

**Московский авиационный институт  
(национальный исследовательский университет)**

**Факультет информационных технологий и прикладной  
математики**

**Кафедра вычислительной математики и программирования**

**Лабораторная работа №7 по курсу «Дискретный анализ»**

Студент: Е. А. Суханов  
Преподаватель: А. А. Кухтичев  
Группа: М8О-306Б  
Дата:  
Оценка:  
Подпись:

**Москва, 2021**

## Лабораторная работа №7

**Задача:** При помощи метода динамического программирования разработать алгоритм решения задачи, определяемой своим вариантом; оценить время выполнения алгоритма и объем затрачиваемой оперативной памяти. Перед выполнением задания необходимо обосновать применимость метода динамического программирования.

Разработать программу на языке C или C++, реализующую построенный алгоритм. Формат входных и выходных данных описан в варианте задания:

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

**Формат входных данных** В первой строке заданы  $1 \leq n \leq 500$  и  $1 \leq m \leq 500$ . В следующих  $n$  строках записаны по  $m$  символов 0 или 1 — элементы прямоугольника.

**Формат результата** Необходимо вывести одно число — максимальную площадь прямоугольника из одних нулей.

# 1 Описание

Давайте вычислять для каждой ячейки площадь одного из прямоугольников, в которых находится эта ячейка. Решая данную задачу перебором, мы можем найти максимальную площадь, но сложность такого алгоритма будет равна  $O((n * m)^3)$ , что очень много.

Попытаемся уменьшить сложность, используя динамическое программирование: будем обрабатывать матрицу построчно. Пусть каждый элемент массива будет дополнительно хранить информацию об прямоугольнике, в который входит этот элемент(если он равен нулю). А именно: высоту, начало и конец прямоугольника.

Обрабатывать строку мы будем следующим образом:

- Если текущая ячейка равна нулю, и является продолжением прямоугольника с прошлой строки, то:
  - Увеличить высоту на 1;
  - Сохранить или сузить границы прямоугольника. Последний вариант применяется, если последовательность нулей уже прямоугольника с прошлой строки;
- Если текущая ячейка равна нулю, и не является продолжением прямоугольника с прошлой строки, то:
  - Увеличить высоту на 1;
  - Задать границы прямоугольника равными границам последовательности нулей;
- Если текущая ячейка равна единице, то:
  - Задать нулевую высоту;
  - Указать ширину всей строки;

Определять границы последовательности нулей будем в два захода:

- Определяем новое значение левой границы, когда встречаем единицу:  $Start = j + 1$ , где  $j$  индекс единицы;
- При втором обходе строки из конца в начало: определяем значение правой границы, когда встречаем единицу:  $End = j$ ;

Таким образом граница прямоугольника определяется полуинтервалом  $[Start, End)$ . Перед началом обработки строки, нужно скопировать информацию об ячейках с прошлой. Именно поэтому мы обнуляем значения, когда встречаем единицу. Таким образом мы переиспользуем уже вычисленные значения.

При этом гарантируется, что ячейки будут хранить площади все-возможных прямоугольников „максимальной“ формы:

- Если ячейка не является продолжением прямоугольника с прошлой строки, то границы прямоугольника будут равны длине последовательности;
- Если ячейка является продолжением прямоугольника с прошлой строки, то либо будет захвачена вся последовательность, либо будет хотя бы одна ячейка, которая заполнится согласно верхнему пункту;
- Если несколько ячеек являются продолжениями разных прямоугольников, то будет хотя бы одна, которая находится между ними и ее границы будут равны длине последовательности;

Таким образом хотя бы одна ячейка хранит информацию (площадь  $Height * (End - Start)$ ) каждого прямоугольника „максимальной“ формы.

Данный алгоритм работает за  $O(n * m)$ , так как нам нужно обработать каждый элемент массива, а так же требует  $O(n * m)$  по памяти для хранения дополнительной матрицы. Но, на самом деле, можно использовать только  $O(n)$  памяти, так как мы работаем только с последней строкой.

## 2 Исходный код

Заголовочный файл solution.hpp:

```
1 | #include "vector"
2 | #include "iostream"
3 |
4 | typedef std::vector<std::vector<char>> TMatrix;
5 |
6 | class TSolution {
7 | public:
8 |     TSolution();
9 |     static int FindLargestArea(const TMatrix\& matrix);
10 | private:
11 |     struct TRectInfo {
12 |         int Height;
13 |         int Start;
14 |         int End;
15 |     };
16 | };
17 |
18 | void ReadMatrix(TMatrix\& matrix);
```

Реализация solution:

```
1  #include "solution.hpp"
2
3  int TSolution::FindLargestArea(const TMatrix& matrix) {
4      int n = matrix.size();
5      int m = matrix[0].size();
6
7      std::vector<std::vector<TRectInfo>> dp(n,
8          std::vector<TRectInfo>(m, {0,0,m}));
9
10     int maxArray = 0;
11     for(int i = 0; i < n; ++i) {
12         int Start = 0;
13         for(int j = 0; j < m; ++j) {
14             if(i > 0)
15                 dp[i][j] = dp[i-1][j];
16
17             if(matrix[i][j] == '0') {
18                 dp[i][j].Height++;
19                 dp[i][j].Start = std::max(Start, dp[i][j].Start);
20             }
21             else {
22                 dp[i][j].Height = 0;
23                 dp[i][j].Start = 0;
24                 Start = j + 1;
25             }
26         }
27
28         int End = m;
29         for(int j = m-1; j >= 0; --j) {
30             if(matrix[i][j] == '0') {
31                 dp[i][j].End = std::min(End, dp[i][j].End);
32             }
33             else {
34                 dp[i][j].End = m;
35                 End = j;
36             }
37
38             int array = dp[i][j].Height * (dp[i][j].End - dp[i][j].Start);
39             if(array > maxArray)
40                 maxArray = array;
41         }
42     }
43
44     return maxArray;
45 }
46
47 void ReadMatrix(TMatrix& matrix) {
48     int n,m;
```

```

49 |     std::cin >> n >> m;
50 |     matrix.resize(n);
51 |
52 |     for(int i = 0; i < n; ++i) {
53 |         matrix[i].resize(m);
54 |         for(int j = 0; j < m; ++j) {
55 |             std::cin >> matrix[i][j];
56 |         }
57 |     }
58 | }

```

Файл main.cpp:

```

1 | #include <iostream>
2 | #include <vector>
3 |
4 | #include "solution.hpp"
5 |
6 | int main() {
7 |     std::ios_base::sync_with_stdio(false);
8 |     TMatrix matrix;
9 |     ReadMatrix(matrix);
10 |    std::cout << TSolution::FindLargestArea(matrix) << std::endl;
11 |    return 0;
12 | }

```

### 3 Консоль

```
$ make
g++ -c -Wall -pedantic -std=c++14 -O2 main.cpp -o main.o
g++ -c -Wall -pedantic -std=c++14 -O2 solution.cpp -o solution.o
g++ -O2 main.o solution.o -o solution
$ ./solution
4 5
01011
10001
01000
11011
4
```



## 4 Тест производительности

Мое решение я буду сравнивать с наивным. На случайной матрице, размером 10x10 и 100x100:

Матрица 10x10:

```
$ ./benchmark <../generated_tests/randomtest10.txt
My Solution : 340us
Default Solution : 290us
```

При маленьких матрицах наивное решение отрабатывает быстрее. Можно сделать вывод, что у решения ДП большая константа.

Матрица 100x100:

```
$ ./benchmark <../generated_tests/randomtest100.txt
My Solution : 444us
Default Solution : 27994us
```

Видно, что сложность наивного решения растет намного быстрее, чем сложность у решения с помощью ДП.

## 5 Выводы

Суть динамического программирования заключается в решении большой задачи, с помощью использования решения такой же задачи, но меньшего размера. Этот метод позволяет создать алгоритм с меньшей сложностью, чем сложность у алгоритма перебором. Однако он не всегда применим. Кроме этого, бывает сложно понять, как именно можно разбить задачу на более мелкие, и как на их основе решить большую.

## Список литературы

[1] *Динамическое программирование ИТМО*

URL: [https://neerc.ifmo.ru/wiki/index.php?title=Динамическое\\_программирование](https://neerc.ifmo.ru/wiki/index.php?title=Динамическое_программирование)  
(дата обращения: 24.08.2021)