

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Построение и анализ алгоритмов»
Тема: Поиск с возвратом

Студент гр. 8304

Кириянов Д.И.

Преподаватель

Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

Построение и анализ алгоритма бэктрекинга (поиск с возвратом) на основе решения задачи квадратирования прямоугольников.

Вариант 4р.

Основные теоретические положения.

Размер столешницы – $N \times M$, где $(2 \leq N \leq 20)$ и $(2 \leq M \leq 20)$. Необходимо найти и вывести: одно число K , задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(прямоугольник) заданного размера $N \times M$, рёбра квадратов меньше рёбер прямоугольника. Далее должны идти K строк, каждая из которых должна содержать три целых числа x , y и w , задающие координаты левого верхнего угла и длину стороны соответствующего обрезка(квадрата).

Описание алгоритма.

1. Вводятся стороны N и M прямоугольника. Происходит поиск минимальной стороны. Создается двумерное представление данного прямоугольника при помощи векторов. Все клетки инициализируются нулями. Создаются два вектора структур для хранения координат. Один для минимального количества квадратов, другой для текущего. Запускается отсчет времени и вызывается рекурсивная функция.
2. Создается структура для хранения координат нового квадрата. Происходит поиск первой свободной клетки в матрице. Если пустая клетка не найдена, это означает, что матрица заполнена и данный случай запоминается как минимальный, а алгоритм возвращается к предыдущему этапу рекурсии. Если клетка найдена, то ее координаты заносятся в структуру и алгоритм продолжает работу.

3. Учитывается, что сторона квадрата должна быть меньше сторон прямоугольника, затем происходит проверка. Если квадрат выходит за пределы прямоугольника, то алгоритм возвращается к предыдущему этапу рекурсии.

4. Происходит проверка размера векторов на то, не превосходит ли размер текущего вектора размер минимального. Если текущий вектор больше, то алгоритм возвращается к предыдущему этапу рекурсии.

5. Проверяется возможность вставки квадрата из найденной клетки. Если такой возможности нет, т.е. не все требующиеся клетки свободны, то алгоритм возвращается к предыдущему этапу рекурсии.

6. После возврата из рекурсии все закрашенные клетки снова инициализируем нулями для проверки других вариантов.

7. Алгоритм просматривает все возможные варианты, записывает результат в отведенный вектор и завершает свою работу.

8. Подсчитывается время работы алгоритма, количество найденных квадратов, количество вариантов покрытия минимальным числом квадратов. Данная информация выводится, после чего выводится список всех длин и координат полученных квадратов. Программа завершает работу.

Оптимизации.

За счет различных проверок, о которых рассказывается в описании алгоритма, программа выполняется более оптимально.

Описание основных структур данных и функций.

```
struct coord {  
    int x;  
    int y;  
    int len;
```

```
};
```

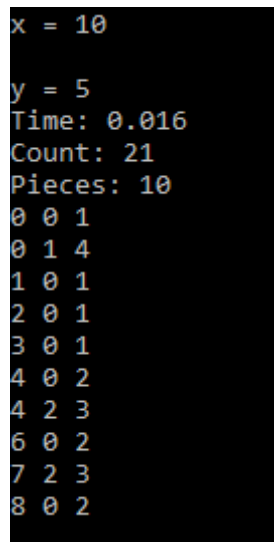
- структура, содержащая координаты и длину стороны квадрата.

```
void solve(std::vector<std::vector<int>>& rectangle, std::vector<coord>& potential,  
std::vector<coord>& answer, int min, int x, int y, int& count)
```

- рекурсивная функция, принимающая двумерное представление прямоугольника, вектор для текущих значений, вектор для минимальных значений, минимальную сторону прямоугольника, значения сторон прямоугольника, переменную для подсчета вариантов покрытия минимальным числом квадратов. Выполняет описанный алгоритм.

Тестирование.

1) Обычный случай.



```
x = 10  
y = 5  
Time: 0.016  
Count: 21  
Pieces: 10  
0 0 1  
0 1 4  
1 0 1  
2 0 1  
3 0 1  
4 0 2  
4 2 3  
6 0 2  
7 2 3  
8 0 2
```

Рисунок 1 – Результат при обычном случае

2) Случай, когда $M=N$.

```
x = 9
y = 9
Time: 0.072
Count: 34
Pieces: 6
0 0 3
0 3 3
0 6 3
3 0 3
3 3 6
6 0 3
```

Рисунок 2 – Результат при $M=N$

3) Неверный ввод данных.

```
x = 2
y = 1
Wrong Input!
```

Рисунок 3 – Результат при неверном вводе

Частичные решения.

Был реализован вывод частичных решений. Каждый раз, когда минимальное возможное количество разбиений меняется, программа выводит данный вариант на экран. Последним выводится конечный результат.

```

1 2 3 3 4 4 5 5 5 5
6 7 3 3 4 4 5 5 5 5
8 8 8 8 8 8 5 5 5 5
8 8 8 8 8 8 5 5 5 5
8 8 8 8 8 8 9 9 9 9
8 8 8 8 8 8 9 9 9 9
8 8 8 8 8 8 9 9 9 9
8 8 8 8 8 8 9 9 9 9
8 8 8 8 8 8 9 9 9 9

1 2 2 2 3 3 3 4 4 4
5 2 2 2 3 3 3 4 4 4
6 2 2 2 3 3 3 4 4 4
7 7 7 7 7 8 8 8 8 8
7 7 7 7 7 8 8 8 8 8
7 7 7 7 7 8 8 8 8 8
7 7 7 7 7 8 8 8 8 8
7 7 7 7 7 8 8 8 8 8
7 7 7 7 7 8 8 8 8 8

1 1 2 2 3 3 4 4 4 4
1 1 2 2 3 3 4 4 4 4
5 5 5 5 5 5 4 4 4 4
5 5 5 5 5 5 4 4 4 4
5 5 5 5 5 5 6 6 6 6
5 5 5 5 5 5 6 6 6 6
5 5 5 5 5 5 6 6 6 6
5 5 5 5 5 5 6 6 6 6
5 5 5 5 5 5 6 6 6 6

Time: 0.422
Count: 34
Pieces: 6
0 0 2
0 2 2
0 4 2
0 6 4
2 0 6
4 6 4

```

Рисунок 4 – Пример частичных результатов

Оценка сложности алгоритма.

1. Затраты по памяти: Если размер входных данных n , то программа хранит n^2 значений под исходных квадрат, и два вектора для хранения минимального и частого разбиения, которые в самом худшем случае имеют оценку n^2 , но в случае с отсутствием входного списка необходимых квадратов, оба списка линейно зависят от n . Исходя из этого, общая оценка затрат по памяти составляет $O(n^2)$.

2. Затраты по времени: использованный алгоритм поиска с возвратом является улучшенным обходом графа. Алгоритм работает за экспоненциальное время, несмотря на то, что значительная часть поддеревьев отбрасывается благодаря начальным оптимизациям. Общая оценка затрат по времени составляет $O(2^n)$.

Вывод.

В ходе работы был построен и проанализирован алгоритм бэктрекинга на основе решения задачи квадратирования прямоугольников. Исходный код программы представлен в приложении А.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>

struct coord { //структура для хранения координат
    int x;
    int y;
    int len;
};

void solve(std::vector<std::vector<int>>& rectangle, std::vector<coord>& potential,
std::vector<coord>& answer, int min, int x, int y, int& count, int paint){
    coord point;
    bool check = false;
    for (int i = 0; i < x; ++i){
        for (int j = 0; j < y; ++j)
            if (rectangle[i][j] == 0){ //поиск первой свободной
                point.x = i; //клетки в матрице
                point.y = j;
                check = true;
                break;
            }
        if (check)
            break;
    }
    if (check == false){ //если свободная клетка не найдена
        count += 1;
        answer = potential; //то случай запоминается как минимальный
        for (int i = 0; i < x; ++i) {
            std::cout << std::endl;
            for (int j = 0; j < y; ++j)
                std::cout << rectangle[i][j]<<" "; //и выводится частичное
        }
        std::cout << std::endl;
        return;
    }
    for (int size = 1; size < min; ++size){ //проверка возможности вставки квадрата со
        стороной size
        if (point.x + size > x || point.y + size > y)
            return; //если выходит за границы
        if (potential.size() + 1 >= answer.size())
            return; //если количество разбиений превосходит уже имеющееся
        for (int i = point.x; i < point.x + size; ++i)
            for (int j = point.y; j < point.y + size; ++j)
                if (rectangle[i][j] != 0) //если пересекается
                    return; //с уже закрашенными квадратами
        for (int i = point.x; i < point.x + size; ++i)
            for (int j = point.y; j < point.y + size; ++j) //если прошло все
                rectangle[i][j] = paint; //то закрашиваем
        point.len = size;
        potential.push_back(point);
        solve(rectangle, potential, answer, min, x, y, count, paint+1); //продолжаем
    }
    for (int i = point.x; i < point.x + size; ++i)
        for (int j = point.y; j < point.y + size; ++j)
            rectangle[i][j] = 0; //возвращаем 0 для проверки других
    }
}
```



```

        potential.pop_back();
    }
}

int main(){
    int x = 0;
    std::cout << "x = ";
    std::cin >> x;
    if (x < 2) {
        std::cout << "Wrong Input!" << std::endl;
        return 0;
    }
    std::cout << "\ny = "; // считывание сторон прямоугольника
    int y = 0;
    std::cin >> y;
    if (y < 2) { //длина стороны должна быть не менее 2
        std::cout << "Wrong Input!" << std::endl;
        return 0;
    }
    int min = 0;
    int max = 0;
    if (y > x) {
        min = x;
    } //поиск минимальной стороны
    else {
        min = y;
    }
    std::vector<std::vector<int>> rectangle(x);
    for (int i = 0; i < x; ++i){
        std::vector<int> side(y);
        for (int j = 0; j < y; ++j) //создание двумерного вектора
            side[j] = 0;
        rectangle[i] = side;
    }
    std::vector<coord> potential; //создание векторов для хранения
    std::vector<coord> answer(x * y + 1); //конечного ответа и текущего ответа
    int count = 0;
    clock_t time = clock(); //считывание времени
    solve(rectangle, potential, answer, min, x, y, count, 1); //запуск рекурсии
    time = clock() - time; //подсчет сколько времени заняла рекурсия
    std::cout << "Time: ";
    std::cout << (double)(time) / CLOCKS_PER_SEC << std::endl;
    std::cout << "Count: " << count << std::endl;
    std::cout << "Pieces: " << answer.size() << std::endl; //вывод всех результатов
    for (size_t i = 0; i < answer.size(); ++i)
        std::cout << answer[i].x
            << " " <<
            answer[i].y
            << " " <<
            answer[i].len
            << std::endl;
    return 0;
}

```