Нахождение наибольшей нулевой подматрицы

Дана матрица a размером $n \times m$. Требуется найти в ней такую подматрицу, состоящую только из нулей, и среди всех таких — имеющую наибольшую площадь (подматрица — это прямоугольная область матрицы).

Тривиальный алгоритм, — перебирающий искомую подматрицу, — даже при самой хорошей реализации будет работать $O(n^2 m^2)$. Ниже описывается алгоритм, работающий за $O(nm)$, т.е. за линейное относительно размеров матрицы время.

Алгоритм

Для устранения неоднозначностей сразу заметим, что n равно числу строк матрицы a, соответственно, m — это число столбцов. Элементы матрицы будем нумеровать в 0-индексации, т.е. в обозначении $a[i][j]$ индексы i и j пробегают диапазоны $i = 0 \ldots n-1$, $j = 0 \ldots m-1$.

Шаг 1: Вспомогательная динамика

Сначала посчитаем следующую вспомогательную динамику: $d[i][j]$ — ближайшая сверху единица для элемента $a[i][j]$. Формально говоря, $d[i][j]$ равно наибольшему номеру строки (среди строк диапазоне от -1 до i), в которой в j-ом столбце стоит единица. В частности, если такой строки нет, то $d[i][j]$ полагается равным -1 (это можно понимать как то, что вся матрица как будто ограничена снаружи единицами).

Эту динамику легко считать двигаясь по матрице сверху вниз: пусть мы стоит в i-ой строке, и известно значение динамики для предыдущей строки. Тогда достаточно скопировать эти значения в динамику для текущей строки, изменив только те элементы, в которых в матрице стоят единицы. Понятно, что тогда даже не требуется хранить всю прямоугольную матрицу динамики, а достаточно только одного массива размера m:

```
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;

	// вычислили d для i-ой строки, можем здесь использовать
эти значения
}
```

Шаг 2: Решение задачи

Уже сейчас мы можем решить задачу за $O(nm^2)$ — просто перебирать в текущей строке номер левого и правого столбцов искомой подматрицы, и с помощью динамики $d[][]$ вычислять за $O(1)$ верхнюю границу нулевой подматрицы. Однако можно пойти дальше и значительно улучшить асимптотику решения.

Ясно, что искомая нулевая подматрица ограничена со всех четырёх сторон какими-то единичками (либо границами поля), — которые и мешают ей увеличиться в размерах и улучшить ответ. Поэтому, утверждается, мы не пропустим ответ, если будем действовать следующим образом: сначала переберём номер i нижней строки нулевой подматрицы, затем переберём, в каком столбце j мы будем упирать вверх нулевую подматрицу. Пользуясь значением $d[i][j]$, мы сразу получаем номер верхней строки нулевой подматрицы. Осталось теперь определить оптимальные левую и правую границы нулевой подматрицы, — т.е. максимально раздвинуть эту подматрицу влево и вправо от j-го столбца.

Что значит раздвинуть максимально влево? Это значит найти такой индекс k_1, для которого будет $d[i][k_1] > d[i][j]$, и при этом k_1 — ближайший такой слева для индекса j. Понятно, что тогда $k_1 + 1$ даёт номер левого столбца искомой нулевой подматрицы. Если такого индекса вообще нет, то положить $k_1 = -1$ (это означает, что мы смогли расширить текущую нулевую подматрицу влево до упора — до границы всей матрицы a).

Симметрично можно определить индекс k_2 для правой границы: это ближайший справа от j индекс такой, что $d[i][k_2] > d[i][j]$ (либо m, если такого индекса нет).

Итак, индексы k_1 и k_2, если мы научимся эффективно их искать, дадут нам всю необходимую информацию о текущей нулевой подматрице. В частности, её площадь будет равна $(i - d[i][j]) \cdot (k_2 - k_1 - 1)$.

Как же искать эти индексы k_1 и k_2 эффективно при фиксированных i и j? Нас удовлетворит только асимптотика $O(1)$, хотя бы в среднем.

Добиться такой асимптотики можно с помощью стека (stack) следующим образом. Научимся сначала искать индекс k_1, и сохранять его значение для каждого индекса j внутри текущей строки i в динамике $d_1[i][j]$. Для этого будем просматривать все столбцы j слева направо, и заведём такой стек, в котором всегда будут лежать только те столбцы, в которых значение динамики $d[][]$ строго больше $d[i][j]$. Понятно, что при переходе от столбца j к следующему столбцу $j+1$ требуется обновить содержимое этого стека. Утверждается, что требуется сначала положить в стек столбец j (поскольку для него стек "хороший"), а затем, пока на вершине стека лежит неподходящий элемент (т.е. у которого значение $d \le d[i][j+1]$), — доставать этот элемент. Легко понять, что удалять из стека достаточно только из его вершины, и ни из каких других его мест (потому что стек будет содержать возрастающую по d последовательность столбцов).

Значение $d_1[i][j]$ для каждого j будет равно значению, лежащему в этот момент на вершине стека.

Ясно, что поскольку добавлений в стек на каждой строчке i происходит ровно m штук, то и удалений также не могло быть больше, поэтому в сумме асимптотика будет линейной.

Динамика $d_2[i][j]$ для нахождения индексов k_2 считается аналогично, только надо просматривать столбцы справа налево.

Также следует отметить, что этот алгоритм потребляет $O(m)$ памяти (не считая входные данные — матрицу $a[][]$).

Реализация

Эта реализация вышеописанного алгоритма считывает размеры матрицы, затем саму матрицу (как последовательность чисел, разделённых пробелами или переводами строк), и затем выводит ответ — размер наибольшей нулевой подматрицы.

Легко улучшить эту реализацию, чтобы она также выводила саму нулевую подматрицу: для этого надо при каждом изменении ans запоминать также номера строк и столбцов подматрицы (ими будут соответственно $d[j]+1$, i, $d1[j]+1$, $d2[j]-1$).

```
int n, m;
cin >> n >> m;
vector < vector<int> > a (n, vector<int> (m));
for (int i=0; i<n; ++i)
	for (int j=0; j<m; ++j)
		cin >> a[i][j];

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;
	while (!st.empty()) st.pop();
	for (int j=0; j<m; ++j) {
		while (!st.empty() && d[st.top()] <= d[j])
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()] <= d[j])
st.pop();
		d2[j] = st.empty() ? m : st.top();
		st.push (j);
	}
	for (int j=0; j<m; ++j)
		ans = max (ans, (i - d[j]) * (d2[j] - d1[j] - 1));
}

cout << ans;
```