CSE321

Introduction to Algorithm Design

Homework #4

Report

Oğuzhan Ağtuş
161044003

①

a) The "only if" part is unimportant, it follows form the definition of "special" array. As for the "if" part, we should prove following expression:

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

It can be proved by using 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:

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

$+ \; A[k,j] + A[k+1,j+1] \leq A[k,j+1] + A[k+1,j]$ (given)

$\rightarrow A[i,j] + \cancel{A[k,j+1]} + \cancel{A[k,j]} + A[k+1,j+1] \leq A[i,j+1] + \cancel{A[k,j]} + \cancel{A[k,j+1]} + A[k+1,j]$

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

b) We have proved the expression in the previous question. So we can use the following.

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

Think about an array like this:

	2		
37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

(1 and 3 mark the first and third columns)

(This is not a special array.)

We can traverse the whole array by analyzing quadruple elements. Example quadruples are showed with blue.

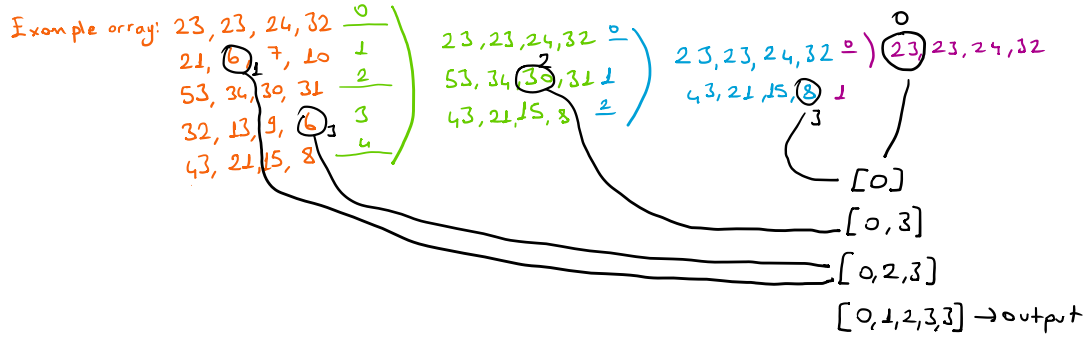
1- $37+6 \leq 23+21$ ✓
2- $23+7 \leq 22+6$ ✗
3- $22+10 \leq 32+7$ ✓

Check all quadruples. If there is more than one equation that is not satisfy the equation the array cannot be convert to special array. If not choose an optimal element from selected quadruple.

Pseudocode:

```
count=0
for i in range(m-1):                          } m x n array
  for j in range(n-1):
    if not (array[i][j] + array[i+1][j+1] <= array[i][j+1] + array[i+1][j]):   } Compare
      count++
      if (count > 1):                         } It needs to change more than one element.
        return false
      else:                                   } Optimize the element that is going to change.
        x=j                                       (coordinate)
        y=i
        if (j != 0):
          x++
        if (i == m-2):
          y++
if (count ==1):                               } change it, calculate a new value.
    change(array, x,y)
    return True
else:
    return false
```

c) I construct a subarray A' of array A which consisting of the even-numbered rows of array A. Recursively find the leftmost minimum for each row in A'. Then find the leftmost minimum in the odd-numbered rows of A.



– Output of mxn array is an array with size m. Each index value represent the index leftmost minimum of this row.

d) $T(m) = T(m/2) + cn + dm$

↳ Divide by two (even_odd)

↳ merging odd numbered with even numbered

② My solution is dividing both array by k/2, and repeat it.

```
def kth_element(array_1, array_2, k, p_1 = 0, p_2 = 0):

    len_1 = len(array_1)
    len_2 = len(array_2)

    if(p_1 == len_1):
        return array_2[p_2 + k - 1]          # if we reach the end of the array-1, then return the array-2's next element

    if(p_2 == len_2):
        return array_1[p_1 + k - 1]          # if we reach the end of the array-2, then return the array-1's next element

    if(k == 0 or k > (len_1 - p_1) + (len_2 - p_2)):   # invalid k value, k=0 or k>m+n
        return -1

    if(k == 1):
        if (array_1[p_1] < array_2[p_2]):
            return array_1[p_1]

        else:
            return array_2[p_2]              # if k=1, then compare first elements, return the smaller one.

    current = k//2

    if(current - 1 >= len_1 - p_1):          # Size of array-1 is smaller than k/2
        if(array_1[len_1 - 1] < array_2[p_2 + current - 1]):   # If last element of array-1 is not kth
            return array_2[p_2 + (k - (len_1 - p_1) - 1)]

        else:
            return kth_element(array_1, array_2, k - current, p_1, p_2 + current)

    if(current - 1 >= len_2 - p_2):          # Size of array-2 is smaller than k/2
        if(array_2[len_2 - 1] < array_1[p_1 + current - 1]):   # If last element of array-2 is not kth
            return array_1[p_1 + (k - (len_2 - p_2) - 1)]

        else:
            return kth_element(array_1, array_2, k - current, p_1 + current, p_2)

    else:                                    # Move pointer of one array k/2 to the right
        if(array_1[current + p_1 - 1] < array_2[current + p_2 - 1]):
            return kth_element(array_1, array_2, k - current, p_1 + current, p_2)

        else:
            return kth_element(array_1, array_2, k - current, p_1, p_2 + current)
```

Its worst case complexity = $O(\log k)$

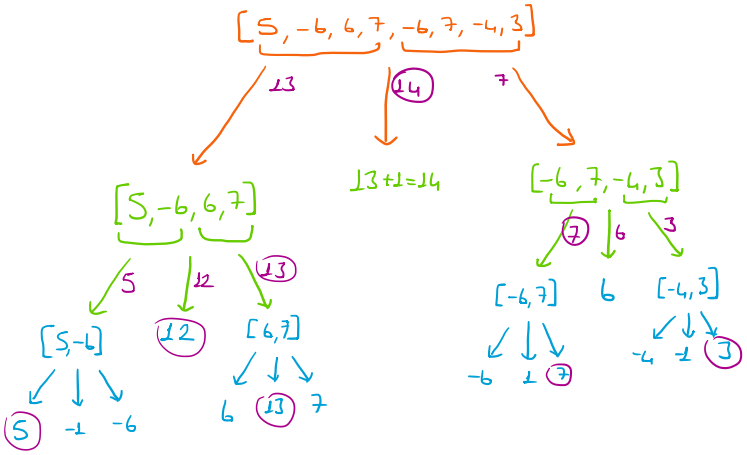
k can be maximum (m+n) where m and n are sizes of arrays.

It is because: In every recursive step m+n decreases by $\frac{m+n}{2}$.

③

My solution is dividing the array into two half recursively. In every recursive step I find the maximum sum of left and right subarrays. And also I find the crossing sum of left and right. In each step I return the start and end indexes of subarrays. So I can reach the contingious subarray directly.

Example:



Time complexity: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$

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

$\Theta(n \log n)$ → from Master Theorem, (case 2)

④

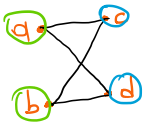
I created a basic graph class to represent my graphs easily.

My approach to solution is coloring the vertexes. I have two colors 1 and 0.

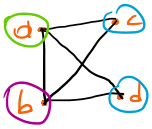
If I can color all vertexes by using only these two colors, it means the graph is bipartite.

Coloring has only one rule: Adjacent vertexes can not have same color.

Example:



→ It's a bipartite graph



→ It's not a bipartite graph. We need a third color.

Time complexity: $O(n)$

- In every recursive step vertex count decrease by one. So we have to traverse all vertexes to color them.

⑤ If we need to simplify the problem, it is a finding maximum problem. We also have to calculate gain from cost and price lists. I divede the array into two subarrays recursively.

Time complexity: $T(n) = 2T(n/2) + 1$

$T(n) = \frac{3n}{2} - 1$

$O(n)$