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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Построение и анализ алгоритмов»

Тема: Поиск с возвратом

Студент гр. 8382 _____ Черцницын П.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

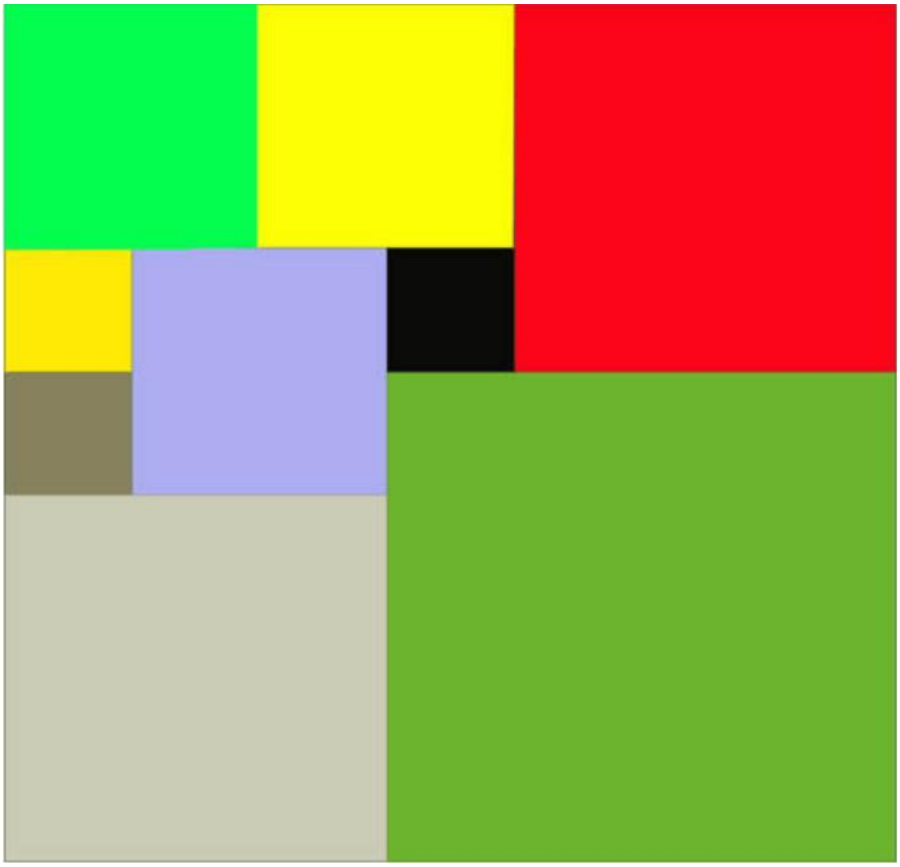
Цель работы.

Построить итеративный алгоритм поиска с возвратом для решения поставленной задачи. Определить сложность полученного алгоритма по операциям и по памяти.

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.



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 дополнительного задания.

1р. Рекурсивный бэктрекинг. Поиск решения за разумное время (меньше 2 минут) для $2 \le N \le 40$.

Описание алгоритма

Рекурсивный алгоритм поиска с возвратом реализован методом void rec_fill(vector<vector<int>>& sqr, int curSpace, int curSize, int countSqr,

vector<Square>& arrOfSqr). Описание всех структур приведено в разделе Описание структур. При нахождении свободной клетки с помощью перебора всех длин сторон в порядке уменьшения на поле добавляется квадрат максимально возможного размера, верхний левый угол которого расположен в этой клетке. Для хранения промежуточных решений создан vector<Square> arrOfSqr, который содержит координаты и размеры квадратов.

Если в ходе алгоритма число квадратов в текущем разбиении при непустом поле становится больше или равным числу квадратов в минимальном разбиении или если поле оказывается заполнено, происходит удаление с поля «хвоста» разбиения: поле освобождается от размещенных последними единичных квадратов, а квадрат, добавленный на поле перед ними или последним, если в конце разбиения нет единичных квадратов, заменяется квадратом, начинающимся в той же точке поля, со стороной на 1 меньше. При этом дальнейший обход матрицы начинается с правого верхнего угла последнего квадрата в новом разбиении, т. е. оттуда, откуда он продолжился бы, если бы квадрат был добавлен при обычном обходе поля.

Удаление последних квадратов разбиения также служит критерием выхода из рекурсии. Если при удалении «хвоста» был удален последний квадрат разбиения, т. е. больше нет нерассмотренных вариантов разбиения, работа алгоритма завершается.

В случае, когда после добавления нового квадрата поле оказывается заполнено, проверяется число квадратов в текущем разбиении. Если оно меньше числа квадратов в минимальном разбиении, минимальное разбиение заменяется текущим, а массив результата minArrOfSqr перезаписывается.

Использованные оптимизации.

Квадраты со сторонами, кратными 2, 3 и 5, обрабатываются отдельно, т. к. имеют особые разбиения, которые легко определить вручную (см. рисунки 1-3). Также реализованы разные начальные условия для применения алгоритма к квадратам.

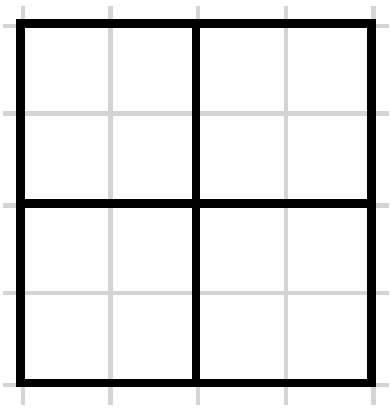


Рис.1 Для длин, кратных 2

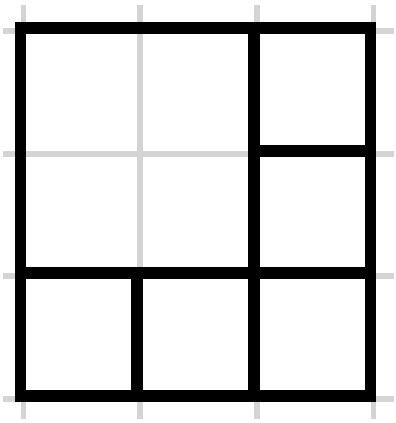


Рис.2 Для длин, кратных 3

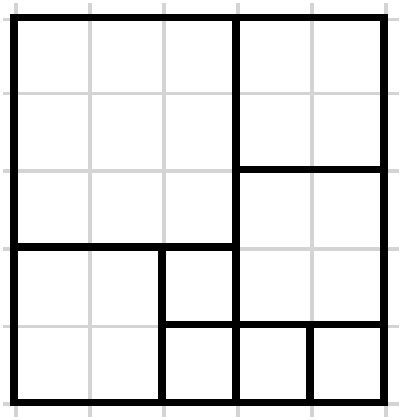


Рис.3 Для длин, кратных 5

Исследование.

N	RunTime
29	~1sec
31	2sec
27	16sec
41	85sec

В ходе исследования времени выполнения было выявлено, что при вводе в качестве размера «столешницы» простых чисел, время выполнения программы увеличивается экспоненциально с увеличением чисел. Например, при N = 29 время выполнения близко к 1 секунде, при N = 31 – 2 секунды, N=37 – 16 секунд, N=41 – 85 секунд (ограничение N <= 40 было сознательно проигнорировано для проведения исследования). При росте не простых чисел такого увеличения не наблюдается, так как для подобных чисел реализована оптимизация – сведение больших чисел к более простым случаям.

Описание структур данных.

Для хранения информации о квадрате была создана структура Square.

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

Три поля типа int – x, y и size, хранащие координаты по x, y и размер квадрата соответственно.

Описание функций и методов.

- void myInput(int &k, vector<vector<int>>& sqr_answ, vector<vector<int>>& sqr, vector<Square>& arrOfSqr, int& S) – функция ввода информации. Занимается тем, что считывает N и заполняет поля, которые от него зависит. Была предусмотрена вероятность ошибки пользователя. Именно поэтому считывается сначала переменная типа string, а после кастится к int с помошью функции cast_strToInt. В итоге мы получаем заполненные N, minNumOfSqr, sqr_answ, k, sqr, arrOfSqr, S).
- int cast_strToInt(string str) – принимает string на вход и кастит его к int. Если вылетает exception, выводит сообщение и возвращает код ошибки -1.
- void print_sqr(vector <vector <int>>& sqr, int k) – выводит sqr на экран.

- int can_insert(vector<vector<int>>& sqr, int x, int y, int size) – проверяет, можем ли мы вставить квадрат со стороной size на координаты x, y. return true, если можем.
- void insert(vector<vector<int>>& sqr, int x, int y, int size) – вставляет квадрат со стороной size на координаты x, y. Не проверяет возможность вставки.
- void remove_last(vector<vector<int>>& sqr, vector<Square>& sqrik_mas) – удаляет последний вставленный на столешницу квадрат.
- void rec_fill(vector<vector<int>>& sqr, int curSpace, int curSize, int countSqr, vector<Square>& arrOfSqr) – рекурсивная функция поиска минимального количества квадратов, необходимого для замощения столешницы.

Тестирование.

```
PS C:\Users\suicc\Desktop\PiAA\lab1> .\a.exe
Enter a number from 2 to 40: -1
Enter a number from 2 to 40: qwe
Bad input: std::invalid_argument thrown
Enter a number from 2 to 40: 1234123412342134
Integer overflow: std::out_of_range thrown
Enter a number from 2 to 40: 9
RunTime: 0 seconds
6
1 1 6
1 7 3
7 1 3
7 4 3
4 7 3
7 7 3
  6  6  6  6  6  6  3  3  3
  6  6  6  6  6  6  3  3  3
  6  6  6  6  6  6  3  3  3
  6  6  6  6  6  6  3  3  3
  6  6  6  6  6  6  3  3  3
  6  6  6  6  6  6  3  3  3
  3  3  3  3  3  3  3  3  3
  3  3  3  3  3  3  3  3  3
  3  3  3  3  3  3  3  3  3
```

```
PS C:\Users\suicc\Desktop\PiAA\lab1> .\a.exe
Enter a number from 2 to 40: 7
RunTime: 0 seconds
9
1 1 4
1 5 3
5 1 3
5 4 2
7 4 1
4 5 1
7 5 1
4 6 2
6 6 2
  4  4  4  4  3  3  3
  4  4  4  4  3  3  3
  4  4  4  4  3  3  3
  4  4  4  4  2  2  1
  3  3  3  1  2  2  1
  3  3  3  2  2  2  2
  3  3  3  2  2  2  2
```

```
PS C:\Users\suicc\Desktop\PiAA\lab1> .\a.exe
Enter a number from 2 to 40: 11
RunTime: 0 seconds
11
1 1 6
1 7 5
7 1 5
7 6 3
10 6 2
6 7 1
6 8 1
10 8 1
11 8 1
6 9 3
9 9 3
  6  6  6  6  6  6  5  5  5  5  5
  6  6  6  6  6  6  5  5  5  5  5
  6  6  6  6  6  6  5  5  5  5  5
  6  6  6  6  6  6  5  5  5  5  5
  6  6  6  6  6  6  5  5  5  5  5
  6  6  6  6  6  6  3  3  3  2  2
  5  5  5  5  5  1  3  3  3  2  2
  5  5  5  5  5  1  3  3  3  1  1
  5  5  5  5  5  3  3  3  3  3  3
  5  5  5  5  5  3  3  3  3  3  3
  5  5  5  5  5  3  3  3  3  3  3
```

```
PS C:\Users\suicc\Desktop\PiAA\lab1> .\a.exe
Enter a number from 2 to 40: 6
RunTime: 0 seconds
4
1 1 3
1 4 3
4 1 3
4 4 3
  3  3  3  3  3  3
  3  3  3  3  3  3
  3  3  3  3  3  3
  3  3  3  3  3  3
  3  3  3  3  3  3
  3  3  3  3  3  3
```

Сложность алгоритма

Без учета оптимизаций сложность алгоритма по операциям составила бы O(N*N), – если бы осуществлялся полный перебор вариантов без ограничения на минимальность решения. Ограничение на минимальность приводит к тому, что часть циклов прерывается до прохождения всех возможных значений, поэтому сложность алгоритма по операциям можно оценить величиной $O(e^N)$.

Сложность по памяти

В лучшем случае (N – не простое) – O(1), т.к. мы уже знаем как метить (см. раздел оптимизация).

В худшем случае (N –простое) – O(N^2), т.к. массивы, с которыми работает рекурсивная функция статические и глубина рекурсии слабо влияет на сложноть.

Выводы.

Была реализована программа, находящая разбиение квадратного поля минимально возможным количеством квадратов. Для решения поставленной задачи был построен и проанализирован рекурсивный алгоритм поиска с возвратом. Полученный алгоритм обладает экспоненциальной сложностью по операциям и квадратичной сложностью по памяти.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД.

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

using std::cout;
using std::cin;
using std::endl;
using std::vector;
using std::string;

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

vector <Square> minArrOfSqr;
int N;
int minNumOfSqr;

void insert(vector<vector<int>>& sqr, int x, int y, int size);

int cast_strToInt(string str)
/* -1 - код ошибки */
{
   int x;
   try
  {
              x = std::stoi(str);
  }
  catch (std::invalid_argument const &e)
  {
     std::cout << "Bad input: std::invalid_argument thrown" << '\n';
      return -1;
  }
  catch (std::out_of_range const &e)
  {
     std::cout << "Integer overflow: std::out_of_range thrown" << '\n';
      return -1;
  }
   return x;
}

void init(vector<vector<int>>& sqr)
{
   sqr.resize(N);
   for (int i = 0; i < N; i++)
   {
      sqr[i].resize(N);
      for (int j = 0; j < N; j++)
      {
         sqr[i][j] = 0;
      }
   }
}
```

```
void myInput(int &k, vector<vector<int>>& sqr_answ, vector<vector<int>>& sqr,
             vector<Square>& arrOfSqr, int& S)
{
   /* Читаем N_str, кастим к int и не завершаем цикл, пока не будет считано
    * корректное число. Попутно выкидываем exception'ы в консоль, чтобы пользователь
    * знал, где косячит */
   string tmp_str;
   do
   {
      cout << "Enter a number from 2 to 40: ";
      cin >> tmp_str;
      N = cast_strToInt(tmp_str);
   } while (N < 2 || N > 40);

   init(sqr_answ);
   init(sqr);

   if (N % 2 == 0)
   {
      k = N / 2;
      N = 2;
   }
   else if (N % 3 == 0)
   {
      k = N / 3;
      N = 3;
   }
   else if (N % 5 == 0)
   {
      k = N / 5;
      N = 5;
   }

   minNumOfSqr = 2 * N + 1;

   arrOfSqr.push_back({ 0, 0, (N + 1) / 2 });
   arrOfSqr.push_back({ 0, (N + 1) / 2, N / 2 });
   arrOfSqr.push_back({ (N + 1) / 2, 0, N / 2 });

   insert(sqr, 0, 0, (N + 1) / 2);
   insert(sqr, 0, (N + 1) / 2, N / 2);
   insert(sqr, (N + 1) / 2, 0, N / 2);
   S = N * N - ((N + 1) / 2) * ((N + 1) / 2) - 2 * (N / 2) * (N / 2);
}

void print_sqr(vector <vector <int>>& sqr, int k)
{
   for (int i = 0; i < N * k; i++)
   {
      for (int j = 0; j < N * k; j++)
      {
         cout.width(3);
         cout << sqr[i][j];
      }
      cout << "\n";
   }
}
int can_insert(vector<vector<int>>& sqr, int x, int y, int size)
{
```

```
    if ((x + size) > sqr.size() || (y + size) > sqr.size()) return 0;
    for (int i = y; i < y + size; i++)
    {
        for (int j = x; j < x + size; j++)
        {
            if (sqr[i][j] != 0) return 0;
        }
    }
    return 1;
}

void insert(vector<vector<int>>& sqr, int x, int y, int size)
{
    for (int i = y; i < y + size; i++)
    {
        for (int j = x; j < x + size; j++)
        {
            sqr[i][j] = size;
        }
    }

}
void remove_last(vector<vector<int>>& sqr, vector<Square>& sqrik_mas)
{
    Square tmp = *(sqrik_mas.rbegin());
    sqrik_mas.pop_back();

    for (int i = tmp.y; i < tmp.y + tmp.size; i++)
    {
        for (int j = tmp.x; j < tmp.x + tmp.size; j++)
        {
            sqr[i][j] = 0;
        }
    }
}


void rec_fill(vector<vector<int>>& sqr, int curSpace, int curSize, int countSqr, vector<Square>& arrOfSqr)
{
    if (countSqr == minNumOfSqr - 1 && curSpace > curSize* curSize) return;

    bool flag = false;
    for (int y = 0; y < N && !flag; y++)
    {
        for (int x = 0; x < N && !flag; x++)
        {
            if (sqr[y][x] == 0)
            {
                if (can_insert(sqr, x, y, curSize))
                {
                    /* Нашли, куда можно вставить квадрат и вставили */
                    insert(sqr, x, y, curSize);
                    flag = true;
                    curSpace -= curSize * curSize;
                    arrOfSqr.push_back({ x, y, curSize });
                }
                else
                {
                    return;
```

```
            }
        }
        else
        {
            x += sqr[y][x] - 1;
        }
    }
  }
  if (countSqr + 1 == minNumOfSqr)
  {
    /*Уже не лучший вариант расположения, так что назад */
    remove_last(sqr, arrOfSqr);
    return;
  }
  if (countSqr + 1 < minNumOfSqr && curSpace == 0)
  {
    minNumOfSqr = countSqr + 1;
    minArrOfSqr.assign(arrOfSqr.begin(), arrOfSqr.end());
    remove_last(sqr, arrOfSqr);
    return;
  }
  for (int i = (N + 1) / 2; i > 0; i--)
  {
    if (i * i <= curSpace)
      rec_fill(sqr, curSpace, i, countSqr + 1, arrOfSqr);
  }
  remove_last(sqr, arrOfSqr);
}

int main()
{
  int k = 1;
  vector<vector<int>> sqr_answ;
  vector<vector<int>> sqr;
  vector<Square> arrOfSqr;
  int S;

  /* myInput читает N и заполняет поля, от него зависящие.
    Также вставляются три первых квадрата */
  myInput(k, sqr_answ, sqr, arrOfSqr, S);

  time_t start = clock();
  for (int i = (N + 1) / 2; i > 0; i--) {
    rec_fill(sqr, S, i, arrOfSqr.size(), arrOfSqr);
  }

  time_t end = clock();
  cout << "RunTime: " << (end - start) / 1000 << " seconds\n";
  cout << minNumOfSqr << "\n";

  for (int i = 0; i < minArrOfSqr.size(); i++)
  {
    cout << minArrOfSqr[i].x * k + 1 << " " <<
      minArrOfSqr[i].y * k + 1 << " " << minArrOfSqr[i].size * k << endl;
    insert(sqr_answ, minArrOfSqr[i].x * k, minArrOfSqr[i].y * k, minArrOfSqr[i].size * k);
  }

  print_sqr(sqr_answ, k);
  return 0;
```

```
}
```