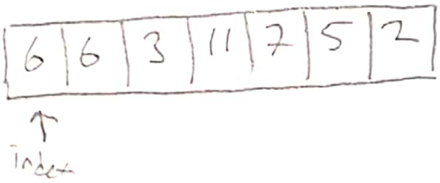
1-) Array = {6, 5, 3, 11, 7, 5, 2}

CheckValue = 5



6	5	3	11	7	5	2

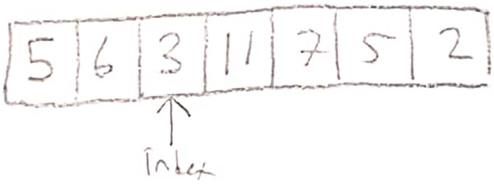
Compare 6 and 5, 5 is smaller so replace index with 6



6	6	3	11	7	5	2

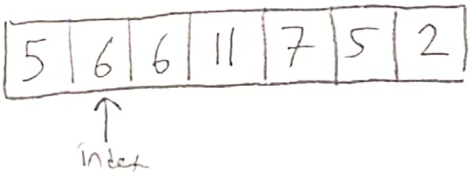
We are in first position replace first element with CheckValue(5)

CheckValue = 3



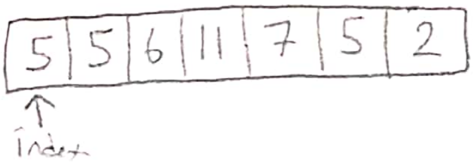
5	6	3	11	7	5	2

Compare 3 and 6, 3 smaller so replace 3 with 6



5	6	6	11	7	5	2

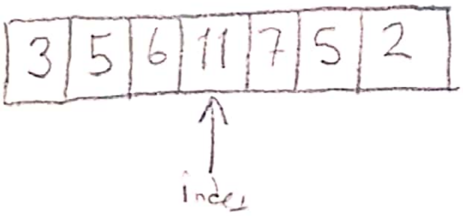
Compare 5 and 6, 5 smaller so replace 6 with 5



5	5	6	11	7	5	2

We are in first position, just replace first element with CheckValue(3)

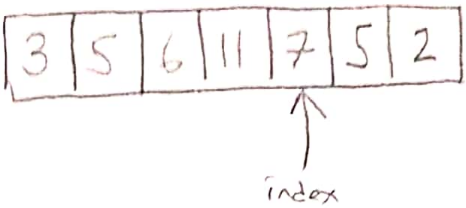
CheckValue = 11



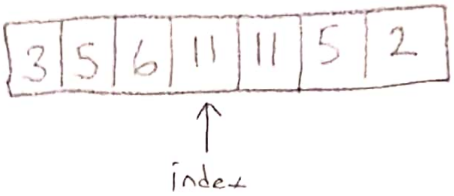
3	5	6	11	7	5	2

Compare 11 and 6, 11 is greater so don't change anything.

CheckValue = 7

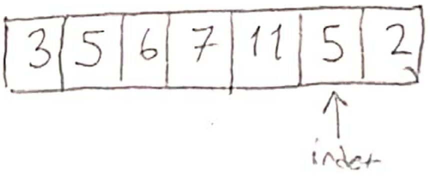


Compare 7 and 11, 11 greater so replace 7 with 11.

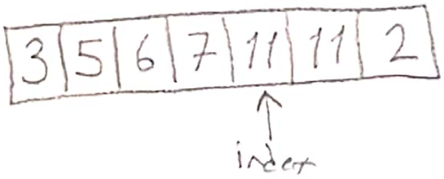


Compare 6 and 7. 6 smaller than 7 so replace index with 7.

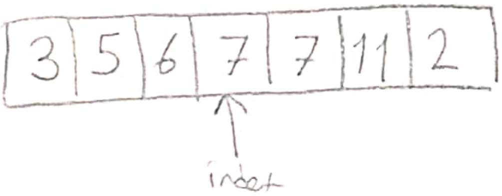
CheckValue = 5



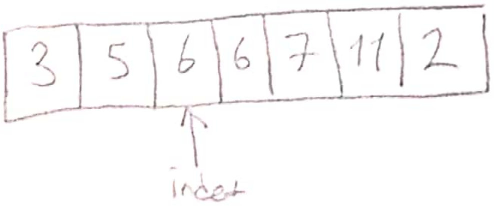
Compare 5 and 11. 11 greater than 5 so replace 5 with 11.



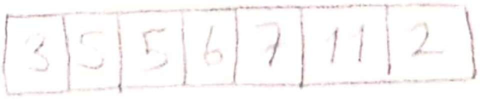
Compare 7 and 5 7 geater than 5 so replace index with 7



Compare 6 and 5 6 greater than 5 so replace index with 6



Compare 5 and 5 5 is not greater than 5 so replace index with 5



CheckValue = 2

3	5	5	6	7	11	2
						↑ index

Compare 11 and 2
11 greater than 2 so
replace index with 11

3	5	5	6	7	11	11
					↑ index	

Compare 7 and 2
7 greater than 2 so
replace index with 7

3	5	5	6	7	7	11
				↑ index		

Compare 6 and 2
6 greater than 2 so
replace index with 6

3	5	5	6	6	7	11
			↑ index			

Compare 5 and 2
5 greater than 2 so
replace index with 5

3	5	5	5	6	7	11
		↑ index				

Compare 5 and 2
5 greater than 2 so
replace index with 5

3	5	5	5	6	7	11
	↑ index					

Compare 3 and 2
3 greater than 2 so
replace index with 3

3	3	5	5	6	7	11
↑ index						

We are in the first pos.
So just replace first pos with
CheckValue(2)

2	3	5	5	6	7	11

Outher loop reach end of array
so our algorithm completed
and array sorted.

2-)

a)

```
function (int n) {
    if(n==1) return;

    for(int i=1; i<=n; i++){  ———— Θ(n)
        for (int j=1; j <=n; j++){  ┐
            Printf("*");            │ Θ(1)
            break;                  ┘
        }
    }
}
```

Firstly if n is equal 1 this function time complexity will be $\Theta(1)$ because it just enter first if statement ad return. So, best case for this algorithm $T(n)_{Best} = \Theta(1)$

If n is not equal 1 time complexity will be $\Theta(n)$ because first loop run n times but inner loop only run 1 time because it will break out after every first iteration. Also print and break statements are constant. So total time complexity will be $\Theta(n)$

$$T(n)_{worst} = T(n)_{Average} = \Theta(n)$$

So we can say;

$$T(n) = O(n)$$

b)

```
     Void function(int n) {
         int count=0;
①        for(int i=n/3; i<=n; i++)
②            for(int j=1; j+n/3<=n; j++)
③                for(int k=1; k<=n; k=k*3)
                     count++   — constant
 }
```

for loop ③

k
1
3
9
⋮
3^k

$3^k = n \Rightarrow k = \log_3 n$

so this loop $O(\log_3 n)$

for loop ②

$\sum_{j=1}^{\frac{2}{3}n} 1$ ↖

for(int j=1; j<=2n/3; j++) This loop runs $\frac{2}{3}n$ times after discarding constants we can see its time complexity $O(n)$

for loop ①

$\sum_{i=n/3}^{n} 1$ ↖

for(int i=n/3; i<=n; i++) This loop runs $n-\frac{n}{3}+1 = \frac{2n}{3}+1$

After Ignore constants we find its time complexity $O(n)$

So; After combine this 3 loop we find our complexity as;

$$O(n^2\log_3 n)$$

3-)

```
Procedure func(L[1:n], desired_num)
    L.sort()
    i=1
    j=n
    prev_pair=null
    while i<j
        Value = L[i] * L[j]
        if (Value = desired_num)
            if( prev_pair = null or prev_pair != (L[i], L[j]))
                yield (L[i], L[j])
                prev_pair = (L[i], L[j])
            end if
            i++
            j--
        end if
        else if( value < desired_num)
            i++
        end else if
        else
            j--
        end else
    end while
end procedure
```

- In my algorithm, firstly it sort given list. For sorting we can use merge, quick or heap sort. Python sort function use merge sort. So we can sort array in O(nlogn) complexity. After that I declare 2 Variable i and j. i starts from the first element of list and j starts from the last element of list and while i<j check L[i]*L[j]

If this Value smaller than our desired number just incremet i by 1

if this value greater than our desired number just decrement j by 1

Also I declare a variable called prev_pair. This variable hold last returned pair. So with using this variable I check new pair is same as the previos pair or not. if they not same I return this pair. So because of that I don't return same pairs multiple times.

So, If value and desired num equals and previos pair and new pair is different yield this new pair and set prev_pair with this pair. Than increment i by 1 and deccrement j by 1.

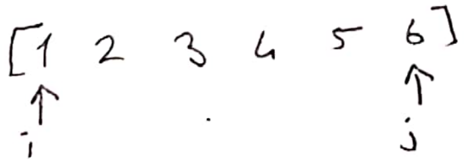
Lets show this algorithm with an exemple;

Lets our list = $\{1, 2, 3, 6, 5, 4\}$ and desired_num = 6

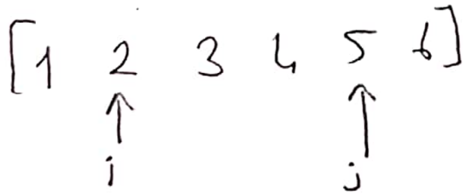
first we sort this list and it becomes $[1, 2, 3, 4, 5, 6]$

this sorting take $O(n \log n)$

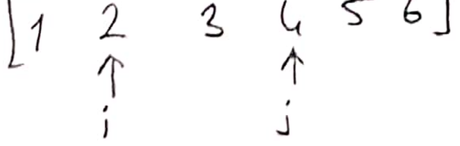
Now we are in while loop



L[i] * L[j] = desired_val
So yield (L[i], L[j])
and increment i by 1 and decrement j by 1



L[i] * L[j] > desired_val
So just decrement j by 1.



L[i] * L[j] > desired_num
So just decrement j by 1

[1 2 3 4 5 6]
↑ ↑
i j

$L[i] + L[j]$ = desired value
So yield $(L[i], L[j])$
and increment i by 1 and decrement
j by 1.

[1 2 3 4 5 6]
↑ ↑
j i

now $i > j$ So break loop and
end function.

So, this while loop just traverse list. Its complexity is $O(n)$
But in the begining of algorithm we have $O(n\log n)$ complexity
sorting.

Hence, this algorithm time complexity;

$$T(n) = O(n\log n + n) = \underline{O(n\log n)}$$

4)
```
procedure mergeBST(BST Tree1, BST Tree2)
        List Tree1_list;
        List Tree2_list;
        inorder(Tree1, Tree1_list) //Create a sorted list with tree1 elems
        inorder(Tree 2, Tree2_list)//create a sorted list with tree2 elems

        List m_list;
        merge_list(Tree1_list, Tree2_list, m_list)

        Merged_Tree = create_merge_tree(m_list, 0, m_list.length)
        return merged_tree
end procedure
```

```
procedure inorder(Tree , list)
    if(Tree != null)
        inorder(Tree.left, list)  ----> $T(\frac{n}{2})$
        list.add(Tree.data)  ----> 1
        inorder(Tree.right, list)  ----> $T(\frac{n}{2})$
    end if
end procedure
```

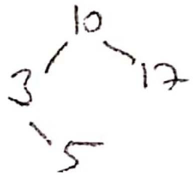
```
procedure Merge_list(L1, L2, L3)
    i=0, j=0, k=0

    While i<L1.lenght and j<L2.lenght
        if(L1[i]<L2[j])
            L3[k++] = L1[i++]
        end if
        else
            L3[k++] = L2[j++]
        end else
    end while
    while i<L1.lenght
        L3[k++] = L1[i++]
    end while
    while j<L2.lenght
        L3[k++] = L2[j++]
    end while
end procedure
```

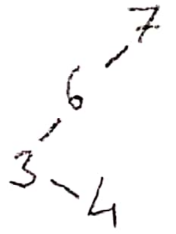
```
procedure create_merge_tree(list, first, last)
    if (first > last)
        return null
    end if
    mid = (first + last) /2
    node = new Node(list[mid])
    node.left = create_merge_tree(list,first,mid-1)
    node.right = create_merge_tree(list, mid+1, last)
    return node
end procedure
```

In my algorithm, mergeBST create 2 list and I fill them with inorder traversal values, So because of inorder traversal of BST these lists will be ordered. for example lets our trees like these:

Tree 1



Tree 2



After inorder traversal for tree 1   List1 will be [3,5,10,17]

After inorder traversal for tree 2   List2 will be [3,4,6,7]

And our inorder traverse algorithm has $T(n)=2T(\frac{n}{2})+1$

So, inorder traverse part has $O(n)$ time complexity

After that I merge this 2 sorted list with merge_list function. This function traverse both list and we know our both lists has same n lenght.
So, this function has $O(n+n) = O(n)$ time complexity.

[3 5 10 17]   [3 4 6 7]

[3 3 4 5 6 7 10 17]

Than, we create our merged tree with turning this list to a BST. Also my create_merge_tree algorithm
$T(n) = 2T(\frac{n}{2}) + c$ So it has $O(n)$ time complexity.

So, My merge algorithm has $O(n+n+n) = O(n)$ time complexity.

5-) This problem can be solved by brute force. We can search every smaller array element in the big array. But this method has $O(n^2)$ complexity. For linear time complexity we can use a hashmap. Firstly we can add smaller array elements to the our hashmap then just iterate bigger array and check 'our hashmap has' these elements or not. This method has linear time complexity Lets see this on pseudo code;

```
procedure find_elements(arr_1, arr_2)
        my_map = new HashMap()
        i=0                                  ]
        while i<arr_1.length                 ]  O(m)
              my_map.add(arr_1[i])           ]
        end while                            ]

        j=0
        while j<arr_2.length                 ]
              if(my_map.contains(arr_2[j]))  ]  O(n)
                      yield arr_2[j]         ]
              end if                         ]
        end while
end procedure
```

As we know, hashmap has constant time complexity for add and contains methods. So, if our smaller array length is m first while loop complexity will be $O(m)$. And if bigger array length is n, second while complexity will be $O(n)$.

So, this algorithm complexity will be $O(m+n) = \underline{O(n)}$