Date: 5th September, 2021

Design and Analysis of Algorithm Lab

2019BTECS00058 Devang
Batch: T7

Assignment: Week 3
Divide and Conquer Strategy

Q1) Implement an algorithm to find the maximum element in an array which is first increasing and then decreasing, with Time Complexity O(Logn).

Algorithm:

1. We use a variation of the Binary Search to achieve the search in O(log N)
2. We look at the middle element of the list. If it's larger than the right and left sides, then we return it
3. If left of the middle element is smaller than the middle and right of the middle is larger than the middle, we recurse through the right side of the list
4. If left of the middle element is larger than the middle and right of the middle is smaller than the middle, we recurse through the left side of the list
5. Base case is if we have 1 element - we return that or we return larger of the 2 in case of 2

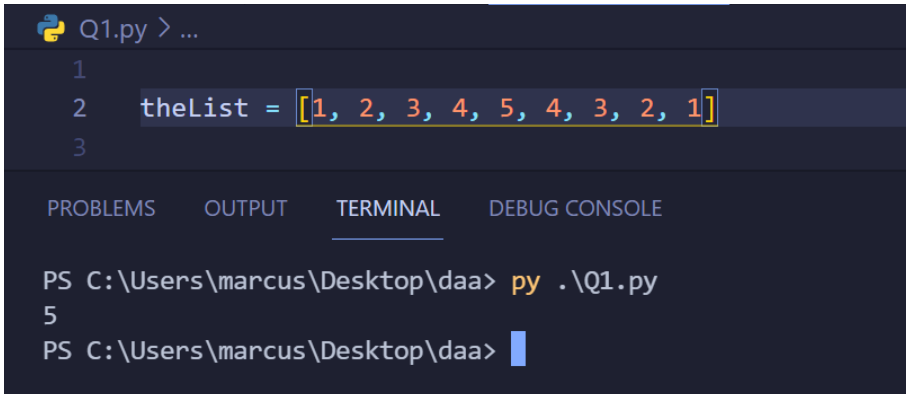
Program:

```
theList = [3, 4, 5, 6, 7, 8, 9, 5, 3, 1]

# Modified version of the binary search
def findMaximumElement(theList, low, high):

    # Base Case
```

```python
    if low == high:
        return theList[low]

    # 2 elements (One of them would be the maximum)
    if high - low == 1:
        if theList[low] > theList[high]:
            return theList[low]
        if theList[low] < theList[high]:
            return theList[high]

    # General Case
    mid = (low + high)//2

    # The Peak
    if theList[mid] > theList[mid+1] and theList[mid] > theList[mid-1]:
        return theList[mid]

    # Peak lies to the left
    if theList[mid] < theList[mid-1] and theList[mid] > theList[mid+1]:
        return findMaximumElement(theList, low, mid-1)

    # Peak lies to the right
    if theList[mid] > theList[mid-1] and theList[mid] < theList[mid+1]:
        return findMaximumElement(theList, mid+1, high)

print(findMaximumElement(theList, 0, len(theList)-1))
```

Output:

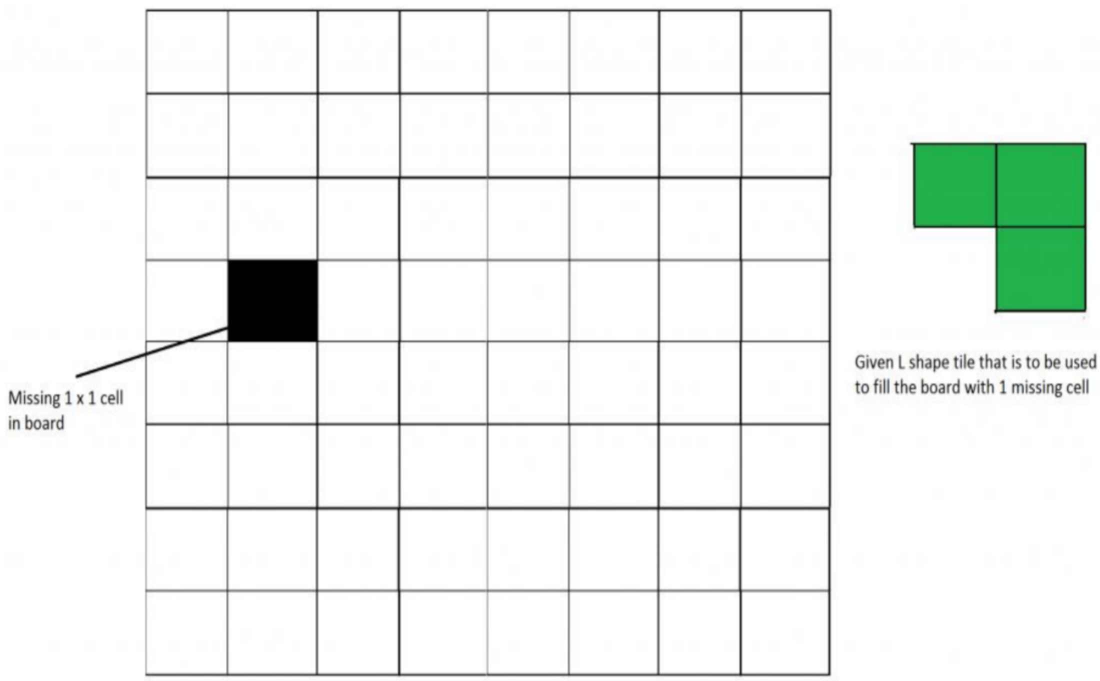
```
Q1.py > ...

theList = [3, 4, 5, 6, 7, 8, 9, 5, 3, 1]

PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

PS C:\Users\marcus\Desktop\daa> py .\Q1.py
9
PS C:\Users\marcus\Desktop\daa>
```



Complexity of proposed algorithm (Time & Space):
Space Complexity: O(1)
Time Complexity: O(Log N)

Your comment (How is your solution optimal?)
The algorithm is very optimised since it does not use any extra space and the complexity is pretty much perfect. It covers all edge-cases and is very scalable. A variation of Binary Search.

Q2) Implement algorithm for Tiling problem: Given an n by n board where n is of form 2^k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell (of size 1 x 1). Fill the board using L shaped tiles. An L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.



Algorithm:

1. We solve by recursive division into 4 parts using the divide and conquer algorithm
2. We solve the base case - if its a 2*2 square - we change all 0's to the L count
3. We add an L to the shape to the 3 squares that don't contain a -1
4. The recursive step is to divide the square into further smaller squares

Program:

```
gridSize = 2**4
countOfL = 0

thePointX = 3
thePointY = 4

initialList = [[0 for i in range(gridSize)] for j in range(gridSize)]
```

```
initialList[thePointX][thePointY] = -1

def makeAnL(lst, x1, y1, x2, y2, x3, y3):
    global countOfL
    countOfL += 1
    lst[x1][y1] = countOfL
    lst[x2][y2] = countOfL
    lst[x3][y3] = countOfL
    return lst

def buildLTiles(theMatrix, n, pointX, pointY):
    global countOfL
    r = 0
    c = 0

    # Base case
    # For the 2*2 square -> make every 0 -> count and that one -1 remains
    if n==2:
        countOfL += 1
        for i in range(n):
            for j in range(n):
                if theMatrix[pointX + i][pointY + j] == 0:
                    theMatrix[pointX + i][pointY + j] = countOfL
        return theMatrix

    for i in range(pointX, pointX + n):
        for j in range(pointY, pointY + n):
            if (theMatrix[i][j] != 0):
                r = i
                c = j

    # Placing an L in square such that 3 parts without the point will get
the L
    if (r < pointX + n // 2 and c < pointY + n // 2):
        makeAnL(theMatrix, pointX + int(n / 2), pointY + int(n / 2) - 1,
pointX + int(n / 2), pointY + int(n / 2), pointX + int(n / 2) - 1, pointY
+ int(n / 2))
    elif(r >= pointX + int(n / 2) and c < pointY + int(n / 2)):
```

```
        makeAnL(theMatrix, pointX + int(n / 2) - 1, pointY + int(n / 2), pointX + int(n / 2), pointY + int(n / 2), pointX + int(n / 2) - 1, pointY + int(n / 2) - 1)
    elif(r < pointX + int(n / 2) and c >= pointY + int(n / 2)):
        makeAnL(theMatrix, pointX + int(n / 2), pointY + int(n / 2) - 1, pointX + int(n / 2), pointY + int(n / 2), pointX + int(n / 2) - 1, pointY + int(n / 2) - 1)
    elif(r >= pointX + int(n / 2) and c >= pointY + int(n / 2)):
        makeAnL(theMatrix, pointX + int(n / 2) - 1, pointY + int(n / 2), pointX + int(n / 2), pointY + int(n / 2) - 1, pointX + int(n / 2) - 1, pointY + int(n / 2) - 1)

    # Recurse the 4 parts of the square
    theMatrix = buildLTiles(theMatrix, int(n / 2), pointX, pointY + int(n / 2))
    theMatrix = buildLTiles(theMatrix, int(n / 2), pointX, pointY)
    theMatrix = buildLTiles(theMatrix, int(n / 2), pointX + int(n / 2), pointY)
    theMatrix = buildLTiles(theMatrix, int(n / 2), pointX + int(n / 2), pointY + int(n / 2))

    return theMatrix

result = buildLTiles(initialList, gridSize, 0, 0)

for x in result:
    for y in x:
        print(y, end=' ')
    print()
```

Output:

```
gridSize = 2**3
countOfL = 0

thePointX = 0
thePointY = 0
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\marcus\Desktop\daa> py .\Q2.py
-1 9 8 8 4 4 3 3
9 9 7 8 4 2 2 3
10 7 7 11 5 5 2 6
10 10 11 11 1 5 6 6
14 14 13 1 1 19 18 18
14 12 13 13 19 19 17 18
15 12 12 16 20 17 17 21
15 15 16 16 20 20 21 21
PS C:\Users\marcus\Desktop\daa>
```

```
gridSize = 2**4
countOfL = 0

thePointX = 3
thePointY = 4
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
IndexError: list index out of range
PS C:\Users\marcus\Desktop\daa> py .\Q2.py
31 31 30 30 26 26 25 25 10 10 9 9 5 5 4 4
31 29 29 30 26 24 24 25 10 8 8 9 5 3 3 4
32 29 33 33 27 27 24 28 11 8 12 12 6 6 3 7
32 32 33 23 -1 27 28 28 11 11 12 2 2 6 7 7
36 36 35 23 23 41 40 40 15 15 14 14 2 20 19 19
36 34 35 35 41 41 39 40 15 13 13 14 20 20 18 19
37 34 34 38 42 39 39 43 16 16 13 17 21 18 18 22
37 37 38 38 42 42 43 43 1 16 17 17 21 21 22 22
52 52 51 51 47 47 46 1 1 73 72 72 68 68 67 67
52 50 50 51 47 45 46 46 73 73 71 72 68 66 66 67
53 50 54 54 48 45 45 49 74 71 71 75 69 69 66 70
53 53 54 44 48 48 49 49 74 74 75 75 65 69 70 70
57 57 56 44 44 62 61 61 78 78 77 65 65 83 82 82
57 55 56 56 62 62 60 61 78 76 77 77 83 83 81 82
58 55 55 59 63 60 60 64 79 76 76 80 84 81 81 85
58 58 59 59 63 63 64 64 79 79 80 80 84 84 85 85
PS C:\Users\marcus\Desktop\daa>
```

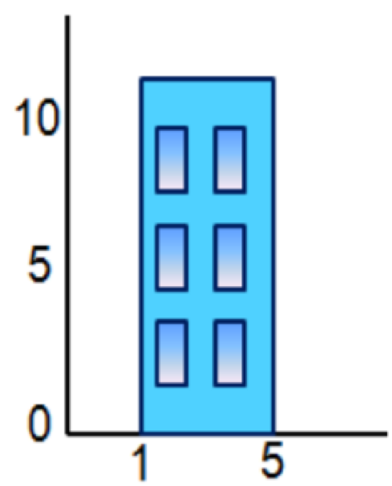
Complexity of proposed algorithm (Time & Space):
Space Complexity: O(1)
Time Complexity: O(N Log N)

Your comment (How is your solution optimal?)
The solution is robust and scalable due to its divide and conquer template.

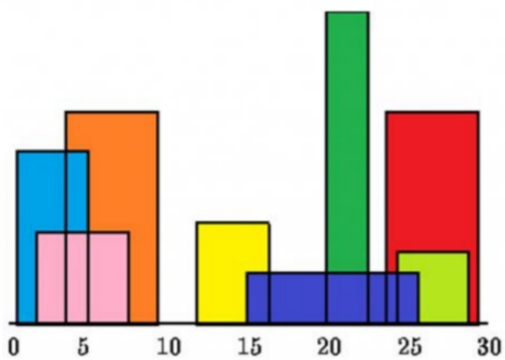
Q3) Implement algorithm for The Skyline Problem: Given n rectangular buildings in a 2- dimensional city, compute the skyline of these buildings, eliminating hidden lines. The main task is to view buildings from a side and remove all sections that are not visible.

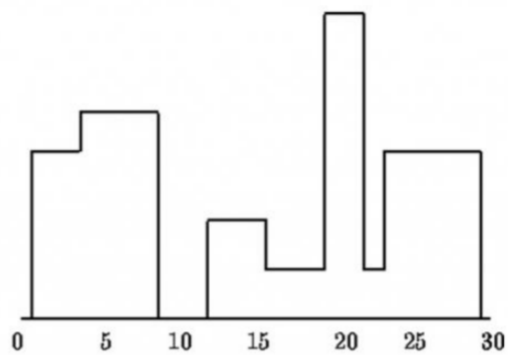
All buildings share common bottom and every building is represented by triplet (left, ht, right):
'left': is the x coordinate of the left side (or wall).
'right': is the x coordinate of the right side.
'ht': is the height of a building.

For example, the building on right side is represented as (1, 11, 5)



A skyline is a collection of rectangular strips. A rectangular strip is represented as a pair (left, ht) where the left is the x coordinate of the left side of the strip and ht is height of strip.
With Time Complexity O(nLogn)





Algorithm:

1. We solve using the recursive divide and conquer algorithm
2. We build the base case - that if n is 0, we return an empty list or if n = 1, we return the list with limits of the one building
3. We recurse of the left and right side of skylines and update the limits with the maximum of the values of key-points encountered

Program:

```
listOfBuildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

def update_output(output, x, y):
    # Update the final output with the new element.
    # if skyline change is not vertical - add the new point
    if not output or output[-1][0] != x:
        output.append([x, y])
    # if skyline change is vertical - update the last point
    else:
        output[-1][1] = y
```

```python
def append_skyline(output, p, lst, n, y, curr_y):
    # Append the rest of the skyline elements with indice (p, n) to the
final output.
    while p < n:
        x, y = lst[p]
        p += 1
        if curr_y != y:
            update_output(output, x, y)
            curr_y = y

def mergeSkylines(left, right):
    n_l = len(left)
    n_r = len(right)
    p_l = p_r = 0
    curr_y  = left_y = right_y = 0
    output = []

    # while we're in the region where both skylines are present
    while p_l < n_l and p_r < n_r:
        point_l, point_r = left[p_l], right[p_r]
        # pick up the smallest x
        if point_l[0] < point_r[0]:
            x, left_y = point_l
            p_l += 1
        else:
            x, right_y = point_r
            p_r += 1
        # max height (i.e. y) between both skylines
        max_y = max(left_y, right_y)
        # if there is a skyline change
        if curr_y != max_y:
            update_output(output, x, max_y)
            curr_y = max_y

    # there is only left skyline
    append_skyline(output, p_l, left, n_l, left_y, curr_y)

    # there is only right skyline
    append_skyline(output, p_r, right, n_r, right_y, curr_y)
```

```python
    return output

def getSkyline(buildings):
    n = len(buildings)

    # Base Case
    if n == 0:
        return []
    if n == 1:
        xStart, xEnd, y = buildings[0]
        return [[xStart, y], [xEnd, 0]]

    # For 2 or more buildings, we recurse to 2 subproblems
    leftSkyline = getSkyline(buildings[:n//2])
    rightSkyline = getSkyline(buildings[n//2:])

    return mergeSkylines(leftSkyline, rightSkyline)

print(getSkyline(listOfBuildings))
```

Output:

```python

listOfBuildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\marcus\Desktop\daa> py .\Q3.py
[[2, 10], [3, 15], [7, 12], [12, 0], [15, 10], [20, 8], [24, 0]]
PS C:\Users\marcus\Desktop\daa>
```

Complexity of proposed algorithm (Time & Space):
Space Complexity: O(N)
Time Complexity: O(N (log N))

Your comment (How is your solution optimal?)
Using the Divide and Conquer - the problem is optimised to solve in O(n log N) compared to the brute force of $O(n^2)$. Therefore it is efficient with respect to space and time.