A standard problem
Solution
Momchil Ivanov

Given the problem constraints one can guess that the answer for each possible range of rows (i,j), (1<=i<=j<=N) should be precomputed before the queries, so that we can answer each of them with O(1) complexity.

Precomputing:

We would start processing the rows from top to bottom and for each chosen j (j would be the upper row bound, meaning the bigger) we would choose all possible lower bounds 1<=i<=j, starting from the top again.

Let dp[i][j] be the are of the biggest table we can fit between rows [i,j], then:

- dp[i][j] = max(dp[i][j-1], max(dp[i+1][j], A_{j-i+1})), where A_{j-i+1} is the are of the biggest table that fits between [i,j] doesn't contain ones and has height (j-i+1).

Given the order in which we are processing the ranges it follows that if we are computing dp[i][j], then we would have information for dp[i][j-1] already but not for dp[i+1][j]. This means that while we are processing all possible lower bounds (i from 1 to j) we would be computing dp[i][j] with the formula:

- dp[i][j] = max(dp[i][j-1], A_{j-i+1})

Once we have traversed all possible lower bounds i from top to bottom, we would traverse them again, but from bottom to top (i.e. starting from j, going to 1) in order to compare our current optimal result for dp[i][j] with dp[i+1][j], i.e. for each i we do:

- dp[i][j] = max(dp[i][j], dp[i+1][j])

The only thing that is left is to compute A_{j-i+1} for all i, where 1<=i<=j. In other words we need to compute the biggest tables of height in the range [1,(j-i+1)] that are lying on row j.

Computing A_{j-i+1}:

For each column of row j we would find how much can we go up until we reach a 1, let h[j] be the number of rows we can go up like this, including the row j. For example, given the table:

0 1 0 0
1 0 0 0
0 0 0 0

For the last row we have the following values of h: h[1] = 1, h[2] = 2, h[3] = h[4] = 3. For each h[k] we would compute the minimum p, where p<=k such that h[l] >= h[k], for all l in [p,k]. Let Left[k] = k-p+1, we define similarly Right[k] = p-k+1, where p is the maximum p (p>=k) such that h[l]>=h[k], for all l, which are in [k,p].

In the given example we have Left = {1,1,1,2}, Right = {4,3,2,1}.
Both Left[] and Right[] can be precomputed (separately) in O(n) complexity, using a stack structure. In it we would keep pairs(h,w), such that if one pair(h1,w1) is on the top of another pair(h2,w2) in the stack then h1 should be bigger than h2. This is how we would use such a stack to compute Left[], for example:

```
sz = -1;

for( int j = 1; j <= m; ++j ){



sum = 0;

while( sz >= 0 ){

  if( st[sz].first >= h[j] ) {sum += st[sz].second; sz--;}

  else break;

}



st[++sz] = make_pair(h[j], sum + 1);

Left[j] = sum + 1;



}
```

In the above C++ code h[] and Left[] are as explained above and st[] is an array of pairs used as a stack.
In a similar way can be computed Right[j] if we simply loop from j = m to 1.

Given Left[] and Right[] we want to compute A[len], where 1<=len<=(j-i+1), and A[len] gives the maximum width of a table with height len that does not contains ones inside and lies on the j^{th} row. It is not difficult to see that A[len] =

max(A[len+1], max(Left[k] + Right[k] - 1)), where max(Left[k]+Right[k]-1) is the maximum such sum for which h[k] = len.

Now that we have computed A[], A_{j-i+1} is simply equal to A[j-i+1]*(j-i+1).

Given the above analysis the solution of this problem should have a complexity of O(N*M). For clarifications please check the author's solution - standard.cpp.