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

Студент гр. 8304 _____ Воропаев А.О.
Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы

Научиться реализовывать алгоритм бэктрекинга, анализировать его сложность и затраты по памяти. Исследование различных оптимизаций в алгоритмах бэктрекинга.

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*N . Он может получить ее, собрав из уже имеющихся обрезков(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.

Расширение.

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.

Шаг 1. Вводится сторона квадрата N. Для данного квадрата создается массив размера N * N, который представляет собой поле квадрата (все клетки изначально пустые, следовательно, каждая клетка имеет значение false). Создаются два вектора структур, хранящих координаты квадрата – tmp(для текущих расчетов) и min(для хранения “минимального” случая). Также вводится кол-во необходимых квадратов в разбиении. Если было введено число больше нуля, то также вводятся длины сторон квадратов, необходимых в разбиении. Начинается полный перебор вариантов для нахождения необходимого разбиения; если был введен 0, то алгоритм ищет минимальное разбиение квадрата.

Шаг 2. Создается текущая структура для хранения нового квадрата, начинается поиск пустой клетки. Если пустая клетка не найдена и не указано

разбиение, то данный квадрат заполнен и запоминается как минимальный случай, алгоритм возвращается к предыдущему шагу. Если же разбиение было указано, то перед тем, как запомнить его в качестве минимального случая, вызывается функция, проверяющая присутствуют ли все необходимые квадраты в найденном разбиении.

Шаг 3. Производится попытка подстановки в текущую клетку квадрата со стороной от 1 до N-1. Если квадрат выходит за границы главного квадрата, происходит возврат, так же если текущий массив структур (квадратов) становится больше чем минимальный массив структур (квадратов) происходит возврат, т.к. нет смысла продолжать алгоритм.

Шаг 4. Проверяем, есть ли пустое место для нового квадрата. Если хоть одна клетка из покрытия новым квадратом непустая, то возвращаемся к предыдущему вызову алгоритма. Если все клетки пусты, то заполняем их новым квадратом, который сохраняется в массив структур tmp. Производим вызов алгоритма для остаточных клеток, если такие имеются.

Шаг 5. После возврата к текущей точке рекурсивно стираем квадрат, чтобы попробовать вставить квадрат побольше.

Шаг 6. Алгоритм завершается, когда просмотрены, проанализированы либо откинуты (из-за большого кол-ва квадратов в разбиении) все варианты разбиений исходного квадрата.

Оптимизации.

Если на каком-либо этапе работы алгоритма кол-во квадратов начинает превышать n+4, где n – это сторона квадрата, то данное разложение считается неверным.

Если n – не простое число, то находится его минимальный кратный делитель. Затем находится минимальное разложения для этого делителя. А координаты каждого квадрата разложения умножаются на n/delimeter.

Описание основных структур данных и функций.

Программа считывает размер квадрата с консоли и выводит результат в консоль.

1) struct element – структура, содержащая координаты и длину сторон для квадрата для заполнения.

x – координата левого верхнего угла квадрата по оси X

y – координата левого верхнего угла квадрата по оси Y

len – длина стороны квадрата

2) void backtracking(int* field, std::vector<element>& tmp, std::vector<element>& min, std::vector<int>& squares) – функция, принимающая два вектора структур min и tmp, матрицу, содержащую поле(field), а также вектор squares, содержащий список необходимых для разбиения квадратов. Выполняет алгоритм описанный выше.

   Field – массив, содержащий текущее состояние поля квадрата.

   Tmp – вектор, в котором будут содержаться частные решения.

   Min – вектор, содержащий частное минимальное решение. После окончания работы алгоритма бэктрекинга в нем содержится минимальное разбиение квадрата.

3) bool check_variant(std::vector<int> squares, std::vector<element> tmp) – функция проверяет, присутствуют ли в минимальном разбиении все необходимые квадраты.

Squares – вектор, содержащий список всех необходимых квадратов в разбиении.

Tmp – вектор, в котором содержится частичное разбиение.

4) void print_particular_answer(std::vector<element>& tmp)

Функция вывода частичных полных разбиений.

Tmp – вектор, содержащий координаты и размеры всех квадратов в разбиении.

5) bool input(std::vector<int> squares)

Функция ввода всех необходимых данных для работы программы.

Squares – вектор, в который вводятся необходимые в разбиении квадраты

Тестирование.

1) Длина стороны квадрата – 7. Кол-во необходимых квадратов – 0.

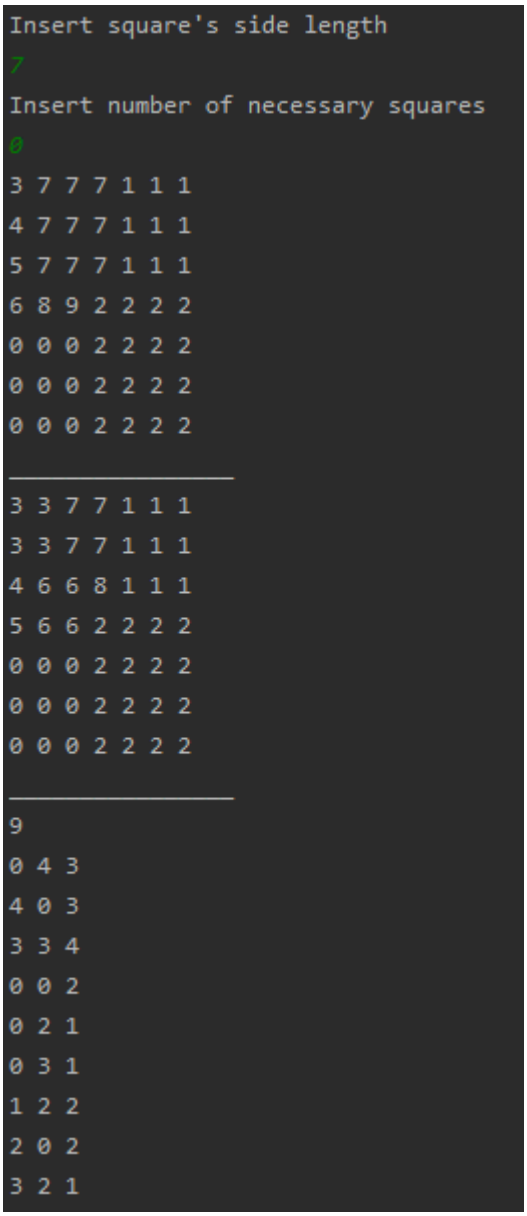

Рис. 1 – Результаты первого теста

2)Сторона квадрата – 3. Кол-во необходимых квадратов – 2, их длины – 1,

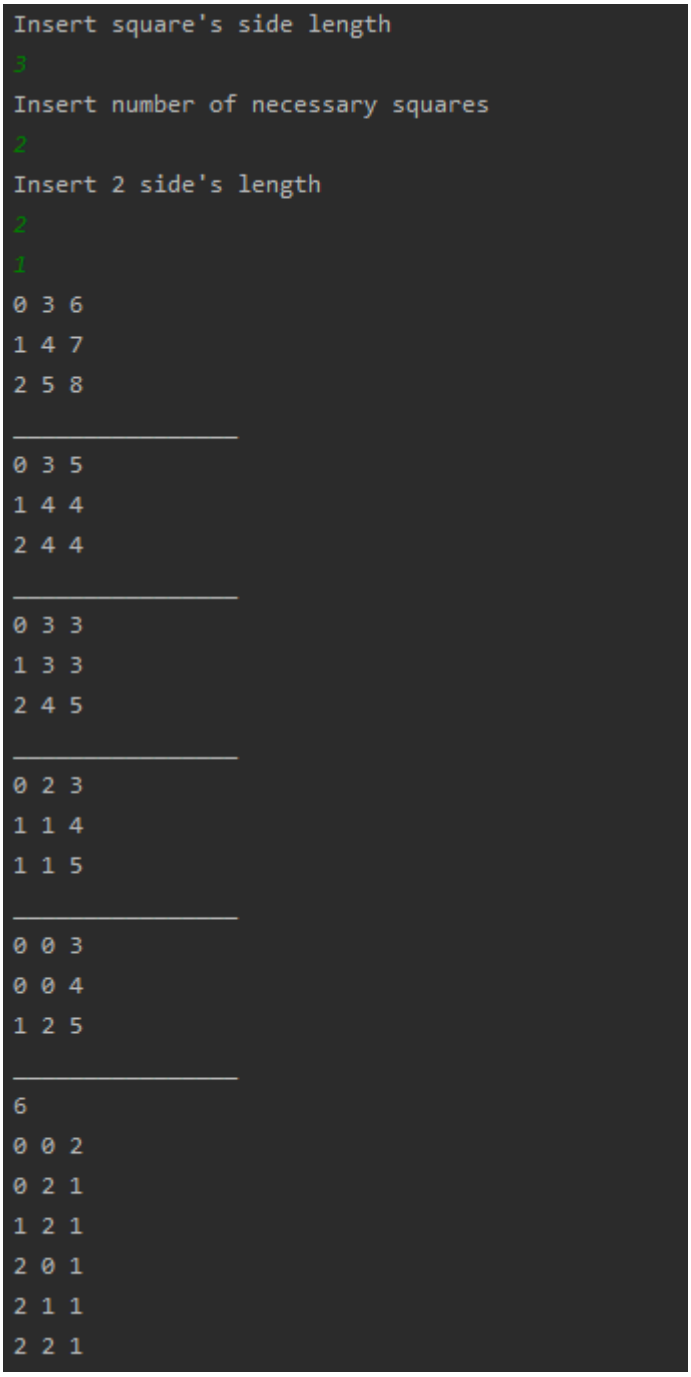

Рис. 2 – результаты 2-го теста

Оценка сложности алгоритма.

Затраты по памяти: Если размер входных данных n, то программа хранит n ^ 2 значений под исходный квадрат, и два вектора для хранения минимального и частного разбиения, которые в самом худшем случае имеют оценку n ^ 2, но в случае с отсутствием входного списка необходимых

квадратов, оба списка линейно зависят от n. Общая оценка затрат памяти : O(n^2).

Затраты по времени: использованный алгоритм поиска с возвратом является улучшенным обходом графа в. Значительная часть поддеревьев отбрасывается благодаря начальным оптимизациям, но тем не менее, алгоритм работает за экспоненциальное время. Была построена аппроксимирующая функция сложности, указанная на графике ниже. Оценка сложности по времени: O(2 ^ N)

Значения сложности для разных N в эксперименте:

(3;4)

(5;15)

(7;64)

(11;1400)

(13;3200)

(17;37000)

(19;101000)

(23;661000)

(29; 6800000)

(31;14652652)

(37;54200000)

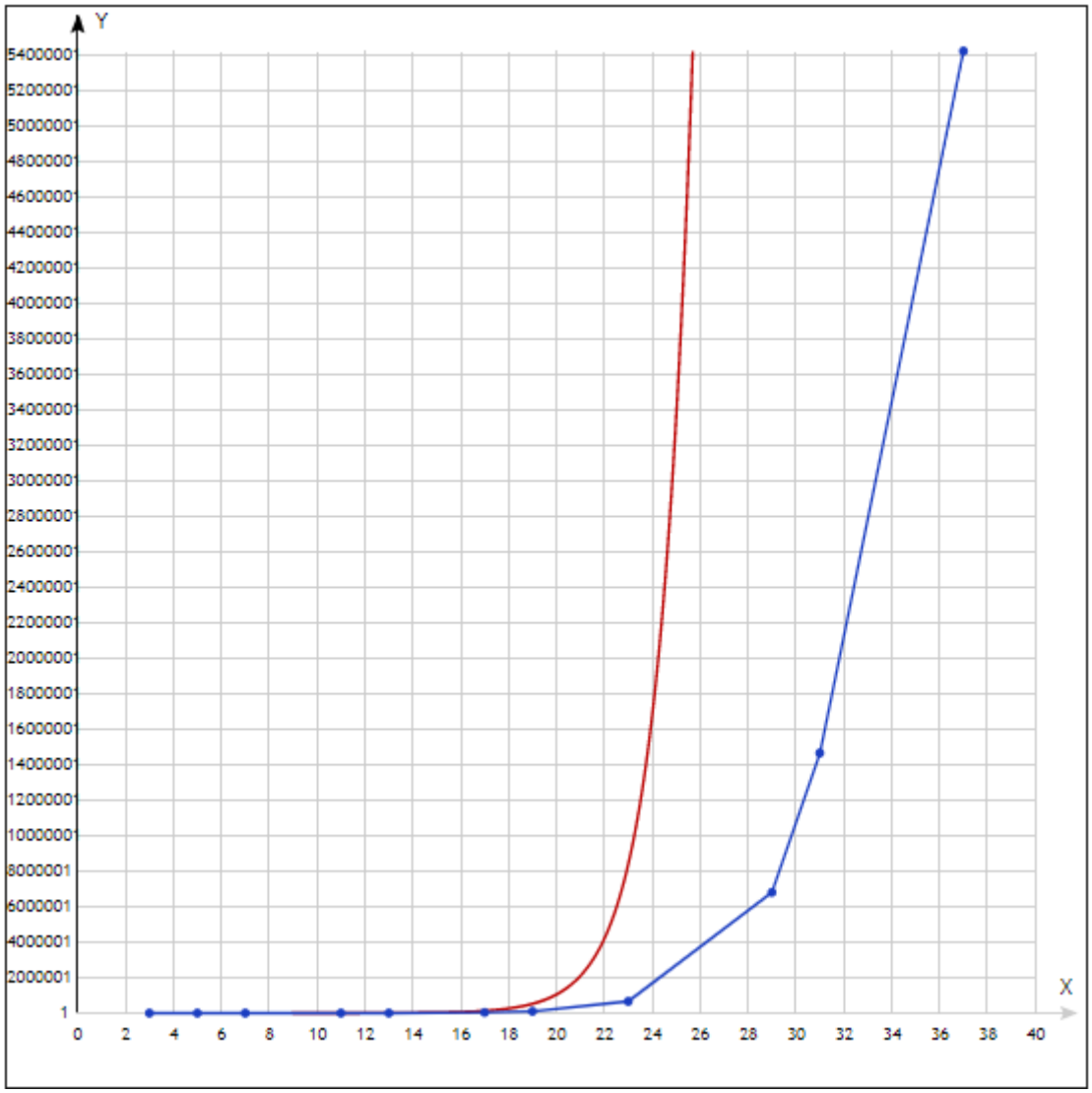

Рис. 3 – График аппроксимирующей функции и функции $f(x) = 2^n$

Вывод.

Был получен опыт в реализации алгоритма бэктрекинга, изучены различные оптимизации, сокращающие время работы алгоритма. Также в ходе работы была проанализирована сложность работы алгоритма и его затраты по памяти.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

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

int size = 1;

struct element {
   int x;
   int y;
   int len;
};

bool check_variant(std::vector<int> squares, std::vector<element> tmp) {

    if(squares.empty())
        return true;

    for(int i = 0; i < tmp.size(); ++i) {
        for(int j = 0; j < squares.size(); ++j) {
            if(tmp[i].len == squares[j]) {
                tmp.erase(tmp.begin() + i);
                i -= 1;
                squares.erase(squares.begin() + j);
                break;
            }
        }
    }
    return squares.empty();
}



void backtracking(int * field, std::vector<element>& tmp, std::vector<element>& min,
std::vector<int>& squares) {
   bool zero_found = false;
   element current;

   for (int i = 0; i < size; ++i) {
      for (int j = 0; j < size; ++j)
         if (field[i * size + j] == 0) {
                zero_found = true;
                current = {i, j, 1};
            break;
         }
      if (zero_found)
         break;
   }

   if (!zero_found) {
            if(check_variant(squares, tmp))
                min = tmp;
            return;
   }

   for (int tmp_len = 1; tmp_len < size; ++tmp_len) {
```

```
        if(squares.empty() && tmp.size() + 1 == min.size()) {
            break;
        }

       if (tmp_len > size - current.x || tmp_len > size - current.y)
          return;

       for (int i = current.x; i < current.x + tmp_len; ++i)
          for (int j = current.y; j < current.y + tmp_len; ++j)
             if (field[i * size + j] == 1)
                return;

       for (int i = current.x; i < current.x + tmp_len; ++i)
          for (int j = current.y; j < current.y + tmp_len; ++j)
                 field[i * size + j] = 1;

         current.len = tmp_len;
         tmp.push_back(current);
         backtracking(field, tmp, min, squares);
       for (int i = current.x; i < current.x + tmp_len; ++i)
          for (int j = current.y; j < current.y + tmp_len; ++j)
                 field[i * size + j] = 0;
       tmp.pop_back();
    }
}

int main() {

    int n;
    std::cout << "Insert square's side length\n";
    std::cin >> n;

    int quantity;
    std::cout << "Insert number of necessary squares\n";
    std::cin >> quantity;
    if(quantity < 0 || quantity > n*n) {
        std::cout << "Impossible number of squares\n";
        return 0;
    }

    std::vector<int> squares;
    int side_l;

    if(quantity != 0) {
        std::cout << "Insert " << quantity << " side's length\n";
        for (int i = 0; i < quantity; ++i) {
            std::cin >> side_l;
            if (side_l > 0 && side_l <= n - 1) {
                squares.push_back(side_l);
            }
        }
    }

    for (size = 2; size < n; ++size)
        if (n % size == 0)
            break;

    int multiplier = n / size;

    std::vector<element> tmp, min(squares.empty() ? size + 4 : size * size);
    if(squares.empty()) {
```

```
        tmp.push_back({ 0, size / 2 + size % 2, size / 2});
        tmp.push_back({ size / 2 + size % 2, 0, size / 2});
        tmp.push_back({ size / 2, size / 2, size / 2 + size % 2 });

        if (size % 2 == 1) {
            int* field = new int[(size / 2 + size % 2) * (size / 2 + size % 2)]();
            field[(size / 2 + size % 2) * (size / 2 + size % 2) - 1] = 1;
            size = size / 2 + size % 2;

            backtracking(field, tmp, min, squares);
        }
        else {
            tmp.push_back({ 0, 0, 1 });
            min = tmp;
        }
    }
    else {
        int *field = new int[size * size]();
        backtracking(field, tmp, min, squares);
        if (!check_variant(squares, min)) {
            std::cout << "Impossible variant!" << std::endl;
            return 0;
        }
    }

    std::cout << min.size() << std::endl;
    for (auto i:min) {
        std::cout << i.x * multiplier << " " << i.y * multiplier << " " << i.len * mul-
tiplier << std::endl;
    }

    return 0;
}
```