Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет «Компьютерные науки и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №7 по курсу

«Дискретный анализ»

Тема работы

“Динамическое программирование”

Студент : Ю.И. Катаев
Группа : М8О-310Б-21
Преподаватель : Н.К. Макаров
Оценка : _____
Дата : _____
Подпись : _____

Москва, 2023

1. Постановка задачи

Задан прямоугольник с высотой n и шириной m, состоящий из нулей и единиц. Найдите в нем прямоугольник наибольшей площади, состоящий из одних нулей.

Формат ввода
В первой строке заданы 1 <= n <= 500 и 1 <= m <= 500. В последующих n строках записаны по m символов 0 или 1 - элементы прямоугольника.

Формат вывода
Необходимо вывести одно число – максимальную площадь прямоугольника из одних нулей.

2. Описание

Динамическое программирование - способ решения сложных задач путём разбиения их на более простые подзадачи. Он применим к задачам с оптимальной подструктурой, выглядящим как набор перекрывающихся подзадач, сложность которых чуть меньше исходной. В этом случае время вычислений, по сравнению с «наивными» методами, можно значительно сократить.
Ключевая идея в динамическом программировании достаточно проста. Как правило, чтобы решить поставленную задачу, требуется решить отдельные части задачи (подзадачи), после чего объединить решения подзадач в одно общее решение. Часто многие из этих подзадач одинаковы. Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений. Это особенно полезно в случаях, когда число повторяющихся подзадач экспоненциально велико.

Данная задача как раз решается с помощью динамического программирования. Но найти прямоугольник наибольшей площади в другом прямоугольнике нетривиальная задача. Пойти во все стороны, поддерживая

какие-то значения, не получится (так получилось бы, если бы мы искали квадрат максимальной площади). Идея решения заключается в сведении этой задачи к другой задаче - нахождение максимальной площади прямоугольника в гистограмме. Для этого нам нужно пробежаться по нашему двумерному массиву, в котором мы храним исходный прямоугольник из нулей и единиц, по столбцам сверху вниз, насчитывая двумерный массив высот. Для каждого столбца делаем так: если мы увидели 0, тогда записываем 1 в массив высот и сохраняем это значение. Если дальше снова 0, тогда мы уже ставим 2 в массив высот, то есть прибавляем 1 к прошлому шагу и так далее. А если встретилась единица,
тогда заполняем эту позицию 0 в массиве высот и обнуляем наш счётчик (память), то есть когда мы встретим в следующий раз 0, то напишем 1 в массив высот.

Следующий шаг - решение задачи нахождения максимальной площади прямоугольника в гистограмме n раз, потому что каждая строка двумерного массива высот это гистограмма. Эта задача решается таким образом. Используя стек, добавляем туда каждый новый прямоугольник. Если высота следующего прямоугольника меньше, чем высота предыдущего, тогда достаем прямоугольники из стека, попутно считая
их общую площадь, пока не попадётся прямоугольник, у которого высота меньше или равна высоте текущего прямоугольника. Также нам нужно поддерживать максимум среди всех найденных площадей за всё время. Это и будет ответом на задачу.
Ещё доставать все прямоугольники нужно, если числа в строке массива высот закончились.
Сложность алгоритма составляет O(n * m) по времени и памяти, так как мы n раз решаем линейную задачу.

3. Исходный код

```
#include <bits/stdc++.h>

using namespace std;

void CleanStack(stack<int> &stack) {
```

```
    while (!stack.empty()) {
        stack.pop();
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr), cout.tie(nullptr);

    int n, m;
    cin >> n >> m;
    vector<vector<int>> matrix(n, vector<int> (m));
    for (int i = 0; i < n; ++i) {
        string row;
        cin >> row;
        for (int j = 0; j < m; ++j) {
            matrix[i][j] = row[j] - '0';
        }
    }
    // в j ячейке хранится номер ближайшей сверху строки, в которой
matrix[i][j]=1
    vector<int> ones_above(m, -1);
    // в j ячейке хранится ближайший слева столбец, который имеет
меньше нулей сверху, чем у j
    vector<int> nearest_left(m);
    // в j ячейке хранится ближайший справа столбец, который имеет
меньше нулей сверху, чем у j
    vector<int> nearest_right(m);

    stack<int> nearest_smaller_height;
    long long result = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (matrix[i][j] == 1) {
                ones_above[j] = i;
            }
        }
        CleanStack(nearest_smaller_height);
```

```
        for(int j = 0; j < m; ++j) {
            while (nearest_smaller_height.size() != 0 &&
ones_above[nearest_smaller_height.top()] <= ones_above[j]) {
                nearest_smaller_height.pop();
            }
            if (nearest_smaller_height.size() > 0) {
                nearest_left[j] = nearest_smaller_height.top();
            } else {
                nearest_left[j] = -1;
            }
            nearest_smaller_height.push(j);
        }

        CleanStack(nearest_smaller_height);
        for (int j = m-1; j >= 0; --j) {
            while (nearest_smaller_height.size() != 0 &&
ones_above[nearest_smaller_height.top()] <= ones_above[j]) {
                nearest_smaller_height.pop();
            }
            if (nearest_smaller_height.size() > 0) {
                nearest_right[j] = nearest_smaller_height.top();
            } else {
                nearest_right[j] = m;
            }
            nearest_smaller_height.push(j);
        }

        for (int j = 0; j < m; ++j) {
            long long current_square = (i - ones_above[j]) *
(nearest_right[j] - nearest_left[j] - 1);
            if (current_square > result) {
                result = current_square;
            }
        }

    }

    cout << result << '\n';
```

```
    return 0;
}
```

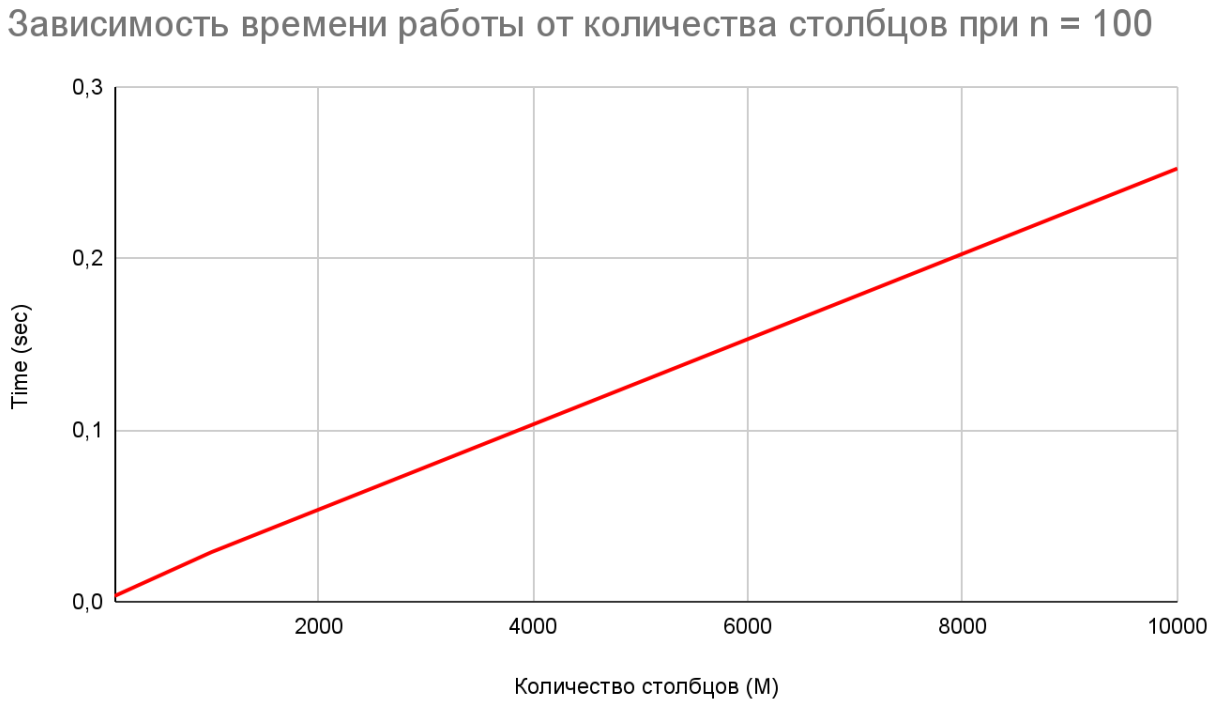
4. Демонстрация работы программы

```
➜  src git:(main) ✗ cat test.txt
4 5
01011
10001
01000
11011
➜  src git:(main) ✗ g++ main.cpp
➜  src git:(main) ✗ ./a.out < test.txt
4
```

5. Тесты производительности

Убедимся, что построенный алгоритм действительно имеет сложность O(n*m). Сначала будем увеличивать количество столбцов (m)

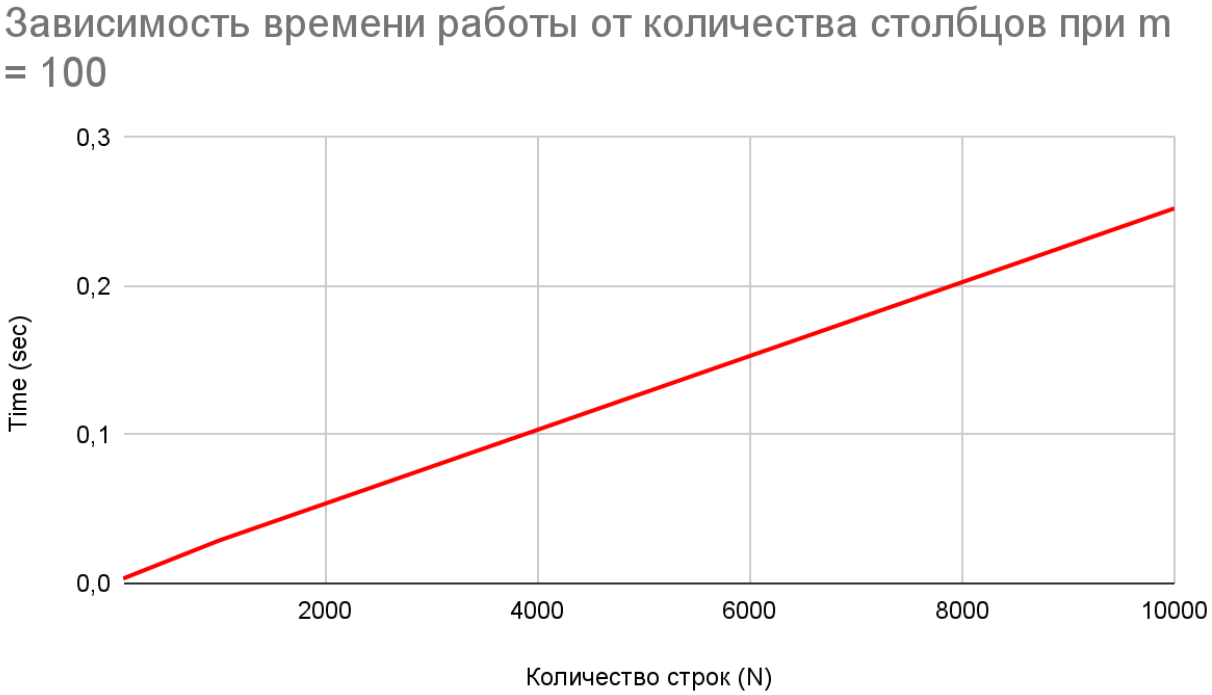
Количество строк (n)	Количество столбцов (m)	Время выполнения (sec)
100	100	0.0032529
100	1000	0.0287571
100	10000	0.252338



Теперь меняем количество строк

Количество строк (n)	Количество столбцов (m)	Время выполнения (sec)
100	100	0.0032529
1000	100	0.0245197
10000	100	0.250727

Увеличиваем и количество строк и количество столбцов



6. Выводы

Для выполнение данной лабораторной работ я применял методы динамического программирования. Этот способ решения помогает решать сложные задачи, которые иначе было бы трудоемко или даже невозможно решить другими методами. Динамическое программирование позволяет сохранять и использовать результаты промежуточных вычислений, что позволяет существенно сократить время выполнения алгоритмов.

Таким образом, изучение и применение динамического программирования является важным этапом в развитии навыков программирования и позволяет решать сложные задачи более эффективно и оптимально.