Project Report

1. Task 1

- Mathematical Recursive Formulation

  Let Side[i][j] be the side length of the maximum square whose top left corner is at (i, j) in the grid, then Side[i][j] = min(Side[i-1][j], Side[i-1][j-1], Side[i][j-1]) + 1 if H[i][j] = C, and Side[i][j] = 0 if H[i][j] != C. Boundary conditions are skipped here which are easy to check.

- Correctness:

  Let Square[i][j] be the maximum square whose top let corner is at (i, j) in the grid. If H[j][j] != C, then Side[i][j] = 0. So next we only consider the case that H[i][j] = C.

  First, the top left corner of Square[i-1][j-1] cannot be to the left of the top left corner of Square[i] [j]. That is, Side[i-1][j-1] +1 <= Side[i][j]. Again the top bottom corner of Square[i-1] [j] must be not higher than the bottom left corner of Square[i][j]. So side[i-1][j] + 1 <=Side[i][j] too. Similarly, the top left corner of Square[i][j-1] must be not to the right of the top left corner of Square[i][j], which means Side[i][j-1] + 1 >= Side[i][j]. Now we can have the following:

  Side[i][j] >= min(Side[i-1][j], Side[i-1][j-1], Side[i][j-1]) + 1

  On the other hand, there are a square whose top left corner is at (i-1, j-1) and side length = min(Side[i-1][j], Side[i-1][j-1], Side[i][j-1]), a rectangle whose top left corner is at (i-1, j) with width = min(Side[i-1][j], Side[i-1][j-1], Side[i][j-1]) and height = 1, and a rectangle whose top left corner is at (i, j-1) with width =1 and height = min(Side[i-1][j], Side[i-1][j-1], Side[i][j-1]). Those two rectangles and one square consist of square whose top left corner is at (i, j) with side length min(Side[i-1][j], Side[i-1][j-1], Side[i][j-1]) + 1.

  So now we can conclude that

  Side[i][j] = min(Side[i-1][j], Side[i-1][j-1], Side[i][j-1]) + 1.

- Time and Space Complexity:

  Since the calculation of Side[i][j] only depends on Side[i-1][j], Side[i-1][j-1], and S[i][j-1], it takes O(1) time to calculate Side[i][j]. There are NxM such Side[i][j] since 0<=i<N and 0 <= j< M. So the total time is Theta(NM). Note that we need store the maximum Side[i][j] for each row i.

  The algorithm implementation can only use two arrays Side1 and Side2 which store Side[i-1][j] and Side[i][j] where 0<=j<M correspondingly. So the total space is O(M).

2. Task 2

- Mathematical Recursive Formulation

  Let Count[i][j] be the number of consecutive cells in row i, i-1, …, i – Count[i][j] + 1 such that Height[i][j] = C but Height[i – Count[i][j]] != C. Then we have

  Count[i][j] = Count[i - 1][j] + 1 if H(i, j) = C

  = 0 if H(i, j) != C

  Let R[i][j] be the maximum rectangle which includes cell (i, j) as the top boundary with height = Count[i][j]. R[i][j] = the rectangle from cells (i-1, j), (i-2, j) ..(i-k, j), (i+1, j), (i+2, j), (i+m, j) such that all those cells have Count() >= Count(i, j) , but Count[i– k -1][j] < Count[i][j] and Count[i+m+1][j]<Count[i][j]. Boundary conditions are skipped here which are easy to check.

- Correctness

  Obviously the calculation of Count[i][j] is correct. Then the correctness follows from (1) for any maximum rectangle at cell (i, j), our algorithm finds it when calculate R[i][j], and (2) our calculation of R[i][j] finds a rectangle.

- Time and Space Complexity

  The calculation of Count[i][j] takes time Theta(NM), while the calculation of R[i][j] takes time Theta(NM^2) since there are NM cells and each cell need Theta(M) time to calculate R[i][j]. So the time complexity is Theta(NM^2).
  We only need to keep two rows of Count: Count[i][j] and Count [I-1][j] and one row for R[i][j] where 0<=j<M. So the space complexity is O(M).

3. Task 3

- Mathematical Recursive Formulation

  Let Count[i][j] be the number of consecutive cells in row i, i-1, …, i – Count[i][j] + 1 such that Height[i][j] = C but Height[i – Count[i][j]] != C. Then we have

  Count[i][j] = Count[i - 1][j] + 1 if H(i, j) = C
  = 0 if H(i, j) != C

  To find the maximum rectangle faster, we use a stack S to store Count[i][j] (0<=j < M) when we process Count[i] for row i such that the stack elements are in ascending order. That is, the top of the stack is always the one with the maximum Count(). When a cell Count[i][k] is smaller than the stack top element, then stack elements are popped until stack stop is smaller than or equal to Count[i][k], and if after that the stack S is empty, then push k into S.

- Correctness

  Obviously the calculation of Count[i][j] is correct. Now we show that our algorithm correctly finds the maximum rectangle by using the stack S.

  When Count[i][k] is processed, if it is greater than the Count[S.top()], then k is push into S. So the ascending order of stack elements is preserved. When Count[i][k] is smaller than or equal to Count[S.top()], then Count[i][k] is the right boundary of the maximum rectangle starts at S.top(). So the maximum rectangle starts at S.top() can be calculated as

  (k – S.top()) * Count[i][k]

  This concludes the correctness of our algorithm

- Time and Space Complexity

  The calculation of Count[i][j] takes time Theta(NM), while the calculation of R[i][j] takes time Theta(NM) since there are NM cells and all cells (i, j) where 0<=j < M for row i need Theta(M) time to calculate R[i][j] for all j (0<=j <M) in row i. This is because each i from 0 to M is pushed to stack S at most once. So the time complexity is Theta(NM).
  We only need to keep two rows of Count: Count[i][j] and Count [i-1][j], one row for R[i][j], and two stacks SL and SR which need at most O(M) space where 0<=i<N. So the space complexity is O(M).

Task 4:

- **Recursive Formulation**

Mathematical Recursive Formulation

Let S[i] be the highest row indices of consecutive cells in row i, i-1, …, i – Count[i][j] + 1 such that abs(Height[i][j] - H[i’][j])<=C. Then we have

S[i][c] = max(t) s.t. c = UPPER_LIMIT[j] – H[i][t]

Let R[i][j] be the maximum rectangle which includes cell (i, j) as the bottom right boundary with height = start_idx[j]. R[i][j] = the rectangle from cells (i-1, j), (i-2, j) ..(i-k, j), (i+1, j), (i+2, j), (i+m, j) such that all those cells are within C of each other via the merge operation done as part of the relaxation.

- Correctness:

Similar to Task 2, we maintain the longest contiguous column with values within C of each other, as a relaxation step combining the maximal area rectangle to find the end coordinates. The optimal substructure i.e. the longest column that does not violate the constraints (i.e. <=C) can be combined/relaxed to form rectangles thus systematic construction of maximal area rectangles becomes possible. Thus by maintaining the above mentioned recurrence we can eventually find the required maximal area rectangle.

- Time and Space Complexity:

  There are N rows, and M columns, looping over them C times, we get the required time complexity of Theta(N.M^2.C), space complexity remains M.C.