

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

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

Студент гр. 9382

Субботин М. О.

Преподаватель

Фирсов М. А.

Санкт-Петербург

2021

Цель работы.

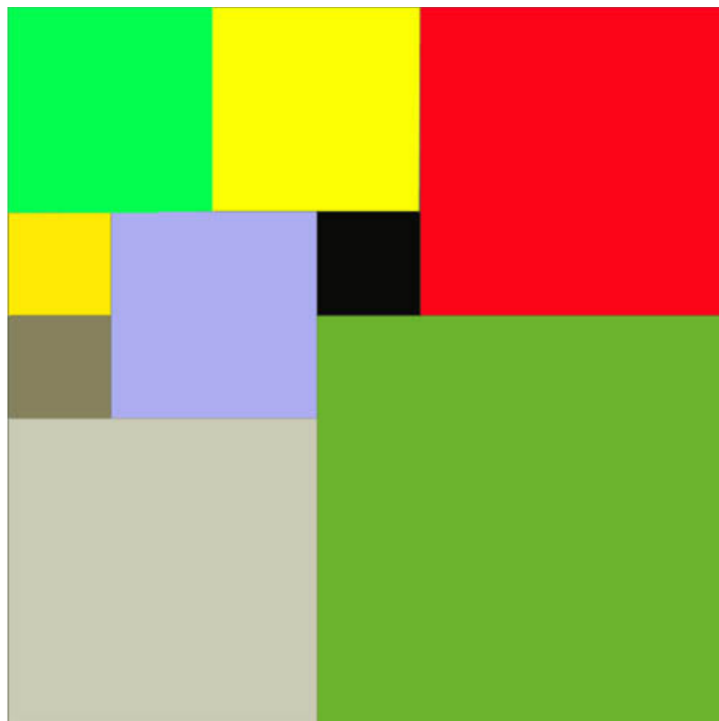
Познакомиться с одним из часто используемых на практике алгоритмом, поиске с возвратом. Получить навыки решения задач на этот алгоритм.

Задание.

Вар. 1и. Итеративный бэктрекинг. Выполнение на Stepik двух заданий в разделе 2.

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 $N \times N$. 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

3 4 1

Описание алгоритма.

Осуществляется проход по квадрату и ищется свободное место для вставки квадратика. Если такое место найдено, то ищется наибольший квадрат, который подойдет в это место. Если такого места не найдено, т.е. квадрат полностью заполнен, проверяется не является ли это разбиение минимальным, затем в обратном порядке вставления квадратиков стираются единичные квадратики (удаляются из стека) и следуемый за ними не единичный квадратик. Затем находится длина стороны последнего удаленного квадрата, наибольшая, но меньше предыдущей длины и в то же самое место вставляется квадрат снова. С помощью такого поиска, вставки и удаления и происходит проход по квадрату. Алгоритм закончит работу, как только сотрется один из “основных” квадратов.

Оптимизации алгоритма.

1. Квадрат с четной стороной очевидным образом можно разбить на 4 квадратика.
2. Если у квадрата составная длина стороны, то ее можно уменьшить и рассматривать квадрат меньшего размера. К примеру, квадрат с стороной 9 можно уменьшить до квадрата со стороной 3, а потом результаты просто до множить на 3.
3. В квадрат, у которого длина стороны – это простое число можно с самого начала точно вставить 3 “основных” квадрата. Один квадрат в углу

размера $(N+1)/2$, а два других по бокам от этого квадрата со сторонами $N - (N+1)/2$. Причем эти квадраты точно будут входить в решение. Поэтому и алгоритм можно заканчивать, как только удалим один из них.

4. Нет смысла дальше рассматривать случай, если уже количество квадратиков превышает наилучший результат.

Сложность алгоритма.

Поскольку хранится двумерный массив, иллюстрирующий информацию о положениях квадратиков, то сложность по памяти будет $O(N^2)$. В программе также присутствуют два вектора, но по памяти вместе они не будут превышать $2N^2$, т.к. в квадрат максимум можно вставить N^2 единичных квадратиков. Также программа итеративная, поэтому нет никаких фреймов занимающие дополнительную память, как это происходит в рекурсивных алгоритмах.

Достаточно проблематично определить точную оценку скорости алгоритма, учитывая оптимизацию, поэтому оценка будет относительно грубая. Всего в квадрате N^2 единичных квадратиков, и каждый из этих квадратиков по самым грубым подсчетам можно использовать как точку для $(N-1)$ разных квадратов.

Т.е. в итоге оценка получится $O(N^{(2(N-1))})$. Можно, конечно, сказать, что раз есть три первых квадрата, то оценка будет явно лучше, да, она будет лучше, но формула будет в крайне нечитаемом виде.

Описание функций и структур данных.

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

- структура квадратика, size – длина стороны, x и y – координаты в основном квадрате

int bigSquare[40][40]; - двумерный массив, который заполняется квадратиками

vector<Square> squareStack; - вектор, служащий стеком, и отвечающий за текущее состояние перебора (кол-во квадратики и сами квадратики)

vector<Square> resultStack; - вектор, в котором находится наилучшее состояние (с наименьшим количеством квадратики)

inline void insertSquare(Square mainSquare) – функция для вставки квадрата, когда известно что в это место квадрат точно вставится.

inline Square insertSquare(int bigSquareSize, Square square) – функция выбирает наибольший размер для вставки квадрата на место и вставляет его

inline Square emptyCell(int bigSquareSize) – функция ищет место, в которое можно вставить квадрат, если не находит это место, то возвращает квадрат с стороной -1.

inline void initialization(Square &square, int size) – функция заполняет основной квадрат тремя начальными квадратами.

inline Square BackWard(const int &size) – функция удаляет из стека последний квадратик и также стирает его с двумерного массива.

void doBackTracking(int size) – главная функция, реализует сам поиск.

Тестирование.

№	Входные данные	Выходные данные	Результат
1	3	6 1 1 2 1 3 1 3 1 1 3 2 1 2 3 1 3 3 1	Правильно
2	5	8 1 1 3 1 4 2	Правильно

		4 1 2 4 3 2 3 4 1 3 5 1 4 5 1 5 5 1	
3	9	6 1 1 6 1 7 3 7 1 3 7 4 3 4 7 3 7 7 3	Правильно
4	13	11 1 1 7 1 8 6 8 1 6 8 7 2 10 7 4 7 8 1 7 9 3 10 11 1 11 11 3 7 12 2 9 12 2	Правильно
5	11	11 1 1 6 1 7 5 7 1 5	Правильно

		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	29	14 1 1 15 1 16 14 16 1 14 16 15 2 18 15 5 23 15 7 15 16 1 15 17 3 15 20 3 18 20 3 21 20 2 21 22 1 22 22 8 15 23 7	Правильно
7	25	8 1 1 15 1 16 10 16 1 10 16 11 10 11 16 5	Правильно

		11 21 5 16 21 5 21 21 5	
8	23	13 1 1 12 1 13 11 13 1 11 13 12 2 15 12 5 20 12 4 12 13 1 12 14 3 20 16 1 21 16 3 12 17 7 19 17 2 19 19 5	Правильно

Выводы.

Был исследован часто используемый на практике алгоритм - поиск с возвратом. Также были получены навыки решения задач на этот алгоритм.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

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

using namespace std;

//структура для квадратиков
struct Square
{
    int size;
    int x;
    int y;
};

//глобальные переменные
int bigSquare[40][40];
vector<Square> squareStack;
vector<Square> resultStack;

//функция для вывода квадрата
void printSquare(int size){
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            printf("%2i ", bigSquare[i][j]);
        }
        printf("\n");
    }
}

//функция для вывода стека
void printStack(const vector<Square>& st){
    for(auto &square : st){
        cout << "{" << square.x << ", " << square.y << ", " << square.size <<
        "}, ";
    }
    cout << endl;
}
```

```

//функция для вставки основных квадратов
inline void insertSquare(Square mainSquare)
{
    for (int i = 0; i < mainSquare.size; ++i)
        for (int j = 0; j < mainSquare.size; ++j)
            bigSquare[mainSquare.y + i][mainSquare.x + j] =
squareStack.size()+1;
}

// функция для вставки квадратов, про которые мы точно не знаем какой их размер
подойдет
inline Square insertSquare(int bigSquareSize, Square square)
{
    square.size--;

    // находим максимальный размер, который подойдет квадрату
    int biggestSize = 1;
    while (biggestSize < square.size && (square.x + biggestSize) < bigSquareSize
&& (square.y + biggestSize) < bigSquareSize && !bigSquare[square.y][square.x +
biggestSize])
        biggestSize++;
    square.size = biggestSize;

    // вставляем квадрат
    for (int i = 0; i < square.size; ++i)
        for (int j = 0; j < square.size; ++j)
            bigSquare[square.y + i][square.x + j] = squareStack.size()+1;

    return square;
}

// функция по нахождению места под новый квадрат
inline Square emptyCell(int bigSquareSize)
{
    for (int i = 0; i < bigSquareSize; ++i)
        for (int j = 0; j < bigSquareSize; ++j)
            //существует не занятое место квадратом
            if (bigSquare[i][j] == 0)
                return {0, j, i};

    // квадрат заполнен
    return {-1, -1, -1};
}

```

```

}

//функция по добавлению основных квадратов
inline void initialization(Square &square, int size)
{
    square = {(size + 1) / 2, 0, 0};
    insertSquare(square);
    squareStack.push_back(square);

    square = {size - (size + 1) / 2, 0, (size + 1) / 2};
    insertSquare(square);
    squareStack.push_back(square);

    square = {size - (size + 1) / 2, (size + 1) / 2, 0};
    insertSquare(square);
    squareStack.push_back(square);

    square = emptyCell(size);
    square.size = size - 1;
}

//функция по удалению квадрата из стека и массива
inline Square BackWard(const int &size)
{
    // вытаскиваем удаляемый квадрат из стека
    Square lastSquare = squareStack.back();
    squareStack.pop_back();

    // очищаем квадрат
    for (int i = 0; i < lastSquare.size; i++)
        for (int j = 0; j < lastSquare.size; j++)
            bigSquare[lastSquare.y + i][lastSquare.x + j] = 0;

    return lastSquare;
}

// основная функция бектрекинга
void doBackTracking(int size)
{
    Square tempSquare;
    initialization(tempSquare, size);
}

```

```

    cout << "Запускаем поиск: " << endl;
    while (true)
    {
        // если удалим один из трех основных квадратов то можно заканчивать
перебор
        if (squareStack.size() == 2){
            cout << "Элементов в текущем стеке два, значит один из основных
квадратов пришлось удалить," <<
                "заканчиваем поиск." << endl;
            break;
        }
        cout << "В текущем стеке " << squareStack.size() << " элементов" <<
endl;

        bool full = false;

        //loop до тех пор, пока не заполнится квадрат
        while (!full)
        {
            // вставляем квадрат
            squareStack.push_back(insertSquare(size, tempSquare));
            cout << "Вставили квадрат размера " << squareStack.back().size << "
на x: " << squareStack.back().x << " y: " << squareStack.back().y << endl;
            printSquare(size);

            // если в текущем стеке уже столько же элементов, сколько и в
результатирующем, то дальше смысла двигаться нет
            if (!resultStack.empty() && squareStack.size() >=
resultStack.size())
            {
                cout << "В текущем стеке элементов больше чем в лучшем, дальше
идти смысла нет" << endl;
                break;
            }

            // ищем место под следующий квадрат
            tempSquare = emptyCell(size);
            if(tempSquare.size > -1) {
                cout << "Нашли свободное место x:" << tempSquare.x << " y:" <<
tempSquare.y << " под следующий квадрат"
                    << endl;

```

```

    }
    else {
        cout << "Квадрат полностью заполнен." << endl;
    }
    full = tempSquare.size == -1;
    tempSquare.size = size - (size + 1) / 2;
}

cout << "Минимальное количество квадратов: ";
(resultStack.empty() ? cout << "еще нет" : cout << resultStack.size())
<< endl;

cout << "Текущее количество квадратов: " << squareStack.size() << endl;
// если квадрат заполнен и текущих элементов меньше лучших, изменим это
if (full && squareStack.size() < resultStack.size() ||
resultStack.empty())
{
    cout << "Текущее количество квадратов оказалось меньше, получаем
новый минимум равный: " << squareStack.size() << endl;
    resultStack = squareStack;
}

/*
* здесь мы оказываемся в двух случаях:
* 1) когда заканчивается место в большом квадрате
* 2) когда путь длиннее чем текущий лучший результат
* в обоих случаях надо удалить последний квадрат
* т.к. единичные квадраты в конце расставляются единственным способом,
поэтому
* можно удалить их, и следующий за ними квадрат
*/
cout << "Будем удалять элементы из стека {x,y,size}: " << endl;
printStack(squareStack);
do
{
    tempSquare = BackWard(size);
} while (squareStack.size() > 2 && tempSquare.size == 1);
cout << "Стек после удаления: " << endl;
printStack(squareStack);
}
}

```

```

int main()
{
    int size;
    cout << "Введите сторону квадрата: ";
    cin >> size;

    //квадраты с четными сторонами всегда можно разделить на 4 равных
    if (size % 2 == 0)
    {
        cout << 4 << endl;
        cout << 1 << " " << 1 << " " << size / 2 << endl;
        cout << size / 2 + 1 << " " << 1 << " " << size / 2 << endl;
        cout << 1 << " " << size / 2 + 1 << " " << size / 2 << endl;
        cout << size / 2 + 1 << " " << size / 2 + 1 << " " << size / 2;
        return 0;
    }

    // находим, если есть делитель стороны квадрата
    // т.к. в квадрат с простыми сторонами точно можем вписать три квадрата
    int divider = 0;
    for (int i = 2; i <= size; i++)
    {
        if (size % i == 0)
        {
            divider = size / i;
            size = i;
            break;
        }
    }

    doBackTracking(size);

    cout << endl << resultStack.size() << endl;
    for (auto &square : resultStack)
        cout << square.x * divider + 1 << " " << square.y * divider + 1 << " "
<< square.size * divider << endl;
    return 0;
}

```