Yusuf Patoglu - 151044027

CSE 321 HOMEWORK 4 REPORT

5-) Warehouse Problem

In our problem we have two arrays first array's elements are costs to buy goods and second one's elements are selling prices of those goods on market. There is a rule in question that says we need to sell each item on next day. So our goal will be finding best day and situation to sell these goods.

For 10 day period we have Costs and Prices arrays. In order to find the profit (or no profit) between buying and selling values we need to substract those to like Prices[index + 1] - Costs[index] and store in new array named with differences. As it is obvious this one takes O(n) time:

```
for x in range (len(C)- 1):
    if((P[x + 1] - C[x]) == 0):
        print("Can't sell for day:", x + 1)
    differences.append(P[x + 1] -C[x])
```

After finding profits and inserting all of them into differences array, now we have to find best buy day value with divide and conquer algorithm. Instead of sorting the list with divide and conquer way we can use quicksort algorithm to find greatest element in list, which is best day value:

```
res = quickSelect(differences, 0, len(differences) - 1, len(differences) - 1)
day=differences.index(res)
return day + 1
```

Worst Case for QuickSelect:

First Step: T(n) = c * n + T(n / 2)

T(n / 2) = c * n / 2 + T(n / 4)

T(n / 4) = c * n / 2 + T(n / 8)

T(n / 8) = c * n / 2 + T(n / 16)

.

.

.

T(2) = 2 * c + T(1)

T(1) = c + ...

If we sum all of the T(n) + T(n-1) + … T (1) we'll get c* (n + n / 2 + n / 4 + .. 1) = **O(n)**

4-) Bipartite Graph Problem

Since the solution of this problem will be with Decrease and Conquer algorithm then we need to visit and check all vertices of graph and see graph is bipartite or not.

In order to achive this I will use BFS algorithm(I will be using Queue) which is decrease and conquer. Also while traversing the graph we need to color each node and check all adjacent ones.

For example for node a lets suppose there are other nodes b,c,d and e. All b,c,d and e need to be different color with a. Otherwise the graph is not bipartite which means it can't be derived into two subgraphs.

We know that we can represent graphs with adjacency list or adjacency matrix. For simple and readable implementation I used adjacency matrix.

```
graph = [[0, 1, 0, 1],
         [1, 0, 1, 1],
         [0,1, 0, 1],
         [1, 1, 1, 0]]
```

Our process continues for each unvisited vertex u and for each neighboor v of u. So we can think it like two nested loops. Therefore the complexity will be **O(V^2).**

```
while(not Q.empty()):                                   Outer Loop
    top = Q.get()

    if(graph[top][top] == 1):
        return False

    for vertice in range (len(graph)):                  Inner Loop
        if(graph[top][vertice] != 0):
            if(colorList[vertice] == -1):
                colorList[vertice] = -2
                Q.put(vertice)
        elif(graph[top][vertice] != 0):
            if(colorList[vertice] == colorList[top]):
                return False

return True
```

Source: GTU - CSE321 Midterm Exam(2016). Question 3.

Yusuf Patoglu - 151044027

3-) Contiguous Subset Problem:

Our problem is finding the sub array whose elements sum is greater than all others. Straightforward solution could be generating all subsets of an array and check each of their sum and return the greater one. But we need to design more clever algorithm like divide and conquer instead of this Brute-Force method.

In order to solve our problem with divide and conquer we need two cursor. As obvious the sum with laying between this cursors. Then we can apply main divide and conquer logic. We will divide the list into two pieces. Left side and right side. We will find greater sum in these two parts. Since we divide our list into 2 pieces the greater sum can be in left side, right side or middle(unlikely). We need to solve both three cases and return greatest one. I will be solving these parts recursively:

```
leftOne = maxContSubArray(mylist, lowBound, mid)
rightOne = maxContSubArray(mylist, mid + 1, highBound)
```

Complexity Analysis:

We have 2 recursive calls for left and right side. Which divides list into two pieces. Therefore our recurrence relation's one part will be 2 * T(n / 2) for sure. Also we have a loop that iterates n times for calculating current part of a list. Since it will be applied to all elements of a list we can say that it's n times. Therefore final equation will be 2 * T(n / 2) + n.

In order to solve the equation with reduction:

a-) T(n) = 2 * T (n / 2) + n

b-) T(n / 2) = 2 * T (n / 2 ^ 2) + n / 2 <--- Replace this one and place into equation a you will get c.

c-) T(n) = 2 * [2 * T (n / 2 ^ 2) + n / 2] + n which is T(n) = (2 ^ 2) * T(n / 2 ^ 2) + n + n

d-) T(n / 2 ^ 2) = 2 * T (n / 2 ^ 3) + (n / 2 ^ 2) <--- Replace this one and place into equation c and you will get e.

e-) T(n) = (2 ^ 3) * T (n / 2 ^ 3) + 3 * n

Now looking at a,c and e we got our pattern. Therefore our main equation will be:

T(n) = (2 ^ k) * T(n / (2^k)) + k * n

Assume that n = (2^k) then k = logn.

T(n) = (2 ^ k) * T(1) + k * n

T(n) = n * 1 + n * logn

Therefore it's O(n + n * logn) which is **O(n * logn)**.

2-) Kth element of two sorted arrays:

In order to solve our problem we need to have two arrays which are sorted.

```
myList1 = [1,3,5,7,9,10,21,324]
myList2 = [0,2,4,6,8,123,344,1111]
```

There is a popular problem: "Finding the median of two sorted arrays." For this median problem we can compare middle elements of sub arrays which are left subarray and right subarray. We can apply the same logic here but instead of comparing middle elements I will be comparing first elements. Our base case is k == 1 where we compare first elements of two sides:

```
if k == 1:
    if list1[0] > list2[0]:
        return list2[0]
    else:
        return list1[0]
```

Recursive step is comparing k / 2. Elements of both arrays.
In order to achieve this we need to keep recursing with k / 2 where k is our input, then slice the both arrays from botttom or up:

```
if(list1[firstPivot - 1] <= list2[secondPivot - 1]):
    return kthElem(list1[firstPivot:], list2, k - firstPivot,  m - firstPivot, n)
else:
    return kthElem(list1, list2[secondPivot:], k - secondPivot, m,  n - secondPivot)
```

The time complexity is obvious because only thing we do is dividing both arrays into 2 pieces nothing more. Therefore it's **O(logn).**

1-) Special Array

a-)

We need to found that = A[i,j]+A[i+1,j+1] <= A[i,j+1]+A[i+1,j]

We can replace i + 1 with k where i < k. So we'll get: A[i,j]+A[k,j+1] <= A[i,j+1]+A[k,j]

We replace k with i + 1, so base case is k = i + 1. Since we want k + 1 form k + 1 = i + 1 + 2 + 3 ..+ n + 1 which is k + 1 = n + i + 1

A[i,j]+A[k,j+1] <= A[i,j+1]+A[k,j]

A[k,j]+A[k+1,j+1] <= A[k,j+1]+A[k+1,j]

➔ A[i,j]+A[k,j+1]+A[k,j]+A[k+1,j+1] <= A[i,j+1]+A[k,j]+A[k,j+1]+A[k+1,j] (Replace)
➔ A[i,j]+A[k+1,j+1] <= A[i,j+1]+A[k+1,j]

b-)

We have a special rule that A[i, j] + A[i + 1, j + 1] ≤ A[i, j + 1] + A[i + 1, j]. So while constructing 2d array we need to check this condition on every iteration. We only have one chance to fix the specified array according to this condition. Simply I will use a counter for this process. If counter is equal to 1 then it means we constructed our array successfully otherwise we couldn't.

```
for x in range (len(specialArr) - 1):

    for y in range (len(specialArr[0]) - 1):
        if((specialArr[x][y] + specialArr[x + 1][y + 1]) < (specialArr[x][y + 1]
            colList.append(specialArr[x][y])
        else:
            counter = counter + 1
            specialArr[x][y] = (specialArr[x][y + 1] + specialArr[x+1][y])
    rowList.append(colList)
    colList = []
if(counter == 1):
    return rowList
else:
    rowList = []
    return rowList
```