Name:Vishal Chauhan

PRN:2020BTECS00090

Ass no:3

Course: Design and analysis of algorithm Lab

Batch:T6 Class:TYCSE,WCE

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

ALGORITHM:

1. Used the divide and conquer strategy and divide the array into two equal parts andcalculate the middle element.
2. Check if the middle element is greater than its both adjacent elements if the conditionis statisfied then print the element.
3. If the middle element is greater than the left element then repeat the steps 1 and 2 inthe left half
    - Else apply the steps 1 and

      2 in the right half

```
// Q1) Implement algorithm to Find the maximum element in an array which is first
increasing and then decreasing,
//  with Time Complexity O(Logn).
#include <iostream>
#include <vector>
using namespace std;
int maximum_using_DAC(vector<int> arr, int l, int r)
{
    while (l < r)
    {
        int mid = l + (r - l) / 2;
        if ((arr[mid] > arr[mid - 1]) && (arr[mid] > arr[mid + 1]))
        {
            return arr[mid];
        }
        else
        {
            // left movement
            if ((arr[mid - 1] > arr[mid]) && (arr[mid] > arr[mid + 1]))
            {
                r = mid - 1;
            }

            // right movement
            else
            {
                l = mid + 1;
            }
        }
    }
    return -2;
}
int main()
{
    vector<int> arr{23, 34, 39, 46, 58, 75, 80,90,100,120,150,200, 40, 20, 10,
5};
    cout << "maximum ele are"
         << ":" << maximum_using_DAC(arr, 0, arr.size() - 1);
}
```

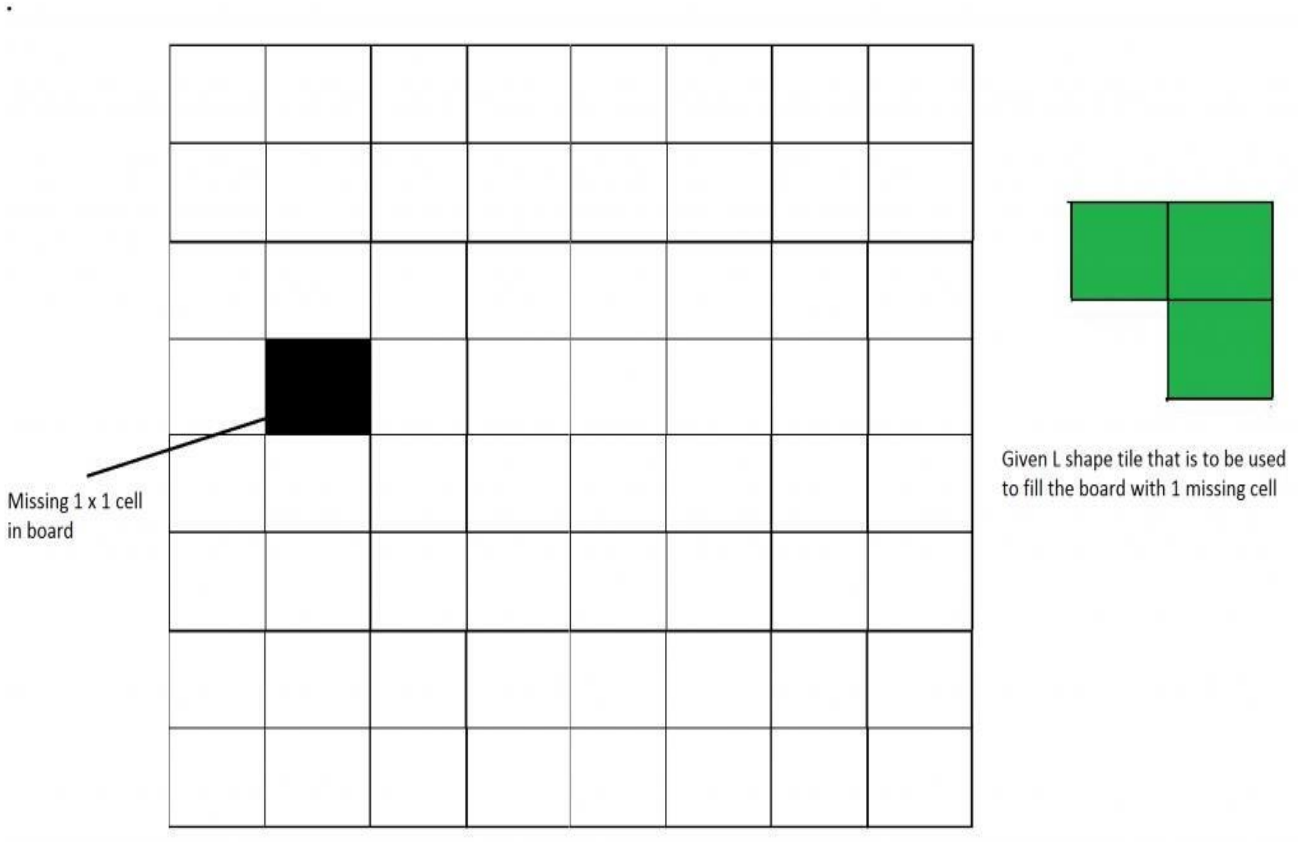
```
OUTPUT:
PS C:\Users\Dell\OneDrive\Desktop\DAA\Ass-1> cd
"c:\Users\Dell\OneDrive\Desktop\DAA\" ; if ($?) { g++ q1.cpp -o q1 } ; if ($?) {
.\q1 }

maximum ele are:200
```

```
complexity Analyse:
TC-Log(n) for binary searching
SC-O(1)
```

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 *1x 1*). Fill the board using L shaped tiles. An *L* shaped tile is a *2 x 2* square with one


Missing 1 x 1 cell in board
Given L shape tile that is to be used to fill the board with 1 missing cell

cell of size *1×1* missing

ALGORITHM:-

1. Input the value of n and the board.

2. Call the recursive function "func", which fills the board. In the function "func":
3. Declare variables that will store the row number and column number of the missing cell

4. Write the base case. The base case is that if n==2, we place an L in the 2x2 board, such that itcovers all the three cells which are not filled. We will fill the board with a new tile, thus increasing the tile number by 1. Fill all the tiles other than the missing tile.

5. Else, find the missing cell's row and column. The cell whose value is -1 is the missing cell.

6. Now, since we've found the missing cell, we need to fill the other three sub-board

quadrantswhich don't have the missing cell using an L. For doing this, we have another "fill" function.

7. If the missing tile is in the 1st quadrant, call the fill function for the second, third, and fourthquadrants. This "fill" function fills the corners of the three quadrants using an L. Thus, increase the tile number each time we call it.
8. If the missing tile is in the third quadrant, call the fill function for the second, first, and fourthquadrants.

9. If the missing tile is in the second quadrant, call the fill function for the first, third, and fourthquadrants.

10. If the missing tile is in the fourth quadrant, call the fill function for the second, third, and firstquadrants.

11. Now that we have four sub-boards, call the function "func" for the four sub-boards.

12. Print the board in the end.

```
#include <bits/stdc++.h>
using namespace std;

int tile_number = 0;

void fill(int x1, int y1, int x2, int y2, int x3, int y3, vector<vector<int>>
&board)
{
    tile_number++;
    board[x1][y1] = tile_number;
    board[x2][y2] = tile_number;
    board[x3][y3] = tile_number;
}

void func(int n, int r, int c, vector<vector<int>> &board)
{
    int missing_cell_row, missing_cell_col;
    if (n == 2)
    {
        tile_number++;
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (board[r + i][c + j] == 0)
                {
                    board[r + i][c + j] = tile_number;
                }
            }
        }
        return;
    }
    else
    {
        for (int i = r; i < r + n; i++)
        {
            for (int j = c; j < c + n; j++)
            {
                if (board[i][j] != 0)
                    missing_cell_row = i, missing_cell_col = j;
            }
        }
    }
    if (missing_cell_row < r + n / 2 && missing_cell_col < c + n / 2)
    {
        fill(r + n / 2, c + (n / 2) - 1, r + n / 2, c + n / 2, r + n / 2 - 1, c +
n / 2, board);
    }

    else if (missing_cell_row >= r + n / 2 && missing_cell_col < c + n / 2)
    {
        fill(r + (n / 2) - 1, c + (n / 2), r + (n / 2), c + n / 2, r + (n / 2) -
1, c + (n / 2) - 1, board);
    }

    else if (missing_cell_row < r + n / 2 && missing_cell_col >= c + n / 2)
    {
        fill(r + n / 2, c + (n / 2) - 1, r + n / 2, c + n / 2, r + n / 2 - 1, c +
n / 2 - 1, board);
    }

    else if (missing_cell_row >= r + n / 2 && missing_cell_col >= c + n / 2)
    {
        fill(r + (n / 2) - 1, c + (n / 2), r + (n / 2), c + (n / 2) - 1, r + (n /
2) - 1, c + (n / 2) - 1, board);
    }

    func(n / 2, r, c + n / 2, board);
    func(n / 2, r, c, board);
    func(n / 2, r + n / 2, c, board);
    func(n / 2, r + n / 2, c + n / 2, board);

    return;
}

int main()
{
    int n = 4;
    vector<vector<int>> board = {
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, -1, 0}};

    func(n, 0, 0, board);

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

```
OUTPUT:
3 3 2 2
3 1 1 2
4 1 5 5
4 4 -1 5
```

```
// TIME AND SPACE COMPLEXITY:-
1)TC=O(N^2)
2)SC=O(1) and used only constant space only
```

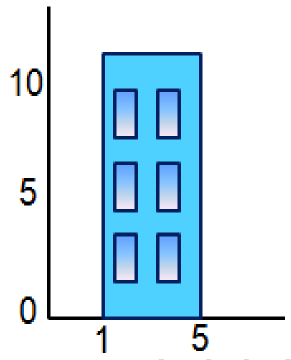
Q3) Implement algorithm for The Skyline Problem: Given n rectangular buildings in a 2- dimensional city, computes the skyline of these buildings, eliminating hidden lines.

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)

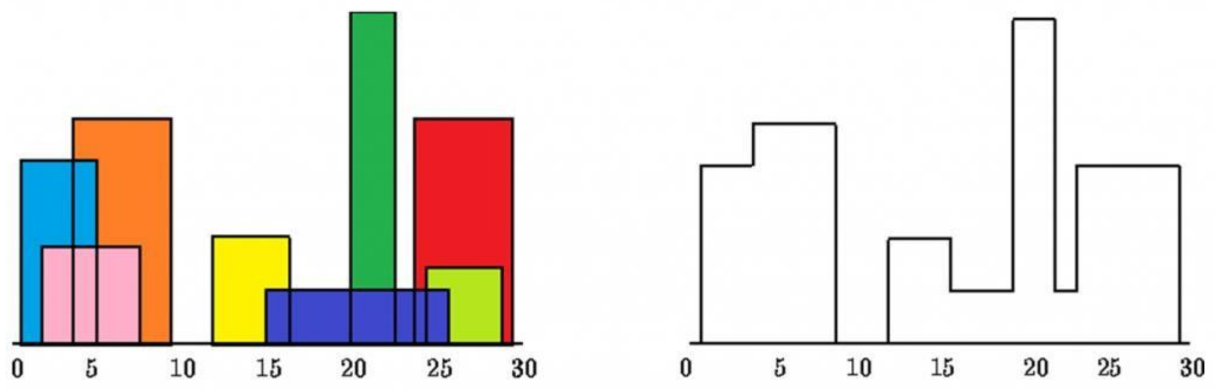
‘left’: is x coordinated of left side (or wall).‘right': is x coordinate of right side

‘ht': is height of 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 left is x coordinate of left side of strip and ht is height of strip.



With Time Complexity *O(nLogn)*

ALGORITHM:-

Height of new Strip is always obtained by takin maximum of following

(a) Current height from skyline1, say 'h1'.

(b) Current height from skyline2, say 'h2'

h1 and h2 are initialized as 0. h1 is updated when a strip from SkyLine1 is added to result and h2 is updated when a strip fromSkyLine2 is added.

Skyline1 = {(1, 11), (3, 13), (9, 0), (12, 7), (16, 0)}

Skyline2 = {(14, 3), (19, 18), (22, 3), (23, 13), (29, 0)}

Result
= {}
h1 = 0,
h2 = 0

Compare (1, 11) and (14, 3). Since first strip has smaller left x,add it to result and increment index for Skyline1.

h1 = 11, New Height =
max(11, 0)Result = {(1, 11)}

Compare (3, 13) and (14, 3). Since first strip has smaller left x,add it to result and increment index for Skyline1

h1 = 13, New Height =
max(13, 0)Result = {(1, 11),
(3, 13)}

Similarly (9, 0) and (12, 7) are added.h1 = 7, New Height = max(7, 0) = 7

Result = {(1, 11), (3, 13), (9, 0), (12, 7)}

Compare (16, 0) and (14, 3). Since second strip has smaller left x,it is added to result.

h2 = 3, New Height = max(7, 3) = 7

Result = {(1, 11), (3, 13), (9, 0), (12, 7), (14, 7)}

Compare (16, 0) and (19, 18). Since first strip has smaller left x,it is added to result.

h1 = 0, New Height = max(0, 3) = 3

Result = {(1, 11), (3, 13), (9, 0), (12, 7), (14, 7), (16, 3)}

Since Skyline1 has no more items, all remaining items of Skyline2are added

Result = {(1, 11), (3, 13), (9, 0), (12, 7), (14, 7), (16, 3), (19, 18), (22, 3), (23, 13), (29, 0)}

One observation about above output is, the strip (14, 7) is redundant (There is already an strip of same height). We remove all redundant strips.

Result = {(1, 11), (3, 13), (9, 0), (12, 7), (16, 3), (19, 18), (22, 3), (23, 13), (29, 0)}

```
#include <iostream>
using namespace std;
```

```
// A structure for building
struct Building
{
    // x coordinate of left side
    int left;

    // height
    int ht;

    // x coordinate of right side
    int right;
};

// A strip in skyline
class Strip
{
    // x coordinate of left side
    int left;

    // height
    int ht;

public:
    Strip(int l = 0, int h = 0)
    {
        left = l;
        ht = h;
    }
    friend class SkyLine;
};

// Skyline: To represent Output(An array of strips)
class SkyLine
{
    // Array of strips
    Strip *arr;

    // Capacity of strip array
    int capacity;

    // Actual number of strips in array
    int n;

public:
    ~SkyLine() { delete[] arr; }
    int count() { return n; }

    // A function to merge another skyline
    // to this skyline
    SkyLine *Merge(SkyLine *other);

    // Constructor
    SkyLine(int cap)
    {
        capacity = cap;
        arr = new Strip[cap];
        n = 0;
    }

    // Function to add a strip 'st' to array
    void append(Strip *st)
    {
        // Check for redundant strip, a strip is
        // redundant if it has same height or left as previous
        if (n > 0 && arr[n - 1].ht == st->ht)
            return;
        if (n > 0 && arr[n - 1].left == st->left)
        {
            arr[n - 1].ht = max(arr[n - 1].ht, st->ht);
            return;
        }

        arr[n] = *st;
        n++;
    }

    // A utility function to print all strips of
    // skyline
    void print()
    {
        for (int i = 0; i < n; i++)
        {
            cout << " (" << arr[i].left << ", "
                 << arr[i].ht << "), ";
        }
    }
};

// This function returns skyline for a
```

```cpp
// given array of buildings arr[l..h].
// This function is similar to mergeSort().
SkyLine *findSkyline(Building arr[], int l, int h)
{
    if (l == h)
    {
        SkyLine *res = new SkyLine(2);
        res->append(
            new Strip(
                arr[l].left, arr[l].ht));
        res->append(
            new Strip(
                arr[l].right, 0));
        return res;
    }

    int mid = (l + h) / 2;

    // Recur for left and right halves
    // and merge the two results
    SkyLine *sl = findSkyline(arr, l, mid);
    SkyLine *sr = findSkyline(arr, mid + 1, h);
    SkyLine *res = sl->Merge(sr);

    // To avoid memory leak
    delete sl;
    delete sr;

    // Return merged skyline
    return res;
}

// Similar to merge() in MergeSort
// This function merges another skyline
// 'other' to the skyline for which it is called.
// The function returns pointer to the
// resultant skyline
SkyLine *SkyLine::Merge(SkyLine *other)
{
    // Create a resultant skyline with
    // capacity as sum of two skylines
    SkyLine *res = new SkyLine(
        this->n + other->n);

    // To store current heights of two skylines
    int h1 = 0, h2 = 0;

    // Indexes of strips in two skylines
    int i = 0, j = 0;
    while (i < this->n && j < other->n)
    {
        // Compare x coordinates of left sides of two
        // skylines and put the smaller one in result
        if (this->arr[i].left < other->arr[j].left)
        {
            int x1 = this->arr[i].left;
            h1 = this->arr[i].ht;

            // Choose height as max of two heights
            int maxh = max(h1, h2);

            res->append(new Strip(x1, maxh));
            i++;
        }
        else
        {
            int x2 = other->arr[j].left;
            h2 = other->arr[j].ht;
            int maxh = max(h1, h2);
            res->append(new Strip(x2, maxh));
            j++;
        }
    }

    // If there are strips left in this
    // skyline or other skyline
    while (i < this->n)
    {
        res->append(&arr[i]);
        i++;
    }
    while (j < other->n)
    {
        res->append(&other->arr[j]);
        j++;
    }
    return res;
}

// Driver Function
```

```
int main()
{
    Building arr[] = {
        {1, 11, 5}, {2, 6, 7}, {3, 13, 9}, {12, 7, 16}, {14, 3, 25}, {19, 18,
22}, {23, 13, 29}, {24, 4, 28}};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Find skyline for given buildings
    // and print the skyline
    SkyLine *ptr = findSkyline(arr, 0, n - 1);
    cout << " Skyline for given buildings is \n";
    ptr->print();
    return 0;
}
```

```
OUTPUT:
Skyline for given buildings is
 (1, 11),  (3, 13),  (9, 0),  (12, 7),  (16, 3),  (19, 18),  (22, 3),  (23, 13),
(29, 0),
```

```
TIME AND SPACE COMPLEXITY:-
The time complexity of the above algorithm is O(NlogN).
```