Ahmed Semih Ozmekik

Question 1

1. We have a definition for an array type called a **special** array: An $m \times n$ array A of real numbers is called a special array if for all i, j , k, and l such that $1 \leq i \leq k \leq m$ and $1 \leq j \leq l \leq n$, we have:

$$A[i,j] + A[k,l] \leq A[i,l] + A[k,j] \tag{1}$$

   (a) Proof that an array is **special** if and only if for all $i = 1, 2, ..., m-1$ and $j = 1, 2, ..., n-1$ we have:

$$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j] \tag{2}$$
$$A[k,j] + A[i+1,l] \leq A[i,l] + A[i+1,j] \qquad k = i+1 \text{ and from } 1 \tag{3}$$
$$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j] \qquad l = j+1 \text{ and from } 3 \tag{4}$$

   We have proved that if the first predicate is true, being if an array is **special** array; the second one is true. Now, let's prove this from the other way around, to show the iff relation:

$$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j] \tag{5}$$
$$A[i,j] + A[i+1,l] \leq A[i,l] + A[i+1,j] \qquad k = i+1 \tag{6}$$
$$A[i,j] + A[i+x,l] \leq A[i,l] + A[i+x,j] \qquad k = i+x \tag{7}$$
$$A[i,j] + A[i+x+1,l] \leq A[i,l] + A[i+x+1,j] \qquad k = i+x+1 \tag{8}$$
$$A[k,j] + A[k+1,l] \leq A[k,l] + A[k+1,j] \qquad i = k \text{ and from } 6 \tag{9}$$

   Add 1 and 9 :

$$A[k,j] + A[k+1,l] + A[i,j] + A[k,l] \leq A[k,l] + A[k+1,j] + A[i,l] + A[k,j] \tag{10}$$
$$A[k+1,l] + A[i,j]A[k+1,j] + A[i,l] \tag{11}$$

   Apply the same induction to column j:

$$A[i,j] + A[j+1,l] \leq A[i,j+1] + A[k,j] \qquad l = j+1 \text{ base} \qquad (12)$$
$$A[i,j] + A[i,j+x] \leq A[i,j+x] + A[i,j] \qquad l = j+x \text{ assume} \qquad (13)$$
$$A[i,j] + A[i,l] \leq A[i,l] + A[i,j] \qquad (14)$$
$$A[i,j] + A[k,j+x+1] \leq A[i,j+x+1] + A[k,j] \qquad l = j+x+1 \qquad (15)$$
$$A[i,l] + A[i,l+1] \leq A[i,l+1] + A[k,l] \qquad j = l \qquad (16)$$

Add 16 and 14 :

$$A[i,l] + A[i,l+1] + A[i,j] + A[i,l] \leq A[i,l+1] + A[k,l] + A[i,l] + A[i,j] \qquad (17)$$
$$A[i,j] + A[k,j+x+1] \leq A[i,j+x+1] + A[k,j] \qquad (18)$$

Equation 9 and 11, 18 and 15 are equal. Therefore:

$$A[i,j] + A[i+1,l] \leq A[i,l] + A[i+1,j] \qquad (19)$$

Holds.

(b) Algorithm that converts a 2D array into the **special** array only if it can be done with changing one single element in the array:

Algorithm 1 Fixing the special array.

1: **procedure** FIXARRAY(A) $kernel \leftarrow [4][m][n](0\text{xcolumn})$
2: $kernel[0,1,2,3][][] \leftarrow [0,0,...,0]$
3: $kernel[0][][] \leftarrow$ apply 2x2 kernel, increase points existing in the incorrect 2x2 matrix
$kernel[1][][] \leftarrow$ apply 3x3 kernel, increase points existing in the incorrect 3x3 matrix
$kernel[2][][] \leftarrow$ apply 4x4 kernel, increase points existing in the incorrect 4x4 matrix
$kernel[3][][] \leftarrow$ apply 5x5 kernel, increase points existing in the incorrect 5x5 matrix
5: $point \leftarrow$ highest ranked point among kernel
6: $point \leftarrow$ find the boundaries among the kernels of the incorrect point
7: **end procedure**

As it stated, we first create a 4 different arrays. This procedure is for finding the element causes this array not to become special array. 2x2 is not enough because we have edges and our point might be there. Instead we create 4 different kernels to test edge points and other points together. As we test, if the kernel results wrong, we increase rank of the point by 1 for that kernel. After all this, we get a rank list with different kernels. Among all of them, we find the point with

highest rank which caused error in all kernels, is the element prevents this array to become special array.
After finding we need to correct the element. Regarding to definition we have an equation given. In 4 different kernels, we apply element in this equation to find the different boundaries. After we got our boundaries, among 4 kernels, we pick the max of the lower boundary, and min of the upper boundary to make our equation to fit to every kernel. Finally, we pick an item in that boundary and fix the item and therefore the fix the array with one move.

(c) Implementation a **divide-and-conquer** algorithm that finds the leftmost minimum element in each row.

```
def leftmost_min(lst, leftmost, row, column, k):
    if row == 1:
        leftmost[0] = lst[0].index(min(lst[0][0:column]))
    else:
        mid = int(ceil(row, 2.0))
        leftmost_min(lst, leftmost, mid, column, 2 * k)
        for i in range(1, row - 1, 2):
            idx = k * i
            end = leftmost[idx + k] + 1
            leftmost[idx] = lst[idx].index(min(lst[idx][leftmost[idx -
                k]:end]))
        if row % 2 == 0:
            idx = k * (row - 1)
            leftmost[idx] = lst[idx].index(min(lst[idx][leftmost[idx -
                k]:column]))
```

We a construct a sub array of A consisting of the even numbered rows of A. And recursively determine the leftmost minimum for each for of this array. We used the definition given for **special** array.

(d) Recurrence relati eon for the running time the algorithm given in (c): m being rows, and n being columns. Each time we split the size to even rows. Merging takes $T(m, n) \in \mathcal{O}(m + n)$ times. Because in each cost of checking is n at most. Recurrence relation becomes like this:

$$T(m, n) = T(m/2, n) + m/2 + n \tag{20}$$

Scoping the resolution:

$$T(m,n) = T(m/2,n) + m/2 + n \tag{21}$$
$$T(m,n) = T(m/4,n) + m/4 + n + m/2 + n \tag{22}$$
$$T(m,n) = T(1) + m \times \sum_{i=1}^{\log m} \frac{m}{2^i} + n \log n \tag{23}$$
$$T(m,n) = T(1) + m \times (2 - 2/m) + n \times \log n \tag{24}$$

Hence, $T(m,n) \in \mathcal{O}(n \log m)$

Question 2

1. From given two sorted arrays A and B, we will design a **divide-and-conquer** algorithm that finds k^{th} element. Here is the implementation of the algorithm:

```
def kth_element(A, B, m, n, k):
    if m > n: # switch arrays to make m <= n
        return kth_element(B, A, n, m, k)
    if m == 0:
        return B[k-1]
    if k == 1:
        return min(A[0], B[0])

    i = min(m, k // 2)
    j = min(n, k // 2)
    if A[i - 1] > B[j - 1]:
        return kth_element(A, B[j:], m, n - j, k - j)
    else:
        return kth_element(A[i:], B, m - i, n, k - i)
```

This algorithm assumes the given k is valid for the arrays. And k is between 1 and $m+n$. Meaning that first element of merged arrays is $k = 1$.
In this algorithm, in each recursion we decide which array to go on for search and then divide our problem regarding to that. If the index of A has reached to 0, we just return the k^{th} element from B. Or if the k^{th} became 1, we return the minimum element among the first elements of A and B.
Other than base cases, we get the i and j as the minimum one among k^{th} and their m and n indices respectively. After finding the i and j, we check which item is bigger on that indices in A and B. Regarding to solution we either decrease the A or B, then continue to searching.

2. **Worst Case**: Our worst case happens when union of A and B are not homogeneous.

This will cause, divide operation to continue until both of the arrays become empty. It will cause A to be divided $\log m$ times, and B to be divided $\log n$ times.

Hence, $T(m, n) \in \mathcal{O}(\log m + \log n)$

Question 3

1. Given an array of integers **A[1 ; ... ; n]** we need to find the contiguous subset having the largest sum within A with a **divide-and-conquer** algorithm. Here is the pseudo code of the algorithm:

Algorithm 2 Finding the contiguous subset having largest sum.

```
1: procedure FINDSUBSET(A, i, j)
2:     if i = j then return A
3:     end if
4:     if sum(A[i : j − 1]) > sum(A[i + 1 : j]) then          ▷ Decrease-by-1
5:         k, l ← i, j − 1
6:     else then
7:         k, l ← i + 1, j
8:     end if
9:     B ← FINDSUBSET(lst, k, l)
10:    if sum(lst[i : j] > sum(B)) then
11:        return lst[i : j]
12:    else then
13:        return B
14:    end if
15: end procedure
```

2. Implementation.

```
def largest_sub(lst, i, j):
    return (i, j - 1) if sum(lst[i:j - 1]) > sum(lst[i + 1: j]) else (i +
        1, j)

def find_largest_cont_sub(lst, i, j):
    if i == j:
        return lst
    k, l = largest_sub(lst, i, j)
    sub = find_largest_cont_sub(lst, k, l)
    return lst[i:j] if sum(lst[i:j]) > sum(sub) else sub
```

Let's explain the algorithm here. The above function returns the sub array, in essence the indices of the sub array. It decreases the array we are searching for the sub list in it, will decrease by one from top or bottom, from which has the largest sum. By this way and with recursion we will be checking each sub array and only returning the larger one. Let's follow the algorithm to see what is going on. This is the array A:

5	−6	6	7	−6	7	−4	3

Once we compare the first 6 (top) or last 6 (bottom) items of array, we decide which one has largest sum and return the indices. Then we compare the current array with this sub array, hence returning array having the largest sum. With this way in each recursion, say current array has 5 elements, hence the sub array has 4 elements and there are contiguous 2 sub arrays, leftmost and the right most. We pick the array having largest sum among them. And once we get it, we again pick the array having the largest sum among the current array with size 5 and largest sub array with size 4, compare them and find the array having the largest sum.

3. **Worst Case**: Recurrence relation is, $T(n) = T(n-1) + 3$.
   Hence, $T(n) \in \mathcal{O}(n^2)$

Question 4

1. Implementation.

```
# G is adjacency list. n is len(G). color = [].
def bipartite(G, n, color):
    unvisited = False
    if not color:
        color = [-1] * len(G)
    if n < len(G):
        if color[n] == -1:
            unvisited = True
            for v in G[n]:
                if color[v] != -1:
                    unvisited = False
                    if color[n] == color[v]:
                        return False
                    else:
                        color[n] = 1 if color[v] == -1 else -1
        if unvisited:
            color[n] = -1
        return bipartite(G, n + 1, color)
    else:
```

```
        return True
```

A bipartite graph is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. Or in other terms, when the chromatic number of the graph is 2, the graph is bipartite. In our algorithm, we are coloring the graph. 1 represents the color and -1 is a representation of not colored. And checking the colors afterwards.
When all vertices are checked, it returns true. In the other case, while peeking the vertices we give it a color and then check the colors with it's adjacency list.

2. **Worst Case**: Let's write the recurrence relation: (In the terminology of graph theory, n being V, vertex number.)

$$T(n) = T(n-1) + E_n + 1 \tag{25}$$
$$T(n) = T(n-2) + E_{n-1} + 2 \tag{26}$$
$$T(n) = T(n-3) + E_{n-2} + 3 \tag{27}$$

We get: $\sum_{i=1}^{n} E_i + n$. Or if we write this in terms of graph theory: $\sum_{i=1}^{V} E_i + V$. Since the sum is $2E$, result is $T(V) = 2E + V$.

Hence, $T(V, E) \in \mathcal{O}(E + V)$

Question 5

1. Our algorithm is merge-sort like algorithm. In both array by dividing them into halves we reach into the leaf elements and from bottom we come back up with adding the two elements from both arrays. Hence returning the largest element. Here is the implementation:

```
def best_goods_day(C, P, i, j):
    if i + 1 == j: # Leaf item
        return i
    m = (i + j) // 2
    day1 = best_goods_day(C, P, i, m)
    day2 = best_goods_day(C, P, m, j)
    return day1 if P[day1] - C[day1] > P[day2] - C[day2] else day2
```

In this algorithm we assumed that. C and P are given as such, respectively.

0	1	2	3	4	5	6	7	8
5	11	2	21	5	7	8	12	13

0	1	2	3	4	5	6	7	8
7	9	5	21	7	13	10	14	20

In this assumption we expect price array to be shifted to left by one. This alignment is very convenient because when we use 10 index for 10 days, 1 item space will be wasted and it's easier to work with this way.
This algorithm is as mentioned, is very similar to divide-and-conquer merge-sort algorithm.

2. **Worst Case**: Recurrence relation is $T(n) = 2 \times T(n/2) + 3$. Applying the master theorem we get: $a = 2$, $b = 2$, $d = 0$.

   Hence, $T(n) \in \mathcal{O}(\log n)$