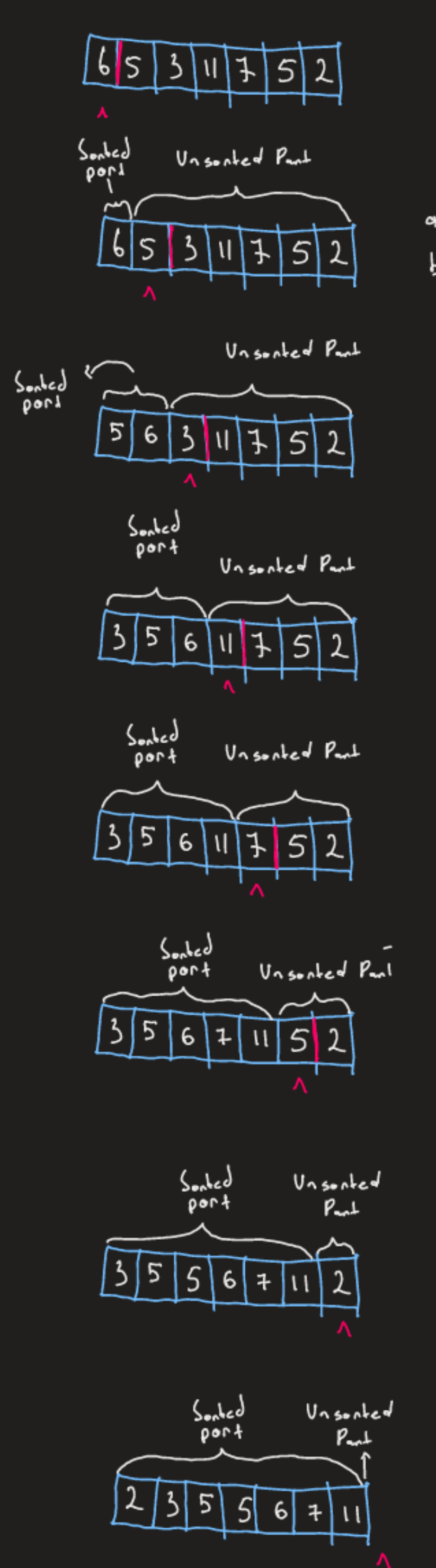
Hw 2

Thursday, November 12, 2020   5:38 PM

1-)

Array = [6,5, 3, 11, 7, 5, 2]



a) 5 < 6 swap
b) start of the array

a) 3 < 6 swap
b) 3 < 5 swap
c) start of the array

a) 11 > 6 do nothing

a) 7 < 11 swap
b) 7 > 6 do nothing

a) 5 < 11 swap
b) 5 < 7 swap
c) 5 < 6 swap
d) 5 = 5 move pointer to next index

a) 2 < 11 swap
b) 2 < 7 swap
c) 2 < 6 swap
d) 2 < 5 swap
e) 2 < 5 swap
f) 2 < 3 swap

Array is sorted.

2-)

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

If n equals 1 $f(n) \in O(1)$ because when n==1 funtion terminates itsel with return but if n > 1, we have to analyze lower parts too. Since first for loop is simply linear there is nothing to say. Then second for loop, it terminates itself with break. So second for loop is O(1) time operation. In the end total complexity can be shown as:

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

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

This function is a bit tricky to analyze but if we look precisely we can see that first two for loops have O(n) complexity. To illustrate first for loop starts from $n/3$ to n and increments by 1. So it is O(n) complex. Second one starts from $n/3+1$ and increment by 1 till it gets bigger than n. Since it iterates linearly it is O(n) complex too. On the otherhand last for loop start from 1 and increments by 3 till it gets bigger than n. So it is $O(\log_3 n)$

$$f(n) \in O(n) \cdot O(n) \cdot O(\log n)$$

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

3-)

```
#%%
from self_balancing_binary_search_tree import SBBST
nums = [1,2,3,6,5,4] # random numbers
#%%
"""
This function's complexity is O(nlogn). If we dive in findPair
function we can see that :
    1-  The first line of this function defines
        a new SBBST which's complexity is O(1) time.
    2-  Then we see a for loop which iterates over
        arr. This one is O(n) operation.
    3-  Inside for loop function checks that the
        remainder of the desired number after modulo
        operation by current element of the array whether
        equals zero or not. This one is O(n) too.
    4-  When if statement in the 3rd part has true as an
        input, function searches target inside SBBST.
        It is O(logn) time operation.
    5-  If SBBST has target function prints the pair.
        This operation is O(1) time.
    6-  Else function inserts current element of the array
        into SBBST. This operation is O(logn).
To sum up, since after 2nd state all operations happens
inside for loop O(n) comes from this linear iteration.
Then O(logn) comes from "search or insert" case. (Since
one of them has to be happen).
Complexity of this function : O(nlogn)
"""
#%%
def findPair (arr, des):
    ST = SBBST()

    for ele in arr:
        if (des%ele ==0):
            target = int(des/ele)
            if ST.search(ST.head,target):
                print("({},{})".format(ele,target))
            else:
                ST.insert(ele)
#%%
ST = SBBST()
arr= [-2,-4,2,4,1]
findPair(arr,8)
```

4-)

We have 2 BST with n nodes and we have to merge one to another. I use the algorithm below:

1-Convert BST-1 to array-1 with in-order traversal
2-Convert BST-2 to array-2 with in-order traversal
3-Merge this 2 array by merge sort like algorithm and get array-3
4-Convert this array-3 to the BST.

My Pseudo Code:

```
# binary tree node
class Node:
    def __init__(self, d):
        self.data = d
        self.left = None
        self.right = None
"""
This method converts a binary tree to a list in inrder traversal order
"""
def storeInorder(node, inorder):
    if node == None:
        return
    # first go left part
    storeInorder(node.left, inorder)
    # add to the array
    inorder.append(node.left.data)
    # then go right part
    storeInorder(node.right, inorder)
"""
This method merges two sorted list
in ascending order. Returns the the
merged list.
m: lenght of the list1
k: lenght of the list2
"""
def merge(list1, list2, m, k):
    i=0
    j=0
    res = list()
    while i<m and j<k:
        if list1[i] < list2[j]:
            res.append(list1[i])
            i+=1
        else:
            res.append(list2[j])
            j+=1
    while i<m:
        res.append(list1[i])
        i+=1

    while j<k:
        res.append(list2[j])
        j+=1

    return res

def sortedListToBST(aList, i, j):
    if(i > j):
        return None

    middle = (i+j)/2
    head = Node(aList[middle])
    head.left = sortedListToBST(aList, i, middle-1)
    head.right = sortedListToBST(aList, middle+1, j)
    return head
```

I have 3 methods to handle this algorithm:

- storeInorder
- Merge
- sortedListToBST

First method converts a BST to sorted list. If we analyze, it is an Recursive function which calls itself N time and does O(1) time Operation inside. So there are O(N) call and O(1) complex operation In every call. So, this method's complexity is O(n) time.

Second method merges to sorted list and return the merged one. This function is iterative. So we need to look up loops inside of it. There are 3 while loops inside of the method. First while loop iterates O(m+n) , second one is O(m) and third one is O(n) time. All this loops are distinct from each other and all the operations inside loops are O(1) time. So to find method's total complexity we need to make the operation that:

O(m+n) + O(m) + O(n) = O(m+n) and since both of our BST's are N- node this is **O(n)**

Third method, converts sorted list to BST. This method is also recursive. Since the list is sorted, method divides list into two part as left and right. It continues this operation till there is nothing to divide. When function reaches there. It stops creating new method calls to create BST. So there are O(n) calls. In the method it calculates the middle index according to arguments then it assigns the value in the "*middle*" index to the head. These are O(1) operations. So to find total complexity of this function:

O(n) * O(1) = O(n) since my list's size is 2n it O(2n) = O(n)

I we look up to whole picture at a glance, we call 3 methods consecutively

2X storeInorder	2 X O(n)
1X Merge	O(n)
1X sortedListToBST	O(n)
RESULT	2 x O(n) + O(n) + O(n) = **O(n)**

5-)

```
#%%
"""
I used set in my implementation. Because python uses Has
hSet and in the Hash Set structure every number has spec
ific hash value.
And getting this hash value is O(1) time operation. Beca
use generally hashed value is produced with the help of
a constant prime
number and modulo.(I also checked my axiom from here: ht
tps://www.jessicayung.com/how-python-implements-
dictionaries/) And a Hash
Set holds all of its element as like multi dimensional a
rray (which referred as "table" in many documents:
https://www.laurentluce.com/posts/python-dictionary-
implementation/) of hashed values, so getting an element
 is O(1) time if you
know the desired element.  Since there is an array in ha
sh sets and elements are placed according to their hash
values, there could
be collisions but python uses probing to overcome this s
ituation and also it increments the vertical size of the
 array. As a result
using Set gives opportunity use add and get operations i
n O(1) time*.
At first function converts the large array to the python
 set. This is O(n) time operation. After that function i
terates over small
array and checks whether the set which is produced from
large array  has the current element of the small array
or not. This is an
O(1) time operation because  according to python documen
ts the Set is implemented by using HashSet. If set has c
urrent  element,
function appends the current element into the result lis
t. Also this one is O(1) time  complex.
To sum up there is a for loop which O(n) and there is an
 other for loop which also O(n) and second for loop has
two sub-operations
inside. Both of them are O(1) time operations. So this f
unction has O(n) time complexity.
    • Note : I used python's doc's for my axioms : https
://wiki.python.org/moin/TimeComplexity
"""
#%%
def findCommon(bArr, sArr):
    temp = set()
    res = list()
    for ele in bArr:
        temp.add(ele)
    for ele in sArr:
        if(ele in temp):
            res.append(ele)
    return res
#%%
arr1 = [1,2,3,4,5,6,7]
arr2 = [1,2,3]
print(findCommon(arr1,arr2))
```