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. 1384 _____ Тапеха В.А.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучить алгоритм поиска с возвратом и его оптимизации для конкретной задачи.

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 (см. рисунок 1).

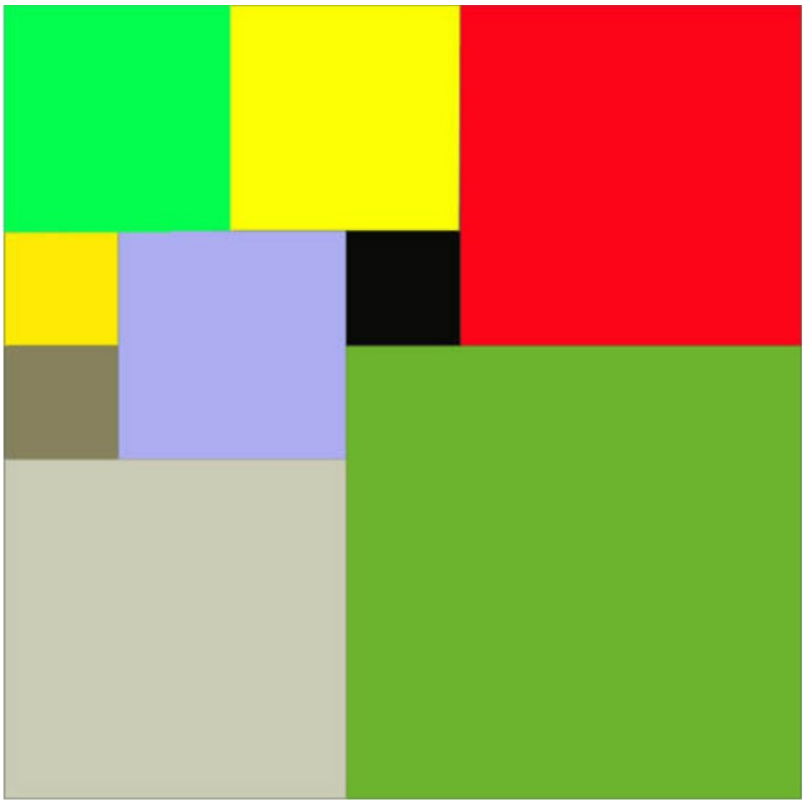

Рисунок 1 - Пример столешницы 7×7

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 ($2 \le N \le 20$).

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 ($1 \le x, y \le$ 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.

Сперва были реализованы классы *Square* и *Matrix*.

Первый класс описывает параметры некоторого квадрата и взаимодействия с ними. В нем хранится поля *std::pair<int, int> coordinates*, которое содержит координаты, и поле *int size*, отвечающее за размер стороны квадрата. Метод *void scale_square(int input_n, int decreased_n)* масштабирует координаты и размер квадрата(размер квадрата на время выполнения алгоритма можно уменьшить благодаря одной из оптимизаций, поэтому необходимо иметь возможность вернуть первоначальный вид параметров квадратов). Здесь же реализован метод *void change_coordinate(int& coordinate, int input_n, int decreased_n),* который масштабирует одну из координат. Он нужен для того, чтобы избежать дублирования кода. Также в классе есть объявление дружественности для функции, перегружающей оператор вывода. При его переопределении есть прямое обращение к закрытым полям класса, поэтому и необходимо объявить дружественность.

Второй класс – класс булевой матрицы. Нужен для отслеживания занятых областей в исходном квадрате. В нем хранится двумерный массив из элементов типа *bool*, который и является булевой матрицей. Метод *void fill_area(int x, int y, int size)* – метод, заполняющий/освобождающий область матрицы. Также был реализован метод *[[nodiscard]] bool check_area(int x, int y, int size) const*, который проверяет свободна ли область заданная область матрицы другим квадратом. В этом же классе реализован геттер *[[nodiscard]] bool get_cell(int x, int y) const*, возвращающий значение элемента матрицы, который находится в заданных координатах.

Также был реализован третий класс – Squaring. Он решает поставленную задачу, то есть он квадрирует квадрат. Он хранит в себе int n – размер изначального квадрата(исходный или уменьшенный), Matrix table – булевая матрица, позволяющая отслеживать занятую область исходного квадрата, int

best_count – минимальное количество квадратов, из которых можно построить исходный квадрат, std::vector<Square> best_arr – массив лучшего решения(хранит координаты и размер обрезков), std::vector<Square> cur_arr – массив, который содержит текущее решение, выстроенное перебором. Основным методом является void squared_table(), в котором вызываются оптимизирующие методы и бэктрекинг. Сам метод с перебором с возвратом – void backtracking(int cur_count, int free_space). Также есть оптимизирующий метод void decrease_size() – метод, уменьшающий исходный размер квадрата(нужен для оптимизации алгоритма). В этом же классе были написаны геттеры, возвращающие best_count и best_array.

Задачу можно разделить на 2 случая: исходное число – простое и исходное число – составное. В первом случае возможен только оптимизированный перебор всех случаев. Если же число составное, то можно утверждать, что квадртирование исходного квадрата с размером, равным ему, будет вести себя так же, как и квадратирование квадрата с размером наименьшего делителя этого числа.

Оптимизация:

1. Исходя из того что составные числа ведут себя так же, как и их наименьший делитель для их достаточно сделать перебор для наименьшего делителя составного числа, а потом масштабировать итоговый массив.
2. Можно заметить, что во всех раскрасках (для простых N) присутствует один большой квадрат размера (N+1)/2 и два смежных ему квадрата размера N-(N+1)/2. Поэтому можно поставить на первом шаге рекурсии сразу 3 этих квадрата, сократив всё древо решений на 75%.
3. Так как есть информация о количестве поставленных квадратов на текущем шаге, то можно прерывать обход тех ветвей, где количество квадратов это значение превосходит.

4. Также отслеживается свободная площадь в квадрате. Благодаря этому была ускорена проверка заполненности квадрата и появилась возможность пропускать квадраты, которые не помещаются в оставшуюся область.
5. Еще одной оптимизацией является то, что можно не рассматривать все решения, когда квадрат еще не заполнен и при этом и он меньше лучшего решения всего на единицу, так как в любом случае минимум в итоге этого решения будет получено такое же количество квадратов, как и в лучшем случае.

Последняя оптимизация является ключевой, так как только после ее внесения была решена усложненная задача(N <= 40).

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в таблице 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	36	4 1 1 18 1 19 18 19 1 18 19 19 18	Программа работает правильно
2.	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	Программа работает правильно
3.	39	6 1 1 26 1 27 13 27 1 13 14 27 13 27 14 13 27 27 13	Программа работает правильно

Выводы.

Был изучен алгоритм поиска с возвратом и его оптимизация для конкретной задачи. В результате работы была написана программа, рекурсивно выполняющая задачу о минимально возможном разбиении квадратного поля на квадраты, используя алгоритм поиска с возвратом и удовлетворяя ограничениям по времени и памяти. Для того, чтобы уменьшить время работы программы были изучены и применены при выполнении задачи методы оптимизации изученного алгоритма.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include "Squaring.h"

int main() {
    int n;
    std::cin >> n;

    Squaring solve(n);
    solve.squared_table();
    auto result = solve.get_best_array();
    int res_count = solve.get_best_count();

    std::cout << res_count << '\n';
    for (size_t i = 0; i < res_count; ++i) {
        std::cout << result.at(i);
    }

    return 0;
}
```

Название файла: Matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

#include <vector>

template<class T>
using matrix = std::vector<std::vector<T>>;

// Класс булевой матрицы
// Нужен для отслеживания занятых областей в исходном квадрате
class Matrix {
public:
    // Конструктор
    explicit Matrix(int size = 0);

     // Геттер, возвращающий значение элемента матрицы, который
находится в заданных координатах
    [[nodiscard]] bool get_cell(int x, int y) const;
     // Метод, проверяющий занята ли область матрицы другим квад-
ратом
    [[nodiscard]] bool check_area(int x, int y, int size) const;

    // Метод, заполняющий/освобождающий область матрицы
    void fill_area(int x, int y, int size);
private:
    matrix<bool> table; // булевая матрица
```

```
};


#endif //MATRIX_H
```

Название файла: Matrix.cpp

```
#include "Matrix.h"

// Инициализация пустой матрицы(все значения равны false).
Matrix::Matrix(int size) {
    for (size_t i = 0; i < size; ++i) {
        table.emplace_back(size, false);
    }
}

// Возвращает значение элемента матрицы.
bool Matrix::get_cell(int x, int y) const {
    return table.at(y).at(x);
}

// Проверка области на наличие других квадратов на ней.
bool Matrix::check_area(int x, int y, int size) const {
    for (int i = y; i < y + size; ++i) {
        for (int j = x; j < x + size; ++j) {
            if (table.at(i).at(j))
                return false;
        }
    }

    return true;
}

// Изменяются все значения в некоторой области на противополож-
ные.
void Matrix::fill_area(int x, int y, int size) {
    for (int i = y; i < y + size; ++i) {
        for (int j = x; j < x + size; ++j) {
            table.at(i).at(j).flip();
        }
    }
}
```

Название файла: Square.h

```
#ifndef SQUARE_H
#define SQUARE_H

#include <iostream>

// Класс, описывающий параметры квадрата и взаимодействия с ними
class Square {
public:
```

```
        // Конструктор
        explicit Square(int x = 0, int y = 0, int size = 0)
                : coordinates(x, y), size(size) {}

        // Метод, масштабирующий координаты и размер квадрата
        void scale_square(int input_n, int decreased_n);
        // Объявление дружественности для функции, перегружающей оператор вывода
            friend std::ostream& operator<<(std::ostream &os, const Square& sq);
    private:
        // Метод, масштабирующий координату
          void change_coordinate(int& coordinate, int input_n, int decreased_n);

         std::pair<int, int> coordinates; // Координаты левого верхнего угла квадрата(x;y)
        int size; // Размер стороны квадрата
    };


    #endif // SQUARE_H
```

Название файла: Square.cpp

```
    #include "Square.h"

    // Масштабирование
    void Square::scale_square(int input_n, int decreased_n) {
        change_coordinate(coordinates.first, input_n, decreased_n);
        change_coordinate(coordinates.second, input_n, decreased_n);
        size = size * input_n / decreased_n;
    }

    // Масштабирование одной из координат
    void Square::change_coordinate(int &coordinate, int input_n, int decreased_n) {
        coordinate = coordinate * input_n / decreased_n + 1;
    }

    // Перегрузка оператора <<
    std::ostream& operator<<(std::ostream &os, const Square& sq) {
              return os << sq.coordinates.first << ' ' << sq.coordinates.second << ' ' << sq.size << '\n';
    }
```

Название файла: Squaring.h

```
    #ifndef SOLUTION_H
    #define SOLUTION_H

    #include "Matrix.h"
    #include "Square.h"
```

```
    // Класс, квадрирующий исходный квадрат
    class Squaring {
    public:
        // Конструктор
         // Всегда можно поставить 2 * n квадратов, поэтому инициали-
зируем best_count = 2 * n + 1
        explicit Squaring(int size)
             :  n(size),  table(size),  best_count(2  *  n  +  1),
best_arr(best_count), cur_arr(best_count) {}

        // Метод, квадратирующий исходный квадрат(решение задачи)
        void squared_table();

        // Геттер массива, хранящего лучшее решение.
        [[nodiscard]] std::vector<Square> get_best_array() const;
         // Геттер, возвращающий минимальное количество квадратов, из
которых строится исходный квадрат
        [[nodiscard]] int get_best_count() const;

    private:
        // Метод, реализующий бэктрекинг
        void backtracking(int cur_count, int free_space);
        // Метод, уменьшающий исходный размер квадрата
        void decrease_size();
         // Метод, масштабирующий массив, в котором хранится лучшее
решение
        void scale_array(int input_n);

        int n; // Размер изначального квадрата(исходный или уменьшен-
ный)
        Matrix table; // Булевая матрица, позволяющая отслеживать за-
нятую область исходного квадрата
        int best_count; // Минимальное количество квадратов, из кото-
рых можно построить исходный квадрат
         std::vector<Square> best_arr; // Массив лучшего решения(хра-
нит координаты и размер обрезков)
         std::vector<Square> cur_arr; // Массив, который содержит те-
кущее решение, выстроенное перебором
    };


    #endif
```

Название файла: Squaring.cpp

```
    #include "Squaring.h"

    // Возвращает минимальное количество квадратов
    int Squaring::get_best_count() const {
        return best_count;
    }

    // Возвращает массив, в котором хранится лучшее решение
```

```
std::vector<Square> Squaring::get_best_array() const {
    return best_arr;
}

void Squaring::squared_table() {
    int input_n = n;
    // Квадраты со стороной n, где n - составное число,
    // квадрируются так же, как и квадраты,
     // у которых размер их стороны равен наименьшему делителю стороны исходного квадрата.
    // Поэтому для оптимазации можно попробовать уменьшить размер исходного квадрата.
    decrease_size();
     // Изначально в матрице размера nxn, отслеживающей занятые области исходного квадрата,
    // свободны все элементы(n * n элементов).
    int free_space = n * n;

     // Можно увидеть, что в квадрат размера n, где n - простое число,
      // всегда можно добавить первым и самым большим квадрат размера (n + 1) / 2
    // И рядом с ним добавить квадраты размером на единицу меньше
    cur_arr.at(0) = Square(0, 0, (n + 1) / 2);
    cur_arr.at(1) = Square(0, (n + 1) / 2, n - (n + 1) / 2);
    cur_arr.at(2) = Square((n + 1) / 2, 0, n - (n + 1) / 2);

     // Для того, чтобы отслеживать заполняемость исходного квадрата,
    // помечаются области соответствующие добавленным квадратам в матрице,
    // которая отслеживает занятые области исходного квадрата.
    table.fill_area(0, 0, (n + 1) / 2);
    table.fill_area(0, (n + 1) / 2, n - (n + 1) / 2);
    table.fill_area((n + 1) / 2, 0, n - (n + 1) / 2);

    // Вычитается площадь занятой области
    free_space -= ((n + 1) / 2) * ((n + 1) / 2);
    free_space -= 2 * (n - ((n + 1) / 2)) * (n - ((n + 1) / 2));

     // Полный перебор всех возможных вариантов квадрирования с учетом оптимизаций.
    backtracking(3, free_space);
     // Масштабирование параметров уменьшенного исходного квадрата.
    scale_array(input_n);
}

void Squaring::backtracking(int cur_count, int free_space) {
    // Если результат хуже или равен лучшему результату,
    // то нет смысла продолжать дальше перебор.
    if (best_count <= cur_count)
        return;

    // Подобрано новое лучшее решение.
```

```
        if (free_space == 0) {
            best_count = cur_count;
            best_arr = cur_arr;
            return;
        }

        // Если текущее решение до начала перебора только на единицу
меньше лучшего,
        // то оно минимум будет такое же, как и лучшее.
        if (cur_count == best_count - 1)
            return;

        // Начало перебора
        for(int x = 0; x < n; ++x) {
            for(int y = 0; y < n; ++y) {
                if (!table.get_cell(x, y)) {
                    // Подбор максимального размера квадрата с учетом
всех ограничений.
                     int max_side = std::min(n - 1, std::min(n - x, n
- y));
                    for(int side = max_side; side > 0; --side) {
                         // Если площадь этого квадрата не превышает
количество свободных клеток,
                        // то можно попробовать его поставить.
                        if (side * side <= free_space) {
                             // Если по текущим координатам область в
матрице свободна,
                            // то можно поставить квадрат
                            if (table.check_area(x, y, side)) {
                                table.fill_area(x, y, side);
                                 cur_arr.at(cur_count) = Square(x, y,
side);
                                        backtracking(cur_count + 1,
free_space - side * side);
                                table.fill_area(x, y, side);
                            }
                        }
                    }
                     // Если не было поставлено ни одного квадрата в
текущих координатах,
                     // то можно не рассматривать дальше решения этой
ветки
                    if (!table.get_cell(x, y))
                        return;
                }
            }
        }
    }

    // Масштабируются все элементы массивы лучшего решения.
    void Squaring::scale_array(int input_n) {
        for (size_t i = 0; i < best_count; ++i) {
            best_arr.at(i).scale_square(input_n, n);
        }
    }
```

```
    // Рассматриваются только 2, 3, 5 в качестве делителей,
    // так как все составные числа <= 40 содержат хотя бы один из
них.
    void Squaring::decrease_size() {
        if (n % 2 == 0) {
            n = 2;
        }
        else if (n % 3 == 0) {
            n = 3;
        }
        else if (n % 5 == 0) {
            n = 5;
        }
    }
```