МИНОБРНАУКИ 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. 1304 Шаврин А.П.

Преподаватель Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучить алгоритм поиска с возвратом (Backtracking). Решить задачу разбиения квадрата на минимальное количество квадратов меньшего размера с использованием алгоритма поиска с возвратом.

Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 11 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 (рисунок 1).

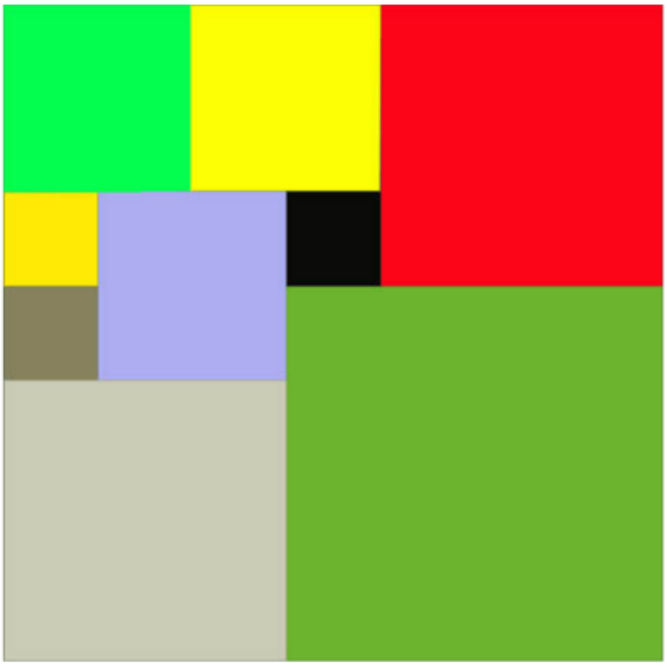

Рисунок 1. Разбиение столешницы размера 7×7 на 9 квадратов.

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

Выполнение работы.

Для решения данной задачи были написаны следующие блоки кода:

1. Функция main, не принимающая входных параметров и реализующая основную логику программы (считывание значения N, создание объекта класса Table, который будет описан ниже, вызов метода класса для решения данной задачи и вызов метода класса для вывода решения).

2. Был объявлен тип Square = std::vector < std::vector <int> >, являющийся вектором векторов с целочисленными значениями. Далее он будет использоваться в качестве метрики для стола.

3. Был написан класс (структура) Poin, имеющая только два публичных поля:

a. Int x – целочисленная координата x.

b. Int y - целочисленная координата y.

4. После был описан класс Table, реализующий хранение всех необходимых полей и методов для решения задачи.

В классе были объявлены следующие приватные поля:

a. Int side – целочисленный размер стороны стола, являющийся наименьшим делителем изначально переданного размера. Данная модификация сделана для ускорения решения задачи.

b. Int factor – целочисленный коэффициент изменения, на который был уменьшен размер стола. Данный параметр необходим для модификации ответа.

c. Square square – площадь стола типа Square, описанного ранее. Данная матрица хранит следующие значения:

- 0, если данная координата свободна
- -1, если данная координата занята
- 1 – N-1, если в этой координате располагается левый верхний угол квадрата.

d. Square best_square – площадь стола типа Square, описанного ранее. Данная матрица хранит наилучшее разбиение стола с теми же обозначениями, что были описаны в предыдущем пункте.

e. Int count_squares – целочисленное значение квадратов, на которые разбит стол.

f. Int record – минимально-возможное целочисленное количество квадратов, необходимых для разбиения стола.

В классе были объявлены следующие приватные методы:

g. void Simplification() - метод, реализующий упрощение задачи, за счет уменьшения стороны.

h. void CreateSquare(Square *square) - метод, реализующий начальное заполнение нулями метрик square и best_square и принимает указатель на нужную метрику.

i. void AddSquare(int start_x, int start_y, int width) - метод, реализующий добавление квадрата в метрику square. Метод принимает в качестве аргументов следующие параметры:

- int start_x – координата x левого верхнего угла добавляемого квадрата.
- int start_y - координата y левого верхнего угла добавляемого квадрата.
- int width – ширина стороны добавляемого квадрата

j. void RemoveSquare(int start_x, int start_y, int width) - метод, реализующий удаление квадарата из метрики square. Метод принимает в качестве аргументов следующие параметры:

- int start_x – координата x левого верхнего угла удаляемого квадрата.
- int start_y - координата y левого верхнего угла удаляемого квадрата.
- int width – ширина стороны удаляемого квадрата

k. int BestStart() - метод, реализующий первоначальную лучшую расстановку из 3х квадратов (рисунок 2).

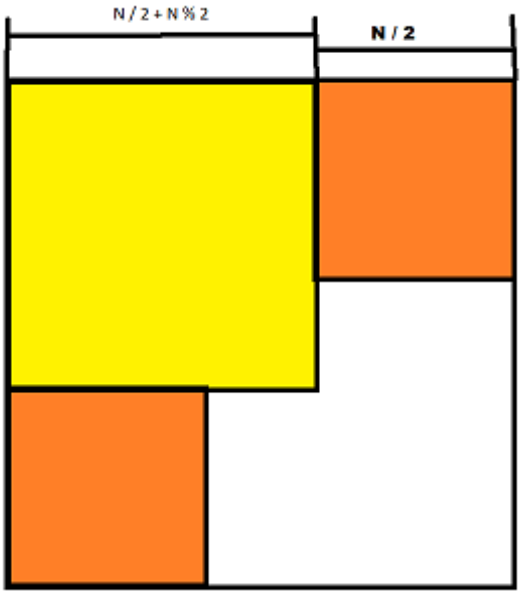

Рисунок 2. Лучшее начальное разбиение.

Метод возвращает целочисленное значение стороны самого большого вписанного квадрата.

l. bool IsPossibleAddition(int start_x, int start_y, int width) - метод, проверяющий возможность добавить квадрат в метрику square. Метод принимает в качестве аргументов следующие параметры:

- int start_x – координата x левого верхнего угла добавляемого квадрата.
- int start_y - координата y левого верхнего угла добавляемого квадрата.
- int width – ширина стороны добавляемого квадрата

Метод возвращает True, если добавление возможно и False, если это невозможно.

m. void Backtracking(int start_x, int start_y, int end) - метод реализующий перебор всех вариантов расстановки способом backtracking. Метод принимает в качестве аргументов следующие параметры:

- int start_x – координата x левого верхнего угла добавляемого квадрата.

- int start_y - координата y левого верхнего угла добавляемого квадрата.
- int end – граница области заполнения, как по x так и по y

n. Point* FindNewAdditionPoint(int start_x, int start_y, int end, int width) - метод, реализует нахождение новой позиции для запуска backtracking. Метод принимает в качестве аргументов следующие параметры:

- int start_x – координата x левого верхнего угла добавляемого квадрата.
- int start_y - координата y левого верхнего угла добавляемого квадрата.
- int width – ширина стороны добавляемого квадрата

Метод возвращает указатель на объект класса Point, в котором хранятся следующие координаты для запуска Backtracking. В случае если таких координат не нашлось, в объекте класса хранятся значения (-1, -1).

В классе были объявлены следующие публичные методы:

o. Table(int N) - конструктор класса, принимающий на вход размер стола.

p. void Assembly() - метод, реализующий решение задачи.

q. void PrintAnswer() - метод, реализующий вывод результата в необходимом формате.

5. Задача решается следующим образом. При создании объекта в конструкторе происходит заполнение первоначальными значениями всех переменных, а сторона стола заменяется на ее наименьший делитель, если такой есть, иначе она остается неизменной. При вызове метода класса Assembly, внутри метода происходит вызов приватного метода BestStart, а возвращаемое им значение записывается в локальную целочисленную переменную best_first_width. После происходит проверка на четность стороны квадрата, если

это так, то происходит вызов добавления последнего квадрата с координатой (best_first_widthб best_first_width) и такой же шириной и на этом решение окончено. Если это не так, то происходит вызов метода Backtracking, с передачей ему в качестве первоначальных значений координат – являющихся результатом целочисленного деления стороны, а в качестве значения конца – размер стола. Функций Backtracking работает таким образом, сначала она проверяет на оптимальность решения (текущее разбиение еще не больше и не равно минимальному рекорду), затем проверяет занята ли клетка с координатами, переданными функции в качестве аргументов. Если клетка занята, вызывается функция FindNewAdditionPoint, которая ищет новую свободную клетку построчно в метрике square, если клетка нашлась (значение не -1, -1), то backtracking вызывается рекурсивно, иначе идет обновление рекорда и метрики best_square. Если клетка оказалась не занята, то вычисляется максимально возможная сторона квадрата, который можно вписать, и идет цикл перебора сторон от максимальной до 1. Внутри цикла происходит проверка на возможность добавления квадрата и его добавление или же переход к новой длине стороны. После добавления квадрата происходит то же, что и в случае, когда левый верхний угол был занят (вызывается функция FindNewAdditionPoint и т.д.).

Исходный код программы (см. Приложение А)

Тестирование (см. Раздел Тестирование)

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные
1.	10	4 1 1 5 1 6 5 6 1 5 6 6 5
2.	3	6 1 1 2 1 3 1 2 3 1 3 1 1 3 2 1 3 3 1
3.	9	6 1 1 6 1 7 3 4 7 3 7 1 3 7 4 3 7 7 3
4.	11	11 1 1 6 1 7 5 6 7 3 6 10 2 7 1 5 7 6 1 8 6 1 8 10 1 8 11 1 9 6 3

		9 9 3

Выводы.

Изучен метод поиска с возвратом (backtracking), а также рассмотрены различные способы модификации данного метода с целью уменьшения затрат памяти и времени работы алгоритма.

Разработана программа, выполняющая считывание с клавиатуры исходных данных и команды пользователя и реализующая решение задачи по определению минимального количества квадратов для представления квадрата большего размера, а также их оптимальная расстановка.

Для оптимизации были использованы следующие решения:

- Разбиение задачи на 3 случая (четная сторона, нечетная составная сторона, простая сторона).
- Хранение ответа, в матрице разбиения квадрата.
- Начальная оптимальная вставка.
- Ограничение лишней рекурсии путем определения рекорда.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Backtracking.cpp

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


// Square - вектор векторов (двумерный массив), который является столом
typedef std::vector < std::vector <int> > Square;


struct Point{
    /*
        Point - класс точки, задающий координаты
    */

    int x;  // координата x
    int y;  // координата y
};


class Table{
    /*
        Table - класс стола, который реализует хранение всех необходимых параметров и методов для решения задачи
    */

    private:
        int side;             // размер стола (минимальный делитель реального размера для ускорения)
        int factor;          // коэфициент изменения, отвечающий за то, во сколько раз изменена сторона стола

        Square square;         // площадь стола (используется как метрика)
```

```
            Square best_square; // лучшая площадь стола, хранящая
наилучшее разбиение

            int count_squares;   // текущее количество квадратов, на
которые разбит стол
            int record;          // минимальное количество квадратов,на
которое разбит стол

            void                                    Simplification();
// метод, реализующий упрощение задачи, за счет уменьшения стороны
            void             CreateSquare(Square              *square);
// метод, реализующий начальное заполнение нулями метрик (square and
best_square)
            void  AddSquare(int  start_x,  int  start_y,  int  width);
// метод, реализующий добавление квадрата в метрику (square)
            void RemoveSquare(int start_x, int start_y, int width);
// метод, реализующий удаление квадрата из метрики (square)
            int                                             BestStart();
// метод, реализующий первоначальную лучшую расстановку из 3х квадратов
            bool  IsPossibleAddition(int  start_x,  int  start_y,  int
width);                // метод, проверяющий возможность добавить квадрат
в метрику (square)
            void  Backtracking(int  start_x,  int  start_y,  int  end);
//  метод  реализующий  перебор  всех  вариантов  расстановки  способом
backtracking
            Point* FindNewAdditionPoint(int start_x, int start_y, int
end,  int  width);   // метод, реализует нахождение новой позиции для
запуска backtracking

        public:
            Table(int N);       // конструктор класса
            void Assembly();    // метод, реализующий решение задачи
            void PrintAnswer(); // метод, реализующий вывод результата
    };


    void Table::Simplification(){
        /*
```

```
            В методе происходит нахождение минимального делителя для
размера стола и его коэфициент изменения (для ускорения)
        */

        for (int deriv = 2; deriv <= this->side / 2; deriv++){
            if (this->side % deriv == 0){
                this->factor = this->side / deriv;
                this->side = deriv;
                break;
            }
        }
    };


    Table::Table(int N){
        /*
            Конструктор класса, реализующий задание начальных
параметров
            Args:
                - N (int): Размер стола
        */

        this->side = N;
        this->factor = 1;

        this->Simplification();
        this->CreateSquare(&this->square);
        this->CreateSquare(&this->best_square);

        this->count_squares = 0;
        this->record = this->side * this->side;
    };


    void Table::CreateSquare(Square *square){
        /*
            Конструктор метрики, реализующий создание и заполнение
метрики нулями
            Args:
```

```
                - square (Square *): Указатель на метрику
        */

        square->reserve(this->side);
        for (int y = 0; y < this->side; y++){
            square->emplace_back(std::vector <int>());
            square->at(y).reserve(this->side);
            for (int x = 0; x < this->side; x++){
                square->back().push_back(0);
            }
        }
    };


    void Table::AddSquare(int start_x, int start_y, int width){
        /*
            Метод, реализующий добавление квадрата в метрику square и
заполнение необходимой площади значением -1,
            а левый верхний угол квадрата значением размера квадрата
            Args:
                - start_x (int): Координата x левого верхнего угла
квадрата
                - start_y (int): Координата y левого верхнего угла
квадрата
                - width (int): Размер добавляемого квадрата (его ширина)
        */

        this->count_squares++;
        for (int y = start_y; y < start_y + width; y++){
            for (int x = start_x; x < start_x + width; x++){
                if (this->square.at(y).at(x) == 0){
                    this->square.at(y).at(x) = -1;
                }
            }
        }
        this->square.at(start_y).at(start_x) = width;
    };
```

```
void Table::RemoveSquare(int start_x, int start_y, int width){
    /*
        Метод, реализующий удаление квадрата из метрику square и заполнение необходимой площади значением 0
        Args:
            - start_x (int): Координата x левого верхнего угла квадрата
            - start_y (int): Координата y левого верхнего угла квадрата
            - width (int): Размер удаляемого квадрата (его ширина)
    */

    this->count_squares--;
    for (int y = start_y; y < start_y + width; y++){
        for (int x = start_x; x < start_x + width; x++){
            if (this->square.at(y).at(x) != 0){
                this->square.at(y).at(x) = 0;
            }
        }
    }
};


int Table::BestStart(){
    /*
        Метод, реализующий заполнение метрики square 3мя квадратами, всегда входящими в лучшее решение
        Returns:
            - best_first_width (int): Размер самого большого вписанного квадрата
    */

    int best_first_width = (this->side / 2) + (this->side % 2);

    this->AddSquare(0, 0, best_first_width);
    this->AddSquare(0, best_first_width, this->side - best_first_width);
    this->AddSquare(best_first_width, 0, this->side - best_first_width);
```

```
        return best_first_width;
    };


    bool Table::IsPossibleAddition(int start_x, int start_y, int
width){
        /*
            Метод, проверяющий возможность добавления квадрата в
метрику square с учетом выхода за границы
            и занятость вставляемой области
            Args:
                - start_x (int): Координата x левого верхнего угла
квадрата
                - start_y (int): Координата y левого верхнего угла
квадрата
                - width (int): Размер добавляемого квадрата (его ширина)
            Returns:
                - True/False (bool): Возможность или не возможность
добавления
        */

        // проверка выхода за границы
        if ((start_x + width > this->side) || (start_y + width > this-
>side)){
            return false;
        }

        // проверка занятости области для добавления
        for (int y = start_y; y < start_y + width; y++){
            for (int x = start_x; x < start_x + width; x++){
                if (this->square.at(y).at(x) != 0){
                    return false;
                }
            }
        }

        return true;
    };
```

```
void Table::Backtracking(int start_x, int start_y, int end){
    /*
        Метод, выполняющий рекурсивный перебор всех возможный вариантов размешения квадратов в заданой области
        Args:
            - start_x (int): Координата x левого верхнего угла области заполнения
            - start_y (int): Координата y левого верхнего угла области заполнения
            - end (int): Граница области заполнения, как по x так и по y
    */


    // проверка не оптимальности решения
    if (this->count_squares >= this->record){
        return;
    }

    // проверка свободности начальной координаты для вставки
    if (this->square.at(start_y).at(start_x) == 0){

        // расчет максимально возможного размера вставляемого квадрата
        int max_width = std::min(end - start_x, end - start_y);

        // перебор размеров вставляемого квадрата
        for (int width = max_width; width >= 1; width--){
            // проверка возможности добавления
            if (this->IsPossibleAddition(start_x, start_y, width)){
                this->AddSquare(start_x, start_y, width);

                // поиск точки для перехода на новую координату
                Point* point = this->FindNewAdditionPoint(start_x, start_y, end, width);

                // проверка на незаполненность метрики square
```

```
                    if ((point->x != -1) && (point->y != -1)){
                        this->Backtracking(point->x, point->y, end);
                    } else {
                        // запись нового рекорда и лучшего разбиения
                        this->record = this->count_squares;
                        this->best_square = this->square;
                    }
                    delete point;
                    this->RemoveSquare(start_x, start_y, width);
                }
            }
        } else {
            // поиск точки для перехода на новую координату
            Point* point = this->FindNewAdditionPoint(start_x, start_y,
end, 1);

            // проверка на незаполненность метрики square
            if ((point->x != -1) && (point->y != -1)){
                this->Backtracking(point->x, point->y, end);
            } else {
                // запись нового рекорда и лучшего разбиения
                this->record = this->count_squares;
                this->best_square = this->square;
            }
            delete point;
        }
    };


    Point* Table::FindNewAdditionPoint(int start_x, int start_y, int
end, int width){
        /*
            Метод, выполняющий поиск новой координаты левого верхнего
угла для вставки квадрата
            Args:
                - start_x (int): Координата x левого верхнего угла
области заполнения
                - start_y (int): Координата y левого верхнего угла
области заполнения
```

```
                - end (int): Граница области заполнения, как по x так и
по y
                - width (int): размер последнего вставленного квадрата
            Returns:
                - point (Point*): Пара координат для запуска
рекурсивного backtracking
        */

        Point *point = new Point();
        point->x = -1;                    // значение x для обозначения
неудачи при поиске
        point->y = -1;                    // значение y для обозначения
неудачи при поиске

        // заполнение текущей строки пока это возможно
        if (start_x + width < end){
            point->x = start_x + width;
            point->y = start_y;
        } else {
            // переход к новой строке, в самое начало по координате x
            if (start_y + 1 < end){
                point->x = this->side / 2;
                point->y = start_y + 1;
            }
        }
        return point;
    };


    void Table::Assembly(){
        /*
            Метод, выполняющий решение задачи разбиения стола
        */

        int best_first_width = this->BestStart();

        // проверка на четность стороны квадрата (для ускорения)
        if (this->side == 2){
```

```
            this->AddSquare(best_first_width, best_first_width, this->side - best_first_width);
            this->record = this->count_squares;
            return;
        }

        this->Backtracking(this->side / 2, this->side / 2, this->side);
        this->square = this->best_square;
    };


    void Table::PrintAnswer(){
        /*
            Метод, выполняющий вывод решения задачи в необходимом формате
        */

        std::cout << this->record << std::endl; // вывод минимального количества квадратов в разбиении

        // вывод координат квадратов в разбиении
        for (int y = 0; y < this->side; y++){
            for (int x = 0; x < this->side; x++){
                if (this->square.at(y).at(x) > 0){
                    std::cout << (y * this->factor + 1) << ' ' << (x * this->factor + 1) << ' ' << (this->square.at(y).at(x) * this->factor) << std::endl;
                }
            }
        }
    };


    int main(){
        /*
            Главная функция, выполняющая работу всей программы
        */

        int N;
```

```
    std::cin >> N;
    Table table = Table(N);
    table.Assembly();
    table.PrintAnswer();

    return 0;
}
```