Finding the largest zero submatrix

Table of Contents

You are given a matrix with `n` rows and `m` columns. Find the largest submatrix consisting of only zeros (a submatrix is a rectangular area of the matrix).

Algorithm

Elements of the matrix will be `a[i][j]`, where `i = 0...n - 1`, `j = 0... m - 1`. For simplicity, we will consider all non-zero elements equal to 1.

Step 1: Auxiliary dynamic

First, we calculate the following auxiliary matrix: `d[i][j]`, nearest row that has a 1 above `a[i][j]`. Formally speaking, `d[i][j]` is the largest row number (from `0` to `i - 1`), in which there is a element equal to `1` in the `j`-th column. While iterating from top-left to bottom-right, when we stand in row `i`, we know the values from the previous row, so, it is enough to update just the elements with value `1`. We can save the values in a simple array `d[i]`, `i = 1...m - 1`, because in the further algorithm we will process the matrix one row at a time and only need the values of the current row.

```cpp
vector<int> d(m, -1);
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
        if (a[i][j] == 1) {
            d[j] = i;
        }
    }
}
```

Step 2: Problem solving

We can solve the problem in $O(nm^2)$ iterating through rows, considering every possible left and right columns for a submatrix. The bottom of the rectangle will be the current row, and using `d[i][j]` we can find the top row. However, it is possible to go further and significantly improve the complexity of the solution.

It is clear that the desired zero submatrix is bounded on all four sides by some ones, which prevent it from increasing in size and improving the answer. Therefore, we will not miss the answer if we act as follows: for every cell `j` in row `i` (the bottom row of a potential zero submatrix) we will have `d[i][j]` as the top row of the current zero submatrix. It now remains to determine the optimal left and right boundaries of the zero submatrix, i.e. maximally push this submatrix to the left and right of the `j`-th column.

What does it mean to push the maximum to the left? It means to find an index `k1` for which `d[i][k1] > d[i][j]`, and at the same time `k1` - the closest one to the left of the index `j`. It is clear that then `k1 + 1` gives the number of the left column of the required zero submatrix. If there is no such index at all, then put `k1` = `-1`(this means that we were able to extend the current zero submatrix to the left all the way to the border of matrix `a`).

Symmetrically, you can define an index `k2` for the right border: this is the closest index to the right of `j` such that `d[i][k2] > d[i][j]` (or `m`, if there is no such index).

So, the indices `k1` and `k2`, if we learn to search for them effectively, will give us all the necessary information about the current zero submatrix. In particular, its area will be equal to `(i - d[i][j]) * (k2 - k1 - 1)`.

How to look for these indexes `k1` and `k2` effectively with fixed `i` and `j`? We can do that in $O(1)$ on average.

To achieve such complexity, you can use the stack as follows. Let's first learn how to search for an index `k1`, and save its value for each index `j` within the current row `i` in matrix `d1[i][j]`. To do this, we will look through all the columns `j` from left to right, and we will store in the stack only those columns that have `d[][]` strictly greater than `d[i][j]`. It is clear that when moving from a column `j` to the next column, it is necessary to update the content of the stack. When there is an inappropriate element at the top of the stack (i.e. `d[][] <= d[i][j]`) pop it. It is easy to understand that it is enough to remove from the stack only from its top, and from none of its other places (because the stack will contain an increasing `d` sequence of columns).

The value `d1[i][j]` for each `j` will be equal to the value lying at that moment on top of the stack.

The dynamics `d2[i][j]` for finding the indices `k2` is considered similar, only you need to view the columns from right to left.

It is clear that since there are exactly `m` pieces added to the stack on each line, there could not be more deletions either, the sum of complexities will be linear, so the final complexity of the algorithm is $O(nm)$.

It should also be noted that this algorithm consumes $O(m)$ memory (not counting the input data - the matrix `a[][]`).

Implementation

```
int zero_matrix(vector<vector<int>> a) {
    int n = a.size();
    int m = a[0].size();
    int ans = 0;
    vector<int> d(m, -1), d1(m), d2(m);
    stack<int> st;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (a[i][j] == 1)
                d[j] = i;
        }

        for (int j = 0; j < m; ++j) {
            while (!st.empty() && d[st.top()]
                st.pop();
            d1[j] = st.empty() ? -1 : st.top()
            st.push(j);
        }
        while (!st.empty())
            st.pop();

        for (int j = m - 1; j >= 0; --j) {
            while (!st.empty() && d[st.top()]
                st.pop();
            d2[j] = st.empty() ? m : st.top();
            st.push(j);
        }
        while (!st.empty())
            st.pop();

        for (int j = 0; j < m; ++j)
            ans = max(ans, (i - d[j]) * (d2[j]
    }
    return ans;
}
```