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

ОТЧЕТ

по практической работе №1

по дисциплине «Построение и анализ алгоритмов»

Тема: Поиск с возвратом

Студентка гр. 8382 _____ Наконечная А. Ю.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2020

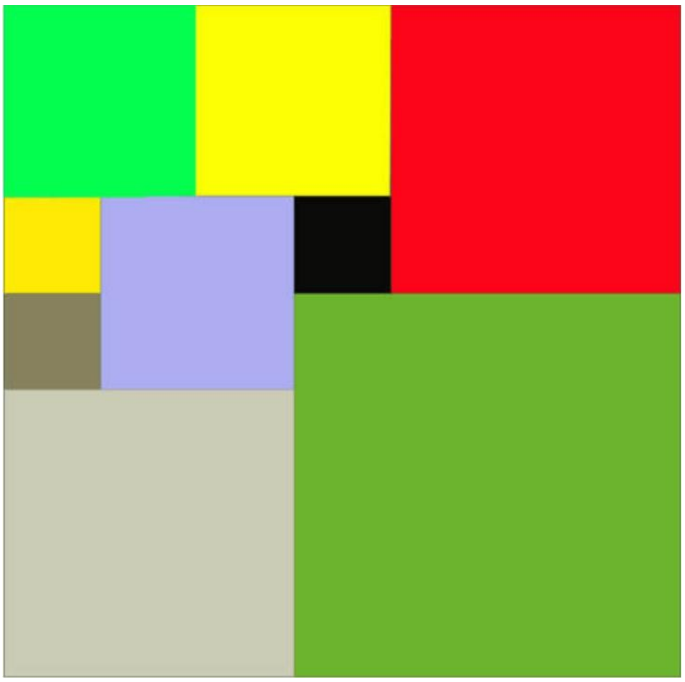
Цель работы.

Ознакомиться с алгоритмом перебора с возвратом и научиться применять его на практике. Написать программу реализовывающую поиск с возвратом.

Постановка задачи.

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≤N≤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≤x, y≤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

Индивидуальное задание.

Вариант 5р.

Рекурсивный бэктрекинг. Возможность задать список квадратов (от 0 до N^2 квадратов в списке), которые обязательно должны быть использованы в покрытии квадрата со стороной N.

Описание алгоритма.

На вход программе подается число от 2 до 40. Если число имеет тип int и входит в заданный промежуток, то программа начинает работу, иначе программа сообщит об ошибке.

Также пользователю необходимо задать количество обязательных квадратов, которые будут участвовать в покрытии столешницы. Если число не будет типа int, тогда программа выведет сообщение об ошибке.

Таким образом, если количество обязательных квадратов больше 0, то программа продолжит работу, выделит память для одномерного массива, в котором будут храниться стороны обязательных квадратов, и попросит пользователя ввести список сторон. Тип массива равен ReqSquares, это структура, которая позволяет хранить сторону и значение координат верхнего левого угла для обязательных квадратов. Также необходимо заметить, что далее осуществляются три проверки необходимые для верной работы алгоритма. Во-первых, что сторона обязательно должна быть числом. Во-вторых, данное число должно быть больше 0, но входить в размер столешницы. В-третьих, выполняется цикл, в котором проверяется, не ввёл ли пользователь квадраты, которые могут выйти за пределы столешницы при попытке их размещения.

Далее происходит установка предела рекурсии. Он подбирается таким образом, чтобы был учтён худший случай разбиения квадрата. В случае, когда число является простым и есть обязательные квадраты, предел будет равен 1 + (tabletopSize - 1) + tabletopSize. Это число говорит о том, что столешница

состоит из одного квадрата со стороной на 1 меньше, чем площадь столешницы, и квадратов площадью 1, занимающих оставшееся пространство. Если квадрат является непростым числом, и пользователь не задал обязательные квадраты, то предел равен 4 (если наименьшим делителем для стороны квадрата является 2), он также может быть равен 6 (если наименьшим делителем для стороны квадрата является 3), или 8 (если наименьшим делителем для стороны квадрата является 5). Это обусловлено тем, что минимальному разбиению для непростых чисел N (Напр.: 4, 9, 15) будет соответствовать разбиение квадрата с длиной стороны равной наименьшему целочисленному делителю числа N не равному единице. Наименьшими целочисленными делителями, учтёнными в программе, являются 2, 3 и 5. Разбиение для таких квадратов равно 4, 6 и 8, таким образом и появляется значение предела.

Так, с помощью функции primeOrNot находится этот самый делитель, если он есть, и далее идет работа уже либо с квадратом длины равной этому делителю, либо с прежним квадратом, если наименьший делитель равен единице и число является простым.

Сам квадрат в памяти программы представлен в виде двумерного массива arr или arrReq. Кроме того, существует стандартный стек squares с типом элементов Square, где Square это структура, которая будет содержать информацию о квадрате, местоположение его левого верхнего угла относительно начала массива arr или arrReq и длину самого квадрата.

Далее необходимо заполнить массив arrReq обязательными квадратами если они есть, для того чтобы рекурсивная функция учла их. Программа начнёт искать свободное место в массиве, чтобы последовательно их разместить, начиная с начала координат.

Для того чтобы можно было проследить за ходом программы, на экран будет выведен массив arrReq с обязательными квадратами, в случае, если они добавлены.

Далее элементы limit, &squares, arr (arrReq), tabletopSize (primeFlag), 0 (recursionCounter) из main подаются в рекурсивную функцию поиска минимального количества квадратов.

В самой рекурсии происходит следующее:

Проверка на достижение предельной глубины рекурсии. Если это предел, то возвращается -1. Осуществляется поиск пустой клетки в массиве arr. Если таковой не оказалось, то возвращается 0. Создаются дополнительные стеки tmpSquares и maxSquares. В первый будут записываться текущие квадраты, а во второй их минимальная последовательность. Далее происходит нахождение максимальной длины квадрата, который можно поместить, начиная с данной точки левого верхнего угла, с помощью функции findLength. Пробуются разные длины квадрата. Для этого происходит заполнение квадрата функцией insertSquare. Рекурсивное обращение из функции в эту же функцию, только с счетчиком рекурсии +1.

Возращенное значение количества квадратов записывается в переменную tmp. Происходит проверка значения tmp на минимальное и, если это так, запоминаем длину текущего квадрата, и переписываем maxSquares значениями из tmpSquares. Если же нет, то опустошаем tmpSquares. Далее очищаем квадрат и переходим к следующей длине квадрата. Когда достигнем максимальной длины квадрата maxLength, копируем в squares значения из maxSquares, записываем в squares текущий квадрат и возвращаем minNum.

Сложность алгоритма.

Алгоритм имеет экспоненциальную временную сложность $O(2^n)$.

Для хранения обязательных квадратов и самой столешницы, которую необходимо заполнить, используется два динамических двумерных массива. Следовательно, по памяти сложность равна $O(n)$.

Описание структур данных.

Структура *Square* представляет из себя данные о квадрате, который участвует в разбиении. В ней содержатся поля — *int x, int y, int size.*

Структура *ReqSquare* представляет из себя данные о квадрате, который точно участвует в разбиении и задаётся пользователем. В ней содержатся поля - *int x, int y, int size.*

Массивы *int** arr* и i*nt** arrReq.* Оба массива хранят в себе столешницу. Если область пуста — элементы массива заполняются нулями, иначе числами равными размеру квадрата, который заполняет данную область. Отличие лишь в том, что в массиве *arrReq* обязательные квадраты могут быть заданы заранее. Массив *arr* изначально пуст.

Массив *ReqSquare *reqSquares.* Используется для считывания обязательных квадратов и заполнения массива *arrReq.*

Контейнер из stl *stack <Square> squares.* Хранит в себе данные о квадратах, которые участвуют в минимальном разбиении.

Контейнер из stl *stack <Square> tmpSquares.* Хранит в себе данные о текущих квадратах, т. е. это квадраты, которые используются для сравнения разных вариантов разбиения.

Контейнер из stl *stack <Square> maxSquares.* Хранит в себе данные о квадратах, которые являются возможным минимальным разбиением.

Описание функций.

int primeOrNot(int tabletopSize) — функция, которая определяет, является ли число простым. Необходима для установки предела рекурсии и, в некоторых случаях, позволяет оптимизировать программу. *int tabletopSize* — число, для которого нужно узнать, является ли оно простым (размер столешницы). Возвращаемым результатом является: 2, 3, 5 (наименьшие делители непростого числа), или 1, если число является простым.

*void insertSquare(int **arr, int x, int y, int lengthSquare)* — функция, которая заполняет пустую область. *int **arr* — массив, хранящий столешницу; *int x, int y, int lengthSquare* - координаты левого верхнего угла вставляемого квадрата и его длина.

void stackCopy(stack <Square> squares, stack <Square>* squaresCopy)* — функция копирования стеков. *squares* — копируемый стек, *squaresCopy* — стек, в который копируют.

*bool findEmptySpace(int** arr, int& x, int& y, int size)* — функция нахождения пустого пространства на столешнице для вставки квадрата. *int** *arr* - массив, хранящий столешницу; *int& x, int& y, int size* — координаты и размер области, которую необходимо проверить. Возвращаемыми значениями являются: true — если проверяемая область свободна, false — если часть области занята (или если область полностью занята).

*int findLength(int** arr, int& x, int& y, int size)* — находит максимальную длину квадрата, который можно вставить в область на столешнице *int** arr*, с координатами и размером *int& x, int& y, int size*. Возвращает наибольшую длину.

int minSquares(int limit, stack <Square> squares, int** arr, int size, int recursionCounter)* — рекурсивная функция, используемая для перебора возможных вариантов расстановки квадратов. *int limit* — предел рекурсии, *stack <Square>* squares* — массив, который будет содержать результат разбиения, *int** *arr* - столешница, *int size* — размер столешницы, *int recursionCounter* — счётчик рекурсии. Возвращаемым результатом будет минимальное количество квадратов, участвующих в разбиении.

Тестирование.

№ теста	Тест	Результат
1	2 2 1 1	Array with required squares 1 0 1 0 ---------- Result is: 4 Required squares is: 1 1 1 1 2 1 ---------- Split squares is: 2 1 1 2 2 1 ---------- Result tabletop splitting is: 1 1 1 1 ----------
2	3 1 2	Array with required squares 2 2 0 2 2 0 0 0 0 ---------- Result is: 6 Required squares is: 1 1 2

		---------- Split squares is: 1 3 1 2 3 1 3 1 1 3 2 1 3 3 1 ---------- Result tabletop splitting is: 2 2 1 2 2 1 1 1 1 ----------
3	7 3 1 2 3	Array with required squares 1 0 0 0 0 0 0 2 2 0 0 0 0 0 2 2 0 0 0 0 0 3 3 3 0 0 0 0 3 3 3 0 0 0 0 3 3 3 0 0 0 0 0 0 0 0 0 0 0 ---------- Result is: 12 Required squares is: 1 1 1 1 2 2 1 4 3 ----------

		Split squares is: 1 7 1 2 1 1 2 7 1 3 1 3 3 7 1 4 4 4 6 1 2 6 3 1 7 3 1 ---------- Result tabletop splitting is: 1 1 3 3 3 2 2 2 2 3 3 3 2 2 2 2 3 3 3 1 1 3 3 3 4 4 4 4 3 3 3 4 4 4 4 3 3 3 4 4 4 4 1 1 1 4 4 4 4 ----------
4	5 1 4	Result is: 10 Required squares is: 1 1 4 ---------- Split squares is: 1 5 1 2 5 1 3 5 1 4 5 1

		5 1 1 5 2 1 5 3 1 5 4 1 5 5 1 ---------- Result tabletop splitting is: 4 4 4 4 1 4 4 4 4 1 4 4 4 4 1 4 4 4 4 1 1 1 1 1 1 ----------
5	6 3 2 2 2	Array with required squares 2 2 0 0 0 0 2 2 0 0 0 0 2 2 0 0 0 0 2 2 0 0 0 0 2 2 0 0 0 0 2 2 0 0 0 0 ---------- Result is: 6 Required squares is: 1 1 2 1 3 2 1 5 2 ---------- Split squares is:

		3 1 4 3 5 2 5 5 2 ---------- Result tabletop splitting is: 2 2 4 4 4 4 2 2 4 4 4 4 2 2 4 4 4 4 2 2 4 4 4 4 2 2 2 2 2 2 2 2 2 2 2 2 ----------
6	2 0	Result is: 4 Required squares is: ---------- Split squares is: 1 1 1 1 2 1 2 1 1 2 2 1 ----------
7	3 0	Result is: 6 Required squares is: ---------- Split squares is: 1 1 2 1 3 1 2 3 1 3 1 1

		3 2 1 3 3 1 ----------
8	5 0	Result is: 8 Required squares is: ---------- Split squares is: 1 1 3 1 4 2 3 4 2 4 1 2 4 3 1 5 3 1 5 4 1 5 5 1 ----------

Выводы.

В ходе выполнения лабораторной работы был изучен и применен на практике алгоритм перебора с возвратом.

ПРИЛОЖЕНИЕ А

Исходный код программы

```
#include <iostream>
#include <stack>
#include <cmath>
using namespace std;

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

struct ReqSquare{
    int x;
    int y;
    int size;
};

int primeOrNot(int tabletopSize, int limit){
    if (tabletopSize % 2 == 0) {
        return 2;
    }
    if (tabletopSize % 3 == 0) {
        return 3;
    }
    if (tabletopSize % 5 == 0) {
        return 5;
    }
    return 1;
}

void insertSquare(int **arr, int x, int y, int lengthSquare) {
    for (int i = 0; i < lengthSquare; i++) {
        for (int j = 0; j < lengthSquare; j++) {
            arr[x + i][y + j] = lengthSquare;
        }
    }
}

void stackCopy(std::stack <Square>* squares, std::stack <Square>*
squaresCopy) {
    while (!squares->empty()) {
        squaresCopy->push(squares->top());
        squares->pop();
    }
```

```
}

bool findEmptySpace(int** arr, int& x, int& y, int size) {
    for (x = 0; x < size; x++) {
        for (y = 0; y < size; y++) {
              if (arr[x][y] == 0 || arr[x][y] == -1) { //если нашли пустое
мест
                return true;
            }
        }
    }
     return (x == size) ? (false) : (true);//не нашли пустое место - false,
иначе true
}

int findLength(int** arr, int& x, int& y, int size) {
    int distance_to_border = (size - x > size - y) ? (size - y - !arr[0][0])
: (size - x - !arr[0][0]);
    int maxLength = 2; //начальная длина квадрата
    for (; maxLength <= distance_to_border; maxLength++) {
        //пока длина квадрата меньше или равна длине до границы, увеличиваем
длину квадрата
        for (int i = 0; i < maxLength; i++) {
             if (arr[x + maxLength - 1][y + i] || arr[x + i][y + maxLength -
1])
                return maxLength - 1;
        }
    }
    return maxLength - 1;
}

// рекурсивная функция для перебора возможных значений расстановки квадратов
int minSquares(int limit, stack <Square>* squares, int** arr, int size, int
recursionCounter) {
      if (limit < recursionCounter) ///если предел рекурсии меньше кол-ва
входов рекурсии
        return -1;
    int x, y;
        if (!findEmptySpace(arr, x, y, size)) ///если не нашли пустое
пространство
        return 0;

    stack <Square> tmpSquares;
    stack <Square> maxSquares;

      int maxLength = findLength(arr, x, y, size); //находим максимальную
длину квадрата, который можно поместить
    //в свободное пространство
```

```
    int tmpLength;
    int minNum = limit;
    int tmp;
    int desiredLength = 1;

    for (tmpLength = 1; tmpLength <= maxLength; tmpLength++) {
        insertSquare(arr, x, y, tmpLength);
         tmp = minSquares(limit, &tmpSquares, arr, size, recursionCounter +
1) + 1;

        if (tmp <= minNum && tmp != 0) {
            minNum = tmp;
            desiredLength = tmpLength;
            while (!maxSquares.empty()) {
                maxSquares.pop();
            }
            stackCopy(&tmpSquares, &maxSquares);
        }
        else {
            while (!squares->empty()) {
                squares->pop();
            }
        }
        for (int i = 0; i < tmpLength; i++)
            for (int j = 0; j < tmpLength; j++)
                arr[x + i][y + j] = 0;
    }

    while (!squares->empty())
        squares->pop();
    stackCopy(&maxSquares, squares);
    squares->push({x, y, desiredLength});

    return minNum;
}

int main() {
    int tabletopSize = 0;
    int reqSquaresNumber = 0;
    int reqSquareSide = 0;
    ReqSquare *reqSquares = nullptr;
    bool reqFlag = false;
    int primeFlag = 0;
    int limit = 0;
    int res = 0;
    stack <Square> squares; //стек структур
    int **arr = nullptr;
    int **arrReq = nullptr;
```

```
    std::cout << "Enter tabletop size" << std::endl;
       if (!(std::cin >> tabletopSize)) {//пока не будет введено число,
выполняем цикл
        std::cout << "Error: size isn't an integer" << std::endl;
        return 0;
    }
       if (tabletopSize < 2 || tabletopSize > 20) {//если размер не
удовлетворяет условию задачи
        std::cout << "Error: tabletop have wrong size" << std::endl;
        return 0;
    }
    //обязательные квадраты - в любом случае участвуют в покрытии
    std::cout << "Enter number of required squares" << std::endl;
     if (!(std::cin >> reqSquaresNumber)) {//пока не будет введено число,
выполняем цикл
        std::cout << "Error: number of required squares isn't an integer" <<
std::endl;
        return 0;
    }
    if (reqSquaresNumber > 0) {
        arrReq = new int* [tabletopSize];
        for (int i = 0; i < tabletopSize; i++)
            arrReq[i] = new int[tabletopSize]();

        reqFlag = true;
         if (reqSquaresNumber < 0 || reqSquaresNumber > pow(tabletopSize,2))
{//если кол-во обязательных квадратов больше N^2 или отрицательное число
                std::cout << "Error: wrong number of required squares" <<
std::endl;
            return 0;
        }
        reqSquares = new ReqSquare[reqSquaresNumber];
          std::cout << "Enter a list of required squares. Format: side" <<
std::endl;
        for (int i = 0; i < reqSquaresNumber; i++) {
                if (!(std::cin >> reqSquareSide)) {//пока не будет введено
число, выполняем цикл
                    std::cout << "Error: side of required square isn't an
integer" << std::endl;
                return 0;
            }
             if (reqSquareSide < 0 || reqSquareSide >= tabletopSize) {//если
сторона обязательного квадрата >= N или отрицательное число
                    std::cout << "Error: wrong side of required square" <<
std::endl;
                return 0;
            }
            for (int j = 0; j < i; j++) {
                if (reqSquareSide + reqSquares[j].size > tabletopSize) {
```

```
                    std::cout << "Error: you can not use such a square" <<
std::endl;
                    return 0;
                }
            }
            reqSquares[i].size = reqSquareSide;
        }
    }

    if (!reqFlag){
        primeFlag = primeOrNot(tabletopSize, limit);
        if (primeFlag){
            limit = 12.15 * sqrt(sqrt(sqrt(sqrt((tabletopSize)))));
        }
        if (primeFlag == 2){
            limit = 4;
        }
        if (primeFlag == 3){
            limit = 6;
        }
        if (primeFlag == 5){
            limit = 8;
        }
    }
    else {
        limit = 1 + (tabletopSize - 1) + tabletopSize;
    }

    if (reqFlag){
        int x = 0, y = 0;
        for (int i = 0; i < reqSquaresNumber; i++){
            if (!findEmptySpace(arrReq, x, y, tabletopSize)){
                cout << "Error: space isn't empty\n";
                return 0;
            }
            if ((x + reqSquares[i].size) > tabletopSize){
                cout << "Error: the square goes beyond the tabletop\n";
                return 0;
            }
            if ((y + reqSquares[i].size) > tabletopSize){
                cout << "Error: the square goes beyond the tabletop\n";
                return 0;
            }
            int reqLength = reqSquares[i].size;
            reqSquares[i].x = x;
            reqSquares[i].y = y;
            insertSquare(arrReq, x, y, reqLength);
            insertSquare(arrReq, x, y, reqLength);
        }
```

```
    }
    //функция вывода массива с квадратами на экран
    if (reqFlag) {
        cout << "Array with required squares" << endl;
        for (int i = 0; i < tabletopSize; i++) {
            for (int j = 0; j < tabletopSize; j++) {
                cout << arrReq[j][i] << " ";
            }
            cout << endl;
        }
        cout << "----------------------------" << endl;
        cout << endl;
    }

    if (primeFlag > 1) {
        arr = new int* [primeFlag];
        for (int i = 0; i < primeFlag; i++)
            arr[i] = new int[primeFlag];
        res = minSquares(limit, &squares, arr, primeFlag, 0);
    }
    if (primeFlag == 1){
        arr = new int* [tabletopSize];
        for (int i = 0; i < tabletopSize; i++)
            arr[i] = new int[tabletopSize];
        res = minSquares(limit, &squares, arr, tabletopSize, 0);
    }
    if (reqFlag){
        res = minSquares(limit, &squares, arrReq, tabletopSize, 0);
    }

    cout << "Result is: " << res + reqSquaresNumber << endl;

    cout << "Required squares is:" << endl;
    for (int i = 0; i < reqSquaresNumber; i++){
        cout << reqSquares[i].x + 1 << " " << reqSquares[i].y + 1 << " " <<
reqSquares[i].size << endl;
    }
    cout << "--------------------" << endl;

    Square tmp{};
    int flagCounter = 1;

    if (reqFlag) {
        cout << "Split squares is:" << endl;
        while (!squares.empty()) {
            if (flagCounter > res) {
                squares.pop();
                continue;
            }
```

```
            tmp = squares.top();
              cout << tmp.x + 1 << " " << tmp.y + 1 << " " << tmp.size <<
endl;
            insertSquare(arrReq, tmp.x, tmp.y, tmp.size);
            squares.pop();
            flagCounter++;
        }
        cout << "------------------" << endl;

        cout << "Result tabletop splitting is:" << endl;
        for (int i = 0; i < tabletopSize; i++) {
            for (int j = 0; j < tabletopSize; j++) {
                cout << arrReq[j][i] << " ";
            }
            cout << endl;
        }
        cout << "----------------------------" << endl;
        cout << endl;
        for (int i = 0; i < tabletopSize; i++)
            delete[] arrReq[i];
        delete[] arrReq;
    }
    else {
        int scale = (primeFlag != 1) ? tabletopSize / primeFlag : 1;
        cout << "Split squares is:" << endl;
        while (!squares.empty()) {
            if (flagCounter > res) {
                squares.pop();
                continue;
            }
            tmp = squares.top();
             std::cout << tmp.x * scale + 1 << " " << tmp.y * scale + 1 << "
" << tmp.size * scale << std::endl;
            insertSquare(arr, tmp.x, tmp.y, tmp.size);
            squares.pop();
            flagCounter++;
        }
        cout << "------------------" << endl;
        for (int i = 0; i < tabletopSize; i++)
            delete[] arr[i];
        delete[] arr;
    }

    return 0;
```