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

Студентка гр. 9383 _____ Лапина А.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения исходного квадрата минимальным количеством квадратов.

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. (рисунок 1)

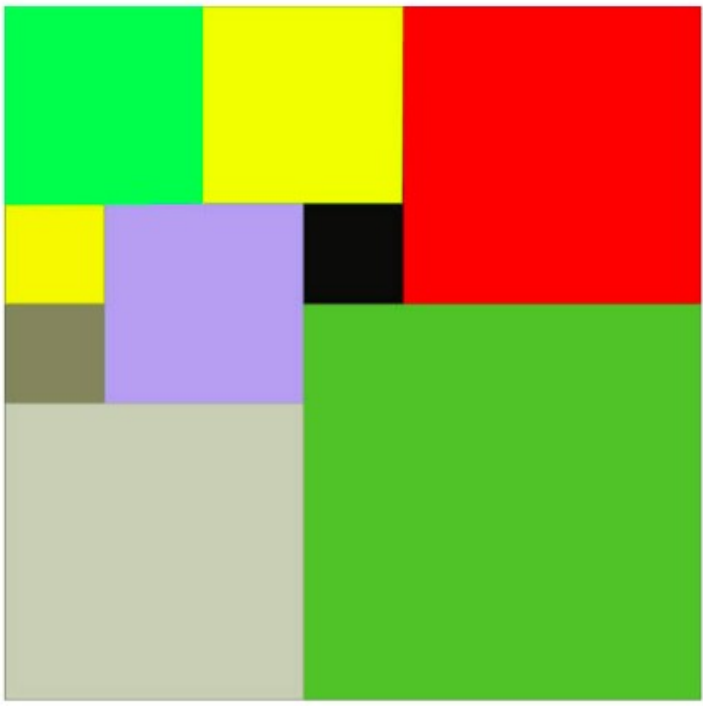

Рисунок 1- пример построения столешницы из 9 обрезков

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

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

Вариант 2и.

Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

Основные теоретические положения.

Бэктрекинг (поиск с возвратом) – это общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве.

Описание метода:

Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

Недостатки:

Метод поиска с возвратом является универсальным. Достаточно легко проектировать и программировать алгоритмы решения задач с использованием этого метода. Однако время нахождения решения может быть очень велико даже при небольших размерностях задачи (количестве исходных данных), причём настолько велико (может составлять годы или даже века), что о практическом применении не может быть и речи. Поэтому при проектировании таких алгоритмов, обязательно нужно теоретически оценивать время их работы на конкретных данных. Существуют также задачи выбора, для решения которых можно построить уникальные, «быстрые» алгоритмы, позволяющие быстро получить решение даже при больших размерностях задачи. Метод поиска с возвратом в таких задачах применять неэффективно.

Выполнение работы:

В программе использованы следующие структуры:

struct Square — для хранения информации о квадрате (координаты его верхнего левого угла по x и y и ширина квадрата)

struct For_backtracking — для хранения данных для итеративного бэктрекинга: текущая заполненность исходного квадрата (map), количество квадратов и вектор из квадратов, которые уже размещены на исходном квадрате.

В программе были реализованы следующие функции:

* void Print_Answer(const int result, List_of_squares result_squares) — печатает результат работы программы (количество квадратов, их координаты и длины);

*void New_Square(Matrix& matrix, Square square) - добавляет новый квадрат на исходный;

*bool isCan_Add_square(int n, Matrix matrix, Square square) — определяет можно ли вместить новый квадрат на исходный;

* void Backtracking(int n, Matrix matrix, int count_ready_squares, int& result, List_of_squares& result_squares) — функция для итеративного бэктрекинга. Где n — длина исходного квадрата, map — информация о заполненности исходного квадрата, count_ready_squares — количество квадратов, result —количество квадратов для записи результата (в ней будет хранится ответ на задачу), result_squares — вектор размещенных квадратов (для ответа);

*void Divisible_2(int n, int&result, List_of_squares& result_squares) — функция для четных чисел (оптимизация 1);

*void Divisible_3(int n, int&result, List_of_squares& result_squares) - функция для рассмотрения чисел, кратных 3 (оптимизация 2);

*void Simple_Number(int n, Matrix& matrix, int&result, List_of_squares& resultSquares) — функция для всех оставшихся (простых) чисел;

*void Time_Information(clock_t start_time, clock_t end_time) - функция для анализа времени работы программы.

Описание алгоритма работы программы:

Программа на вход получает число n — сторона исходного квадрата, измеряем время до начала работы программы, результат записываем в переменную start_time. Заводим переменную result, в которой будем хранить минимальное число квадратов, изначально оценим результат как 3+2*(n/2)+1 = 3+n+1, так как для простых чисел мы изначально фиксируем 3 квадрата (помечены как 1, 2 и 3), а остальное пространство заполняем как на рисунке 2 (пример для числа 7) — то есть 2 стороны по n/2 (помечены * на рисунке) и остается еще 1 квадрат (зеленый)

			*	2	2	2
			*	2	2	2
			*	2	2	2
*	*	*	1	1	1	1
3	3	3	1	1	1	1
3	3	3	1	1	1	1
3	3	3	1	1	1	1

Рисунок 2 — пример выбора начального значения для квадрата со стороной 7

Затем, в зависимости он значения n выполняем действия:

1) если число кратно 2 — четное, то обращаемся к функции Divisible_2

2) если число кратно 3, то обращаемся к Divisible_3

3) иначе (остались только простые числа (до n <=20)) — обращаемся к Simple_Number, в которой фиксируется 3 квадрата, а после вызывается функция для бектрекинга - Backtracking, в которой перебираются варианты заполнения оставшегося пространства и выбирается оптимальный.

В программе были использованы следующие оптимизации:

1) Было замечено, что для чисел, кратных 2, минимальное количество квадратов равно 4. Для этого реализована функция Divisible_2.
2) А для чисел, кратных 3, минимальное количество квадратов равно 6. Для этого реализована функция Divisible_3.

Анализ времени работы программы

Благодаря вышеописанным оптимизациям программа для чисел кратных 2 и 3 будет работать быстро, поэтому рассмотрим время работы программы для простых чисел:

Длина исходного квадрата	Время работы
5	0.000371
7	0.001208
11	0.012885
13	0.024968
17	0.172427
19	0.806468

На рисунке 3 приведен график зависимости времени работы программы в секундах в зависимости от длины стороны исходного квадрата (n <= 20):

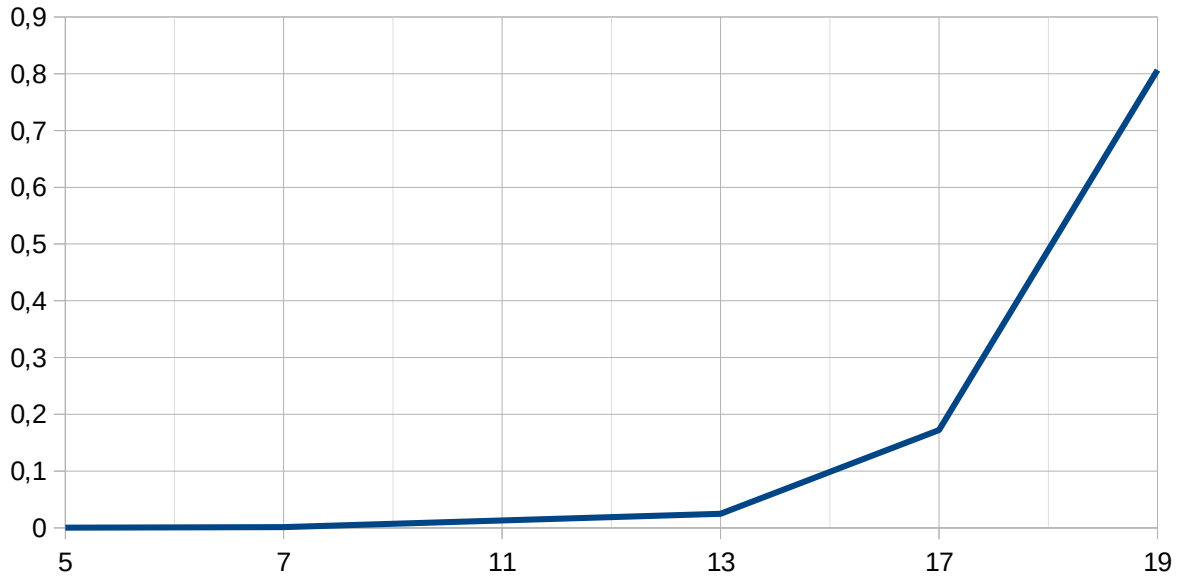

Рисунок 3 — График зависимости работы программы от длины стороны квадрата

Проанализировав полученный график, можно сделать вывод, что время работы алгоритма экспоненциально зависит от размера квадрата.

Тестирование

1) Входные данные: 5

Выходные данные:

8

1 1 2

1 3 1

2 3 1

3 1 1

3 2 1

3 3 3

4 1 2

1 4 2

0.000469

2) Входные данные: 11

Выходные данные:

11

1 1 4

1 5 2

3 5 2

5 1 2

5 3 2

5 5 1

5 6 1

6 5 1

6 6 6

7 1 5

1 7 5

0.013596

3) Входные данные: 8

Выходные данные:

4

1 1 4

1 5 4

5 1 4

5 5 4

0.000156

4) Входные данные: 9

Выходные данные:

6

1 1 3

1 4 3

1 7 3

4 1 3

7 1 3

4 4 6

0.00021

Также были реализованы тесты:

1) Для проверки корректности работы функции isCan, которая проверяет можно ли поместить квадрат. Для этого были рассмотрены случаи, когда функция работает корректно, когда на вход поступает квадрат, который нельзя поместить (пересечение) и случай, когда квадрат выходит за пределы исходного квадрата.

2) Для проверки добавления квадрата — функции New_Square. Проверка осуществляется следующим образом — после добавления нового квадрата, смотрим заполнились ли нужные поля на карте «1».

Результаты работы написанных тестов предоставлены на рисунке 4

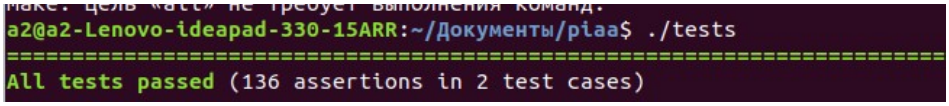

Рисунок 4 — Результаты тестов программы

Разработанный программный код см. в приложении А.

Вывод.

Применен на практике алгоритм поиска с возвратом для заполнения квадрата минимальным количеством квадратов. Придумана оптимизация, для чисел, кратных 2 и 3.

Приложение А
Исходный код программы

Название файла: main.cpp

```
#include "lb1.hpp"

int main() {
    clock_t start_time = clock();
    int n;
    std::cin >> n;
    int result = 3+n+1;
    List_of_squares result_squares;  //для итоговых значений
    if (n % 2 == 0) {  //для чисел, делящихся на 2
        Divisible_2(n, result, result_squares);
    }
    else if (n % 3 ==0){//для чисел, делящихся на 3
        Divisible_3(n, result, result_squares);
    }
    else{               //для оставшихся (простых) чисел
        Matrix matrix(n);
        for (int i = 0; i < n; i++)
            matrix[i] = std::vector<int>(n, 0);
        Simple_Number(n, matrix, result, result_squares);
    }
    Print_Answer(result, result_squares);
    clock_t end_time = clock();
    Time_Information(start_time, end_time);
    return 0;
}
```

Название файла: lb1.cpp

```
#include "lb1.hpp"

void Print_Answer(const int result, List_of_squares result_squares)
{
    std::cout << result << "\n";
    for (size_t i = 0; i < result_squares.size(); ++i) {
                std::cout  <<  result_squares[i].x  +  1  <<  "  "  <<
result_squares[i].y + 1<< " " << result_squares[i].width <<"\n";
    }
}

void New_Square(Matrix& matrix, Square square) { //добавляет новый
квадрат
    for (int i = square.x; i < square.x + square.width; i++)
        for (int j = square.y; j < square.y + square.width; j++)
            matrix[i][j] = 1;
}

bool  isCan_Add_Square(int  n,  Matrix  matrix,  Square  square){
//можно ли вместить квадрат
    int w = square.width;
           return  (square.x  +  w  <  n)&&(square.y  +  w  <
n)&&(matrix[square.x][square.y  +  w]  ==  0)&&(matrix[square.x  +  w]
```

```
[square.y]   ==   0)&&(matrix[square.x   +   w][square.y   +   w]   ==
0)&&(square.x>=0)&&(square.y>=0);
      }

      void Backtracking(int n, Matrix matrix, int count_ready_squares,
int& result, List_of_squares& result_squares) {
          List_of_squares interim_squares; //для промежуточных значений
          std::stack<For_backtracking> st;
              st.emplace(matrix,  count_ready_squares,  interim_squares);
//добавляем элемент
          while (!st.empty()){
              For_backtracking last = st.top();  //берем верхний элемент

              st.pop();  //удаляем верхний

              bool A = true;
              Point point = {-1, -1};
              for (int x = 0; x < n / 2 + 1; x++) {
                  for (int y = 0; y < n / 2 + 1; y++) {
                      if (last.matrix[x][y] == 0) {
                          point = {x, y};
                          A = false;
                          break;
                      }
                  }
                  if (point.first > -1) {
                      break;
                  }
              }

              if (A) {
                  result = last.count;
                  result_squares = last.list_of_square;
              }

              if (last.count + 1 >= result){
                  continue;
              }

              for (int w = 0; w < n - 1; w++) {

                  Point start = {point.first, point.second};

                                      if  (isCan_Add_Square(n,  last.matrix,
Square(point.first,point.second, w))) {
                           for (int x = start.first; x <= point.first + w; x+
+) {
                              last.matrix[x][point.second + w] = last.count +
1;
                      }
                        for (int y = start.second; y <= point.second + w;
y++) {
                            last.matrix[point.first + w][y] = last.count +
1;
                      }
                      start.first++;
                      start.second++;
```

```
                        last.list_of_square.push_back(Square({point.first,
point.second, w + 1}));
                                    st.emplace(last.matrix,  last.count  +  1,
last.list_of_square);   //добавили элемент
                        last.list_of_square.pop_back();
                    }
                    else {
                        break;
                    }
                }
            }
        }

        void     Divisible_2(int     n,     int&result,     List_of_squares&
result_squares) {   //делящиеся на 2
            result_squares = {     { 0, 0, n / 2 }, { 0, n / 2, n / 2}, { n
/ 2, 0, n / 2}, { n / 2, n / 2, n/ 2} };
            result = 4;
        }

        void     Divisible_3(int     n,     int&result,     List_of_squares&
result_squares) { //делящиеся на 3
            result_squares = {     { 0,  0,  n/3 },  { 0,  n/3,  n/3},  { 0,
2*(n/3),  n/3  },  {  n/3,  0,  n/3},  {  2*(n/3),  0,  n/3},  {  n/3,  n/3,
2*(n/3)} };
            result = 6;
        }

        void    Simple_Number(int    n,    Matrix&    matrix,    int&result,
List_of_squares& resultSquares) { // n - простое число
            List_of_squares ready_squares;   //для начальных квадратов

            New_Square(matrix, Square(n / 2, n / 2, n / 2 + 1));
            ready_squares.emplace_back(n / 2, n / 2, n / 2 + 1);

            New_Square(matrix, Square(n - n / 2, 0, n / 2));
            ready_squares.emplace_back(n - n / 2, 0, n / 2);

            New_Square(matrix, Square(0, n - n / 2, n / 2));
            ready_squares.emplace_back(0, n - n / 2, n / 2);

                  Backtracking(n,   matrix,   ready_squares.size(),   result,
resultSquares);
                        copy(ready_squares.begin(),    ready_squares.end(),
back_inserter(resultSquares));
        }

        void Time_Information(clock_t start_time, clock_t end_time) {
            std::cout << (float)(end_time- start_time) / CLOCKS_PER_SEC <<
"\n";
        }
```

Название файла: lb1.hpp

```
        #include <bits/stdc++.h>
```

```
    struct Square {   //структура для хранения квадратов
        Square(int x, int y, int width){
            this->x = x;    //координата по x
            this->y = y;    //координата по y
            this->width = width;    //ширина квадрата
        }
        int x = 0;
        int y = 0;
        int width = 0;
    };

    using List_of_squares = std::vector<Square>;
    using Matrix = std::vector<std::vector<int>>;
    using Point = std::pair<int, int>;

    struct For_backtracking {
           For_backtracking(Matrix matrix, int count, List_of_squares
list_of_square){
            this->matrix = matrix;
            this->count = count;
            this->list_of_square = list_of_square;
        }
        Matrix matrix;
        int count;
        List_of_squares list_of_square;
    };

    void New_Square(Matrix& matrix, Square square);
    void    Print_Answer(const    int    result,    List_of_squares
result_squares);
    bool isCan_Add_Square(int n, Matrix matrix, Square square);
    void Backtracking(int n, Matrix matrix, int count_ready_squares,
int& result, List_of_squares& result_squares);

    void    Divisible_2(int    n,    int&result,    List_of_squares&
result_squares); //для чисел, кратных 2
    void    Divisible_3(int    n,    int&result,    List_of_squares&
result_squares); //для чисел, кратных 3

    void   Simple_Number(int   n,   Matrix&   matrix,   int&result,
List_of_squares& resultSquares); //для простых чисел

    void Time_Information(clock_t start_time, clock_t end_time); //для
вычисления времени
```

Название файла: tests_programm.cpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "../source/lb1.hpp"

TEST_CASE("Тест для функции isCan") {
  int n = 8;
  Matrix matrix = Matrix(n);
  for (int i = 0; i < n; i++)
  matrix[i] = std::vector<int>(n, 0);
```

```
        SECTION("Выход за пределы") {
             REQUIRE(isCan_Add_Square(n,  matrix,  Square(5,  4,  15))  ==
false);
            REQUIRE(isCan_Add_Square(n,  matrix,  Square(-100,  100,  2))  ==
false);
             REQUIRE(isCan_Add_Square(n,  matrix,  Square(3,  50,  4))  ==
false);
            REQUIRE(isCan_Add_Square(n,  matrix,  Square(20,  20,  5))  ==
false);
             REQUIRE(isCan_Add_Square(n,  matrix,  Square(3,  -9,  3))  ==
false);
            REQUIRE(isCan_Add_Square(n,  matrix,  Square(200,  7,  10))  ==
false);
           REQUIRE(isCan_Add_Square(n,  matrix,  Square(100,  -200,  20))  ==
false);
        }

        SECTION("Пересечения") {
          Square square(3, 3, 4);
          New_Square(matrix, square);
          REQUIRE(isCan_Add_Square(n, matrix, Square(2, 3, 3)) == false);
          REQUIRE(isCan_Add_Square(n, matrix, Square(3, 3, 3)) == false);
        }

        SECTION("Корректная работа") {
          Square square(0, 0, 3);
          New_Square(matrix, square);
          REQUIRE(isCan_Add_Square(n, matrix, Square(3, 3, 3)) == true);
          REQUIRE(isCan_Add_Square(n, matrix, Square(0, 3, 2)) == true);
          REQUIRE(isCan_Add_Square(n, matrix, Square(5, 5, 1)) == true);
        }

      }

      TEST_CASE("Тест для функции New_Square") {
        int n = 12;
        Matrix matrix = Matrix(n);
        for (int i = 0; i < n; i++)
          matrix[i] = std::vector<int>(n, 0);

        SECTION("Средний случай") {
          Square square(3, 4, 3);
          New_Square(matrix, square);
          for (int i = square.x; i < square.x + square.width; i++)
            for (int j = square.y; j < square.y + square.width; j++)
              CHECK(matrix[i][j] != 0);
        }

        SECTION("Нижний правый угол") {
          Square square(8, 8, 4);
          New_Square(matrix, square);
          for (int i = square.x; i < square.x + square.width; ++i)
            for (int j = square.y; j < square.y + square.width; ++j)
              CHECK(matrix[i][j] != 0);
        }

        SECTION("Нижний левый угол") {
          Square square(7, 0, 5);
```

```
  New_Square(matrix, square);
  for (int i = square.x; i < square.x + square.width; ++i)
    for (int j = square.y; j < square.y + square.width; ++j)
      CHECK(matrix[i][j] != 0);
}
SECTION("Верхний левый угол") {
  Square square(0, 0, 7);
  New_Square(matrix, square);
  for (int i = square.x; i < square.x + square.width; i++)
    for (int j = square.y; j < square.y + square.width; j++)
      CHECK(matrix[i][j] != 0);
}

SECTION("Верхний правый угол") {
  Square square(0, 7, 5);
  New_Square(matrix, square);
  for (int i = square.x; i < square.x + square.width; i++)
    for (int j = square.y; j < square.y + square.width; j++)
      CHECK(matrix[i][j] != 0);
}
```