

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

ОТЧЕТ
по практической работе №1
по дисциплине «Построение и анализ алгоритмов»
Тема: Поиск с возвратом

Студент гр. 8382

Кобенко В.П.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

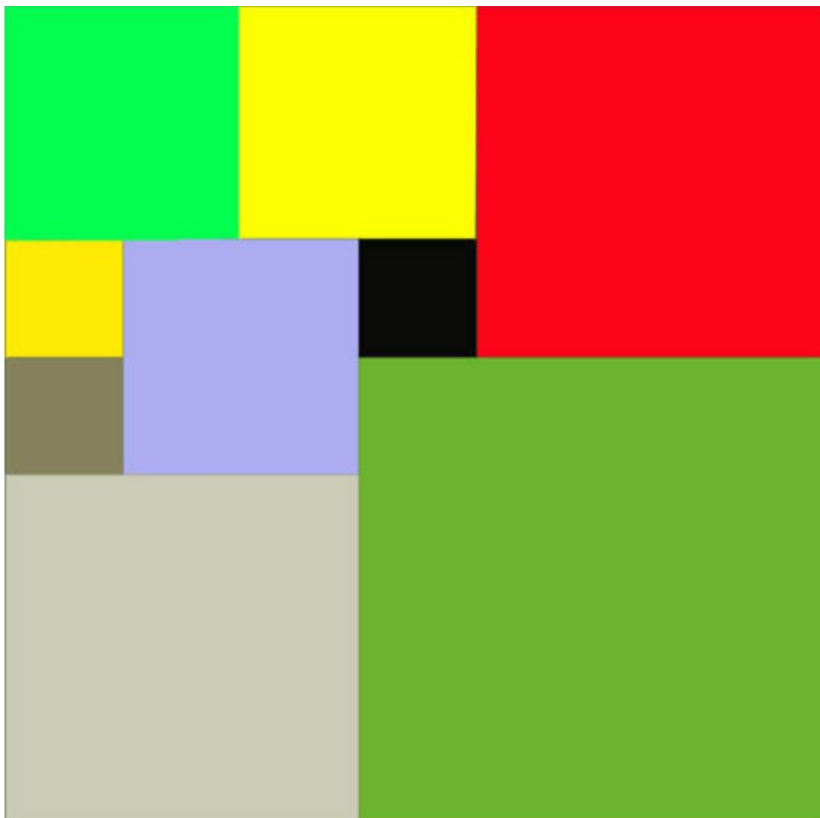
Цель работы.

Ознакомиться с алгоритмом перебора с возвратом и научиться применять его на практике. Написать программу реализовывающую поиск с возвратом.

Постановка задачи.

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

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

Индивидуальное задание.

Вариант 3р.

Рекурсивный бэктрекинг. Выполнение на Stepik всех трёх заданий в разделе 2.

Описание алгоритма.

На вход программе подается число от 2 до 40. Если число входит в

заданный промежуток, то программа начинает работу, иначе программа сообщит об ошибке.

Если рассмотреть разбиение всех квадратов длины от 2 до 40, то можно вывести следующее утверждение: минимальному разбиению (разбиению с наименьшим количеством квадратов) для непростых чисел N (Напр.: 6, 21, 33) будет соответствовать разбиение квадрата с длиной стороны равной наименьшему целочисленному делителю числа N не равному единице. Так, с помощью функции `mlt` находится этот самый делитель `mul`, если он есть и далее идет работа уже либо с квадратом длины `mul`, либо с прежним квадратом, если наименьший делитель N равен единице.

Сам квадрат в памяти программы представлен в виде двумерного массива `arr`. Кроме того, существует стандартный стек `sqr`s с типом элементов `Square`, где `Square` это структура, которая будет содержать информацию о квадрате, местоположение его левого верхнего угла относительно начала массива `arr` и длину самого квадрата.

Далее элементы `limit`, `&sqr`s, `arr`, N (`mul`), 0 из `main` подаются в рекурсивную функцию поиска минимального количества квадратов, где 0 из этого списка означает 0-й уровень рекурсии.

В самой рекурсии происходит следующее:

Проверка на достижение предельной глубины рекурсии. Если это предел, то возвращается -1. Осуществляется поиск пустой клетки в массиве `arr`. Если таковой не оказалось, то возвращается 0. Создаются дополнительные стеки `srqs_tmp` и `sqr`s_min. В первый будут записываться текущие квадраты, а во второй их минимальная последовательность. Далее происходит нахождение максимальной длины квадрата, который можно поместить, начиная с данной точки левого верхнего угла, с помощью функции `max_length`. Пробуются разные длины квадрата. Для этого

происходит заполнение квадрата функцией *fill_square*. Рекурсивное обращение из функции в эту же функцию, только с счетчиком рекурсии +1.

Возвращенное значение количества квадратов записывается в переменную *k*.

Происходит проверка значения *k* на минимальное и, если это так, запоминаем длину текущего квадрата, *k_min* и переписываем *sqrs_min* значениями из *sqrs_tmp*. Если же нет, то опустошаем *sqrs_tmp*. Далее очищаем квадрат. И переходим к следующей длине квадрата. Когда достигнем максимальной длины квадрата *max* копируем в *sqrs* значения из *sqrs_min*, записываем в *sqrs* текущий квадрат и возвращаем *k_min*.

После выхода из рекурсии выводится *k_min*, значения квадратов в нужном масштабе и освобождается память.

Сложность алгоритма $O(2^N)$. Объяснить это можно тем, что для каждого маленького квадрата существует 2 варианта размещения : либо он ставиться в большой квадрат, либо нет.

Описание структур.

Таблица 1 – Описание структур данных

Название структуры	Объект	Описание
<i>struct Square {}</i>	<i>int x;</i>	Координата квадрата x относительно начала массива.
	<i>int y;</i>	Координата квадрата y относительно начала массива.
	<i>int length;</i>	Длина квадрата.

Описание функций.

Таблица 2 – Описание функций

Сигнатура	Параметры	Описание
<i>int mlt(int N)</i>	N – длина квадрата	Возвращает наименьший делитель не равный нулю, если это непростое число. В случае с простым числом возвращает 1.
<i>void out (int **arr, int N)</i>	arr – массив клеток квадрата N – длина массива	Демонстрация массива.
<i>void fill_square (int **arr, int x, int y, int length_square)</i>	arr – массив клеток квадрата x – координата x квадрата y – координата y квадрата length_square – длина квадрата	Инициализирует квадрат в массиве в соответствии с параметрами x, y и length_square
<i>int empty_cell (int **arr, int &x, int &y, int N)</i>	arr – массив клеток квадрата x – координата x квадрата y – координата y квадрата N – размер массива	Находит пустую клетку в массиве arr и переводит значения x и y в соответствии с ее координатами. Возвращает 1 в случае, когда нет пустых клеток, и 0 в обратном.

<pre>void stack_copy (stack < Square> *sqrs, stack < Square> *sqrs_copy)</pre>	Sqrs – стек для копирования sqrs_copy – копируемый стек	Копирует стек.
<pre>void empty_stack (stack < Square> *sqrs)</pre>	sqrs – опустошаемый стек	Опустошает стек.
<pre>int min_sqrs(int limit , stack <Square> *sqrs, int **arr, int N, int count_rec)</pre>	limit – предельное значение счетчика рекурсии sqrs – стек для квадратов arr – массив клеток квадрата N – размер массива Count_rec – счетчик	Рекурсивная функция для перебора возможных значений расстановки квадратов.

Тестирование.

Таблица 3 – Тестирование

№ теста	Тест	Результат
1	2	4 1 1 1 1 2 1 2 1 1 2 2 1
2	6	4

		1 1 3 1 4 3 4 1 3 4 4 3
3	13	11 1 1 7 1 8 6 8 1 6 7 8 1 7 9 3 7 12 2 8 7 2 9 12 2 10 7 4 10 11 1 11 11 3
4	51	Error
5	37	37 15 1 1 19 1 20 18 20 1 18 19 20 11 9 21 3 19 24 7 19 31 7 20 19 2 22 19 5 26 24 2 26 26 12

		27 19 4 27 23 1 28 23 3 31 19 7
6	25	8 1 1 5 1 6 5 1 11 5 1 16 10 6 1 10 6 11 5 11 11 15 16 1 10
7	33	6 1 1 11 1 12 11 1 23 11 12 1 11 12 12 22 23 1 11

Вывод.

В ходе выполнения лабораторной работы был изучен и применен на практике алгоритм перебора с возвратом.

ПРИЛОЖЕНИЕ А

Исходный код программы

```
#include <iostream>
#include <cmath>
#include <stack>

// #define DEBUG

using namespace std;
// структура описывающая квадрат
struct Square{
int x;
int y;
int length;
};

// функция возвращает наименьший делитель не равный нулю, если число непростое,
// иначе возвращает 1
int mlt(int N){
if(N%2 == 0)
return 2;
if(N%3 == 0)
return 3;
if(N%5 == 0)
return 5;
return 1;
}

// функция для вывода массива
void out(int **arr,int N){
for(int m = 0; m < N; m++){
for(int l = 0; l < N; l++){
cout << arr[m][l]<< ' ';
cout << endl;
}
cout << endl;
}

// функция инициализации квадрата в массиве в соответствии с параметрами x, y и
length_square
void fill_square(int **arr, int x, int y, int length_square){
for(int i = 0; i < length_square; i++)
for(int j = 0; j < length_square; j++)
arr[x+i][y+j] = length_square;
}

// функция копирования стека
void stack_copy(stack <Square> * sqrs, stack <Square> * sqrs_copy){
while(!sqrs->empty()){
sqrs_copy->push(sqrs->top());
}
```

```
sqrs->pop();
```

```
}
```

```
}
```

```
// функция нахождения пустой клетки в массиве arr. Переводит значения x и y в соответствии с ее координатами.
```

```
// Возвращает 1, когда нет пустых клеток, и 0 в обратном.
```

```
int empty_cell(int **arr, int &x, int &y, int N){
```

```
for(x = 0; x < N; x++)
```

```
for(y = 0; y < N; y++)
```

```
if(arr[x][y] == 0)
```

```
return 0;
```

```
if(x == N)
```

```
return 1;
```

```
else
```

```
return 0;
```

```
}
```

```
// функция опустошения стека
```

```
void empty_stack(stack <Square> *sqrs){
```

```
while(!sqrs->empty())
```

```
sqrs->pop();
```

```
}
```

```
// функция нахождения максимальной длины квадрата, которого можно вписать в массив
```

```
int max_length(int **arr, int &x, int &y, int N){
```

```
int distance_to_the_border = (N-x > N-y) ? (N-y-!arr[0][0]) : (N - x - !arr[0][0]);
```

```
int length_square = 2;
```

```
for(; length_square <= distance_to_the_border; length_square++){
```

```
for(int i = 0; i < length_square; i++){
```

```
if(arr[x + length_square - 1][y + i] || arr[x + i][y + length_square - 1])
```

```
return length_square - 1;
```

```
}
```

```
}
```

```
return length_square - 1;
```

```
}
```

```
// рекурсивная функция для перебора возможных значений расстановки квадратов
```

```
int min_sqrs(int limit, stack <Square> *sqrs, int **arr, int N, int count_rec){
```

```
if(limit < count_rec)
```

```
return -1;
```

```
#ifdef DEBUG
```

```
out(arr, N);
```

```
#endif
```

```
int x, y;
```

```
if(empty_cell(arr, x, y, N)){
```

```
cout << "Rec number is " << count_rec << endl;
```

```
return 0;
```

```
}
```

```
stack <Square> sqrs_tmp;
```

```
stack <Square> sqrs_max;
```

```
int max = max_length(arr, x, y, N);
```

```
int length_square, k_min = limit + 1, k, need_length = 1;
```

```

for(length_square = 1; length_square <= max; length_square++){
    fill_square(arr, x, y, length_square);
    k = min_sqr(limit, &sqr_tmp, arr, N, count_rec+1)+1;
#ifdef DEBUG
    cout << "CURRENT K : " << k << ", K_MIN : " << k_min << endl;
    out(arr, N);
#endif
    if(k < k_min && k != 0){
        k_min = k;
        need_length = length_square;
        while(!sqr_max.empty()){
            sqr_max.pop();
        }
        stack_copy(&sqr_tmp, &sqr_max);
    }
    else
        empty_stack(&sqr_tmp);
    for(int i = 0; i < length_square; i++)
        for(int j = 0; j < length_square; j++)
            arr[x+i][y+j] = 0;
    }
    while(!sqr->empty())
        sqr->pop();
    stack_copy(&sqr_max, sqr);
    sqr->push({x, y, need_length});
    return k_min;
}

int main(){
    int N, min_k;
    cin >> N;
    if(!(N>=2 && N<=40)){ // проверка на корректность данных
        cout << "ERROR" << endl;
        return 0;
    }
    stack<Square> sqr;
    int mul = mlt(N);
    int limit = 6.37 * sqrt(sqrt((mul == 1) ? N : mul));
    int **arr;
    if(mul!=1){
        arr = new int *[mul];
        for(int i = 0; i < mul;i++)
            arr[i] = new int[mul]();
        min_k = min_sqr(limit, &sqr, arr, mul, 0);
    }
    else {
        arr = new int *[N];
        for(int i = 0; i < N; i++)
            arr[i] = new int[N]();
        int half = N - N/2;
        fill_square(arr, 0, 0, half);
        fill_square(arr, half, 0, half - 1);
        fill_square(arr, 0, half, half - 1);
    }
}

```

```

min_k = min_sqr(limit - 3, &sqr, arr, N, 0) + 3;
sqr.push({half, 0, half - 1});
sqr.push({0, half, half - 1});
sqr.push({0, 0, half});
}
cout << min_k << endl;
Square tmp;
int scale = (mul != 1) ? N/mul : 1;
while(!sqr.empty()){
tmp = sqr.top();
cout << tmp.x * scale + 1 << " " << tmp.y*scale + 1 << " " << tmp.length*scale << endl;
sqr.pop();
}
if(mul!=1) // отчистка массива
for(int i = 0; i < mul; i++)
delete [] arr[i];
else
for(int i = 0; i < N; i++)
delete [] arr[i];
delete [] arr;
return 0;
}

```