

ВМИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

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

Студент гр. 9382

Герасев Г.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Получить представление о решении NP — полных задач, изучить такой метод решения, как поиск с возвратом, проследить зависимость количества операций для решения поставленной задачи от входных данных.

Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 1 до $N-1$, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу - квадрат размера N . Он может получить ее, собрав из уже имеющихся обрезков(квадратов).

Например, столешница размера 7×7 может быть построена из 9 обрезков.



Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные

Размер столешницы - одно целое число N ($2 \leq N \leq 20$).

Выходные данные

Одно число K , задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N . Далее должны идти K строк, каждая из которых должна содержать три целых числа x , y и w , задающие координаты левого верхнего угла ($1 \leq x, y \leq N$) и длину стороны соответствующего обрезка(квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Вар. 1и. Итеративный бэктрекинг. Выполнение на Stepik двух заданий в разделе 2.

Описание алгоритма.

Создается стек в котором в формате тройке находятся все поставленные квадраты – $(x, y, size)$, а также карта, на которой отмечаются, какие из полей уже были заполнены квадратами раньше.

Понятно, что замощение для квадрата размера n и наименьшего простого делителя p будут одинаковы – ведь разбив квадрат размером делителя p получившиеся квадраты можно просто домножить на k , получив квадраты, разбивающие уже n .

Тогда будем предполагать, что квадрат имеет стороны простой длины. Тогда в его замощении обязательно будут квадраты размера $n/2$, $n/2 - 1$, $n/2 - 1$ в левом верхнем, правом верхнем и левом нижнем углах соответственно. (без доказательства). Тогда работать алгоритм будет только с оставшимся свободным пространством.

Алгоритм работает следующим образом. Находится самая левая верхняя свободная клетка. В этот угол вставляется наибольший квадрат. Далее находится следующая клетка. Если клеток не осталось, то сначала сравнивается нынешнее замощение с текущим наилучшим, и заменяется, если найденное – лучше. Далее последний вставленный квадрат заменяется на меньший на 1 и алгоритм продолжает работу. Если последний квадрат размера 1, то он удаляется и последним считается квадрат до него.

Понятно, что алгоритму незачем «двигать квадраты» – достаточно просто уменьшать их размер и вставлять новые, ведь если подвинуть квадрат образуется полость, которая в дальнейшем будет заполнена – значит алгоритм просто расставил квадраты точно также, вплотную друг к другу, но в другом порядке. Значит «двигать квадраты» для перебора всех возможных замощений не требуется.

Описание функций и структур данных.

Структуры точек и квадратов:

- Point (int x, y)
- Square (unsigned x, y, size)

Класс карты расположения квадратов, с соответствующими методами:

```
class SquarePavingMap
```

```
    unsigned int size = 0; – размер карты
```

```
    std::array< std::array<bool, MAX_MAP_SIZE>, MAX_MAP_SIZE>
```

```
map; – карта
```

Методы:

```
bool isInBound(int coord); – На карте ли координата
```

```
void add(Square square); – добавление квадрата на карту
```

```
void remove(Square square) – удаление квадрата с карты;
```

```
unsigned int giveMaxSquareSize(Point point); – нахождение максимального  
квадрата, который можно поставить по переданной координате на карте.
```

```
Point giveFirstFree(); – первая свободная на карте точка, {-1, -1} при  
неудаче
```

```
void view(); – печать в терминал состояние карты
```

Класс карты со стеками для решения задачи:

```
class SquarePaver
```

```
std::vector<Square> bestPaving; – лучшее найденное разбиение
```

```
std::vector<Square> currentPaving; – нынешнее разбиение
```

```
unsigned int squarePavingMapSize; – размер карты
```

```
SquarePavingMap squarePavingMap; - карта
```

Методы:

void addSquare(Square square) – добавление квадрата (стек и карта)

void removeLastSquare() – удаление последнего квадрата (стек и карта)

void removeLastSmallSquares() – удаление всех квадратов с конца = 1

void updatePaving() – установка нового наилучшего разбиения

void viewBestPaving() – печать лучшего разбиения

void viewCurrentPaving() – печать нынешнего разбиения

bool possibleToFindBetterPaving() – стоит ли останавливать поиск разбиения (ввиду слишком большого числа квадратов)

SquarePaver(unsigned int size) – инициализация, добавляется 3 квадрата на карту на этом этапе

void view() – печать в терминал всех полей
объекта

void findBestPaving() – поиск бектрекингом наилучшего разбиения

void viewFancyWay(unsigned int modifier) – вывод в виде приемлимый
стефиком

Оценка сложности по времени.

Поскольку используется довольно большое количество оптимизаций, то дать точную оценку сложности алгоритма – трудоемкая задача. Было принято решение дать алгоритму верхнюю границу того, сколько квадратов он переберет.

N – размер квадрата. Есть N^2 свободных клеток и N размеров квадрата, которые будут перебираться.

Ставим первый квадрат, его можно поставить $N^2 * N$ способами.

Ставим второй квадрат, его можно поставить $(N^2-1) * N$ способами.

И так дойдем до последней клетки. Получаем $O((N^2)! * N^N)$.

Оценка сложности алгоритма по памяти.

Всего используется два матрица, которые содержат текущее решение и оптимальное решение, поэтому сложность по памяти – $O(2 * N^2)$, где N – размер квадрата.

Тестирование.

Тестирование проведено с помощью системы Stepik. Также результаты представлены в таблице ниже.

Таблица 1. Результаты работы программы

Входные данные	Выходные данные без промежуточного вывода
3	6 1 1 2 1 3 1 3 1 1 2 3 1 3 2 1 3 3 1
5	8 1 1 3 1 4 2 4 1 2 3 4 2 4 3 1 5 3 1 5 4 1 5 5 1
7	9 1 1 4 1 5 3 5 1 3 4 5 2

	4 7 1 5 4 1 5 7 1 6 4 2 6 6 2
9	6 1 1 6 1 7 3 7 1 3 4 7 3 7 4 3 7 7 3
11	11 1 1 6 1 7 5 7 1 5 6 7 3 6 10 2 7 6 1 8 6 1 8 10 1 8 11 1 9 6 3 9 9 3
12	4 1 1 6 1 7 6 7 1 6 7 7 6
37	15 1 1 19 20 1 18 20 19 2 22 19 5

	27 19 11
	1 20 18
	19 20 1
	19 21 3
	19 24 8
	27 30 3
	30 30 8
	19 32 6
	25 32 1
	26 32 1
	25 33 5

Выводы.

Был реализован поиск с возвратом для поиска минимального количества непересекающихся квадратов, заполняющих исходный квадрат. Была написана рекурсивная функция, которая выполняет поставленную задачу.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <vector>
#include <array>

#define MAX_MAP_SIZE 40

struct Square
{
    unsigned int x, y, size;
};

struct Point
{

```

```

        int x, y;
    };

class SquarePavingMap
{
private:
    unsigned int size = 0;
    std::array< std::array<bool, MAX_MAP_SIZE>, MAX_MAP_SIZE> map;

    void viewAll();

public:
    SquarePavingMap(const unsigned int inputSize = 0);
    bool isInBound(int coord);
    void add(Square square);
    void remove(Square square);
    unsigned int giveMaxSquareSize(Point point);
    Point giveFirstFree();
    void view();
};

void printStack(const std::vector<Square>& st)
{
    for(auto &square : st)
    {
        std::cout << "{" << square.x << ", " << square.y << ", " <<
square.size << "} ";
    }
    std::cout << '\n';
}

unsigned int smallestDerivative(unsigned int n)
{
    unsigned int i = 3;
    if (n % 2 == 0) return 2;
    else {
        while(n%i && i*i <= n) {
            i += 2;
        }
        return ((n%i) ? n : i);
    }
}

class SquarePaver

```

```

{
private:
    std::vector<Square> bestPaving;
    std::vector<Square> currentPaving;

    unsigned int squarePavingMapSize;
    SquarePavingMap squarePavingMap;

    void addSquare(Square square)
    {
        currentPaving.push_back(square);
        squarePavingMap.add(square);
    }

    void removeLastSquare()
    {
        Square lastSquare = currentPaving.back();
        currentPaving.pop_back();
        squarePavingMap.remove(lastSquare);
    }

    void removeLastSmallSquares()
    {
        Square lastSquare = currentPaving.back();
        while(lastSquare.size == 1 && currentPaving.size() > 1)
        {
            removeLastSquare();
            lastSquare = currentPaving.back();
        }
    }

    void updatePaving()
    {
        bestPaving = currentPaving;
    }

    void viewBestPaving()
    {
        std::cout << "\nBestPavingMap -- \n";
        printStack(bestPaving);
    }

    void viewCurrentPaving()
    {

```

```

        std::cout << "\nCurrentPavingMap -- \n";
        printStack(currentPaving);
    }

    bool possibleToFindBetterPaving()
    {
        return ((bestPaving.size() == 0) || currentPaving.size() <
bestPaving.size());
    }

public:
    SquarePaver(unsigned int size)
    {
        squarePavingMapSize= size;
        squarePavingMap = SquarePavingMap(size);

        Square square = {0, 0, (size + 1) / 2};
        addSquare(square);

        square = {0, (size + 1) / 2, size - (size + 1) / 2};
        addSquare(square);

        square = {(size + 1) / 2, 0, size - (size + 1) / 2};
        addSquare(square);
    }

    void view()
    {
        viewBestPaving();
        viewCurrentPaving();
        std::cout << "\nsquarePavingMap -- \n";
        squarePavingMap.view();
        std::cout << '\n';
    }

    void findBestPaving()
    {
        while (currentPaving.size() >= 3) // Because if we have changed
initialisited squares we are out of options.
        {
            Point firstFree = squarePavingMap.giveFirstFree();
            Square lastSquare;
            while ((firstFree.x != -1 && possibleToFindBetterPaving()))
            {

```

```

        // std::cout << "firstFree -- (" << firstFree.x << ", " <<
firstFree.y << ")\n";
        // view();

        unsigned int currentMaxSize =
squarePavingMap.giveMaxSquareSize(firstFree);
        lastSquare = Square{(unsigned int) firstFree.x, (unsigned
int) firstFree.y, currentMaxSize};
        addSquare(lastSquare);
        firstFree = squarePavingMap.giveFirstFree();
    }
    // view();

        if (bestPaving.size() == 0 || bestPaving.size() >
currentPaving.size())
    {
        updatePaving();
    }

        // What to do when the map is filled? Delete all squares == 1
and one square after that
        removeLastSmallSquares();

        if (currentPaving.size() <= 3)
            break;

        lastSquare = currentPaving.back();
        removeLastSquare();
        lastSquare.size--;
        addSquare(lastSquare);
    }
}

void viewFancyWay(unsigned int modifier)
{
    std::cout << bestPaving.size() << '\n';
    for(auto &square : bestPaving)
    {
        std::cout << (square.x*modifier)+1 << " " <<
(square.y*modifier)+1 << " " << (square.size*modifier) << '\n';
    }
}

};

```

```

int main()
{
    unsigned int size;
    std::cin >> size;
    unsigned int n = smallestDerivative(size);
    unsigned int modifier = size/n;

    SquarePaver squarePaver(n);
    squarePaver.findBestPaving();
    squarePaver.viewFancyWay(modifier);

    return 0;
}

SquarePavingMap::SquarePavingMap(const unsigned int inputSize)
{
    if (inputSize <= MAX_MAP_SIZE)
        size = inputSize;

    for(int i=0; i<MAX_MAP_SIZE; i++)
    {
        for(int j=0; j<MAX_MAP_SIZE; j++)
            map[i][j] = false;
    }
}

bool SquarePavingMap::isInBound(int coord)
{
    return (0 <= coord && coord < size);
}

void SquarePavingMap::add(Square square)
{
    for (int i = square.x; (i<square.x + square.size) && isInBound(i); i+
+)
    {
        for (int j = square.y; (j<square.y + square.size) && isInBound(j);
j++)
            map[i][j] = true;
    }
}

void SquarePavingMap::remove(Square square)
{

```

```

        for (int i = square.x; (i<square.x + square.size) && isInBound(i); i+
+)
        {
            for (int j = square.y; (j<square.y + square.size) && isInBound(j);
j++)
                map[i][j] = false;
        }
    }

void SquarePavingMap::view()
{
    for (int i=0; i<size; i++)
    {
        for (int j=0; j<size; j++)
            std::cout << map[i][j] << ' ';
        std::cout << "\n";
    }
}

void SquarePavingMap::viewAll()
{
    for (int i=0; i<MAX_MAP_SIZE; i++)
    {
        for (int j=0; j<MAX_MAP_SIZE; j++)
            std::cout << map[i][j] << ' ';
        std::cout << "\n";
    }
}

Point SquarePavingMap::giveFirstFree()
{
    for (int i=0; i<size; i++)
    {
        for (int j=0; j<size; j++)
            if (map[i][j] == false) return Point{i, j};
    }
    return Point{-1, -1};
}

unsigned int SquarePavingMap::giveMaxSquareSize(Point point)
{
    int currentSize = 0, x = point.x, y = point.y;
    bool flag = true;

```

```

do
{
    currentSize++;
    for (int i=0; i<currentSize; i++)
    {
        if (isInBound(x) && isInBound(y-i))
        {
            if (map[x][y-i])
            {
                flag = false;
                break;
            }
        }
        else
        {
            flag = false;
            break;
        }

        if (isInBound(x-i) && isInBound(y))
        {
            if (map[x-i][y])
            {
                flag = false;
                break;
            }
        }
        else
        {
            flag = false;
            break;
        }
    }
    x++;
    y++;
} while (flag);
currentSize--;
return (currentSize);
}

```