CSE 321 Homework 4 Solutions

1-) An m × n array A of real numbers is called a ***special*** array if for all i, j , k, and l such that 1 ≤ i < k ≤ m and 1 ≤ j < l ≤ n, we have:

$$A[i, j] + A[k, l] \leq A[i, l] + A[k, j]$$

1a-) The "only if" part is trivial, it follows form the definition of special array.

As for the "if" part, let's first prove that:

$$A[i, j] + A[i + 1, j + 1] \leq A[i, j + 1] + A[i + 1, j]$$

$$A[i, j] + A[k, j + 1] \leq A[i, j + 1] + A[k, j]$$

Where *i* < *k*.

Let's prove it by induction. The base case of *k*=*i*+1 is given. As for the inductive step, we assume it holds for *k*=*i* + *n* and we want to prove it for *k*+1=*i*+*n*+1. If we add the given to the assumption, we get:

$$A[i, j] + A[k, j + 1] \leq A[i, j + 1] + A[k, j]$$

$$A[k, j] + A[k + 1, j + 1] \leq A[k, j + 1] + A[k + 1, j]$$

$$A[i, j] + A[k, j + 1] + A[k, j] + A[k + 1, j + 1] \leq A[i, j + 1] + A[k, j] + A[k, j + 1] + A[k + 1, j]$$

$$A[i, j] + A[k + 1, j + 1] \leq A[i, j + 1] + A[k + 1, j]$$

1b-) First, look for the leftmost minimum elements for each row. For Special arrays, a rule is that the leftmost elements should always go in the right direction or stay where they are. This means that, if the 0th element of the first row is its leftmost minimum, then we need to have an index greater or equal to 0 for the next row. An example of a correct sequence of leftmost minimum elements is:

$$[1, 2, 2, 3, 5, 5, 7]$$

In this example, we can see that the sequence of indices of leftmost minimums of each row is a non-decreasing sequence. If we have given a one-off Special array, we can simply look at the sequence of leftmost minimum indices. An example of a faulty by one element Special array may have the following leftmost minimum indices:

$$[1, 2, 1, 3, 5, 5, 6]$$

In this sequence, 2nd index, or the 1 in the 3rd element is faulty (shown by red). To fix this, we can simply make the second row's leftmost minimum element as its 2nd or 3rd elements.

And we can do that by giving its 2nd or 3rd element a value lesser than the current minimum of that row.

Pseudocode:

```
procedure fix_non_special(array[rows, cols]):
    leftmost_mins = []
    for each row in array:
        leftmost_mins.add(min(row))
    end for
    faulty_row = -1
    faulty_element = -1
    foreach lm_min in leftmost_mins:
        if lm_min is smaller than its previous element
            faulty_row = lm_min
            faulty_element = leftmost_mins[lm_min]
        if lm_min is greater than its next element
            faulty_row = lm_min
            faulty_element = leftmost_mins[lm_min]
        end for
        new_min_value = array[faulty_row, faulty_element]
        new_min_index = leftmost_mins[faulty_row - 1]
        return faulty_row, new_min_index, new_min_value
end procedure
```

1c-) Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row in A'. Then compute the leftmost minimum in the odd-numbered rows of A.

If μi is the index of the i-th row's leftmost minimum, then we have

$$\mu i - 1 \leq \mu i \leq \mu i + 1$$

For i=2k+1, k≥0, finding μi takes μi+1−μi−1+1 steps at most, since we only need to compare with those numbers. Thus

$$\begin{aligned}
T(m,n) &= \sum_{i=0}^{m/2-1} (\mu_{2i+2} - \mu_{2i} + 1) \\
&= \sum_{i=0}^{m/2-1} \mu_{2i+2} - \sum_{i=0}^{m/2-1} \mu_{2i} + m/2 \\
&= \sum_{i=1}^{m/2} \mu_{2i} - \sum_{i=0}^{m/2-1} \mu_{2i} + m/2 \\
&= \mu_m - \mu_0 + m/2 \\
&= n + m/2 \\
&= O(m+n).
\end{aligned}$$

1d-) The divide time is O(1), the conquer part is T(m/2) and the merge part is O(m+n). Thus,

$$\begin{aligned}
T(m) &= T(m/2) + cn + dm \\
&= cn + dm + cn + dm/2 + cn + dm/4 + \cdots \\
&= \sum_{i=0}^{\lg m-1} cn + \sum_{i=0}^{\lg m-1} \frac{dm}{2^i} \\
&= cn \lg m + dm \sum_{i=0}^{\lg m-1} \\
&< cn \lg m + 2dm \\
&= O(n \lg m + m).
\end{aligned}$$

2-) By using a divide and conquer approach, similar to the one used in binary search, we can attempt to find the k'th element in a more efficient way.

Algorithm:

We compare the middle elements of arrays arr1 and arr2,

let us call these indices mid1 and mid2 respectively.

Let us assume arr1[mid1] k, then clearly the elements after

mid2 cannot be the required element. We then set the last

element of arr2 to be arr2[mid2].

In this way, we define a new sub-problem with half the size of one of the arrays.

We always reduce the problem size to half of the arrays so the complexity is equal to O(log(m)+log(n)) in the worst case.

3-) We can find the contiguous subset with the largest sum can be found by using the following steps:

1) Divide the given array in two halves
2) Return the maximum of following three
   a) Maximum subarray sum in left half (Make a recursive call)
   b) Maximum subarray sum in right half (Make a recursive call)
   c) Maximum subarray sum such that the subarray crosses the midpoint

The lines 2.a and 2.b are simple recursive calls. How to find maximum subarray sum such that the subarray crosses the midpoint? We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid-point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with sum point on right of mid + 1. Finally, combine the two and return.

maxSubArraySum() is a recursive method and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \Theta(n)$$

The above recurrence is similar to Merge Sort and can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is Θ(nLogn).

4-) Following is a simple algorithm to find out whether a given graph is Birpartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set U).

2. Color all the neighbors with BLUE color (putting into set V).

3. Color all neighbor's neighbor with RED color (putting into set U).

4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.

5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite). As we can see the BFS solutions has a decrease-and-conquer approach because we don't need to do backtracking after each vertex is colored.

Time Complexity of the given approach is same as that Breadth First Search. On the other hand, both the given algorithm and BFS change by the input type of the graph data. If the input consist of vertices and edges then the complexity becomes O(V^2) or if the graph is represented using adjacency list, then the complexity becomes O(V+E) where V is the vertex set and E is the edge set.

5-) This question doesn't make so much sense because of the expression that "store the goods for one day and sell them the next day". We could sell the goods in any day after the day that we bought the goods. So any reasonable answer will be considered as correct. The following example is one of them:

- ➔ Find the profit for each day by subtracting storing price of day k from market price on day k +1.
- ➔ Find the maximum profit by a divide-and-conquer approach.

The complexity of the algorithm below is equal to O(n) [to calculate profits] + O(logn) [to find max profit] = O(n).