**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: Перебор с возвратом

|  |  |  |
| --- | --- | --- |
| Студент гр. 1304 |  | Поршнев Р.А. |
| Преподаватель |  | Шевелева А.М. |

Санкт-Петербург

2023

## Цель работы.

Изучить основные принципы решения задач с помощью 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. Он может получить её, собрав из уже имеющихся обрезков (квадратов).

Например, столешница 7 на 7 может быть построена из 9 обрезков ([см. рис. 1](#Рис1))

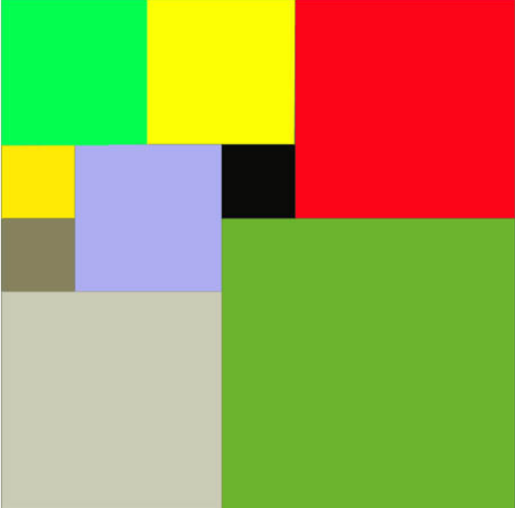
****

Рис. 1 – Оптимальное заполнение квадрата 7 на 7

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, целое число 2 <= N <= 30.

Выходные данные: K – минимальное число обрезков, из которых можно построить квадрат, и K строк с числами x, y, w, где x, y – координаты обрезка, w – длина обрезка.

## Основные теоретические положения.

Для успешного решения данной задачи, а под этим подразумевается получение правильного ответа на задачу за ограниченное время, использовался приём, который часто используется в решении задач перебора с возвратом, а именно, метод ветвей и границ. Идея данного метода заключается в том, чтобы отсекать заведомо неоптимальные решения. В данной работе использовались следующие оптимизации перебора с возвратом:

1) Если длина стороны квадрата *N* – составное число, то квадрат со стороной минимального простого делителя числа *N* имеет такое же количество минимальных квадратов, как и квадрат со стороной *N*. Более того, квадрат со стороной *N* является увеличенной копией квадрата со стороной минимального простого делителя числа *N*;

2) В левый верхний угол ставится квадрат со стороной , где *N* – сторона искомого квадрата, *q* – минимальный простой делитель числа *N*. Однако, если *N* – простое число, то *q* приравнивается к *N*;

3) Под квадрат со стороной ставится квадрат со стороной так, что он граничит с квадратом выше него и с левой границей столешницы. Правее квадрата со стороной ставится квадрат со стороной n таким образом, что правая сторона меньшего квадрата примыкает к правой границе столешницы, а левая – к большому квадрату ([см. рис. 2](#Рис2));

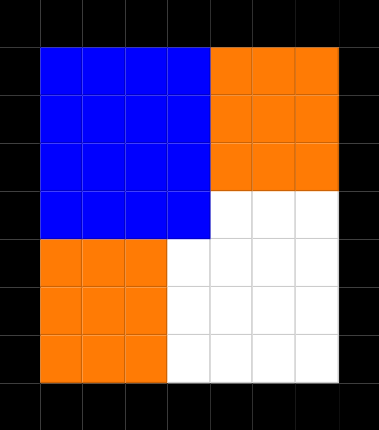


Рис. 2 – Оптимальная расстановка первых трёх квадратов в квадрате 7 на 7

4) Если количество квадратов, входящих в текущий набор для составления столешницы, превышает рекорд, то данная ветвь возможного решения заканчивается;

5) Если площадь текущего набора квадратов равна площади столешницы, то происходит проверка на улучшение рекорда. Вне зависимости от результата данная ветвь решения заканчивается;

6) Если количество квадратов со стороной 1 и 3 повторяется более пяти раз, то данная ветвь возможного решения заканчивается. Если количество квадратов со стороной 2 и 4 повторяется более четырёх раз, то данная ветвь решения тоже заканчивается. Данная оптимизация позволила решить задачу менее чем за 2 секунды для столешниц размером до 31.

## Выполнение работы.

В ходе решения задачи было реализовано два класса: *Factorization* и *Squares*.

В классе *Factorization* были реализован конструктор, который принимает в качестве аргумента размер столешницы и вносит его в поле данного класса, также происходит инициализация переменной *p* первым простым числом, а также инициализируется вектор типа *bool*, в котором в качестве аргумента будет выступать число от *0* до *n,* а значение данного элемента вектора – простое число или нет.

Также в классе были реализованы следующие методы:

* *void Sieve()* – решето Эратосфена. В данном методе происходит перебор простых чисел от *4* до *n*. Входные данные: *n* – сторона исходного квадрата;
* *int IsComposite()* – данный метод проверяет, является ли *n* простым число. Выходные данные: если число простое, то метод возвращает его наименьший простой делитель, а иначе – само число *n*;

В классе Squares реализован конструктор, в котором текущий рекорд *minNumbOfSquares* инициализируется достаточно большим числом (для надёжности стоит выбрать как минимум 1601). Также в данный конструктор передаётся размер столешницы и наименьший делитель её длины. Также в данном конструкторе инициализируется пустой квадрат, куда будут паковаться квадраты меньшего размера. Для удобства создаётся рамка для квадрата, состоящая из чисел -1.

Также в классе были реализованы следующие методы:

* *void Solution()* – данный класс запускает метод *InitiateThreeStartSquares()*, а также выводит количество квадратов и набор из этих квадратов. Каждый квадрат имеет свой номер от 1 по *minNumbOfSquares,* поэтому для вывода левого верхнего угла квадрата с номером numb достаточно пройти всё поле и вывести координаты клетки с номером numb в том случае, если она встречается впервые. Для подсчёта размера стороны квадрата c номером *numb* требуется посчитать количество клеток с номером *numb* и извлечь квадратный корень из данного значения;
* *void FindMinNumbOfSquares(squareInfo sqInf)* – данный метод отвечает за поиск решения на исходную задачу. В данном методе используются оптимизации, перечисленные в основных теоретических положениях. Идея расстановки текущего квадрата заключается в том, что, находясь в свободной клетке, нужно занять максимально свободную площадь квадратом, а затем запустить этот же метод из следующей потенциально свободной клетки, которая находится правее. Если клетка свободна, то снова нужно пытаться занять максимально свободную площадь квадратом, а если клетка занята другим квадратом, то продолжать поиски, двигаясь справа-налево сверху-вниз. В случае попадания на рамку, стоит перейти к рассмотрению потенциально свободных клеток на следующей строке. Причём при переходе стоит учитывать, что если значение строки меньше , то происходит переход на следующую строку и на столбец, равный , ведь клетки левее будут заведомо заняты самым большим квадратом. Если значение строки больше , то происходит переход на следующую строку и на столбец, равный , ведь клетки левее будут заведомо заняты квадратом со стороной . В случае возврата из последнего вызова данного метода выбирается квадрат со стороной меньше предыдущей на 1. Также в данном методе обновляется информации о количестве каждого квадрата со стороной от 1 до 5. Входные данные: текущее состояние структуры с данными заполняемого квадрата;
* *int MaxLengthOfSquare(std::vector<std::vector<int>> map, int x, int y)* – данный метод предназначен для поиска максимально свободной площади, куда можно вставить квадрат. Входные данные: текущее состояние заполненности квадрата, координаты *(x, y)* – точка старта поиска. Выходные данные: максимальный размер квадрата, который можно вставить;
* *std::vector<std::vector<int>> FillingOfSquare(std::vector<std::vector<int>> map, int x, int y, int lengthOfSide, int color, int direction)* – данный метод в зависимости от параметра *direction* может выполнять 2 функции: создание и обрезание текущего квадрата. Обрезание используется для того, чтобы заново не перерисовывать уже существующий квадрат. Входные данные: текущее состояние заполненности квадрата, координаты стартовой точка заливки или обрезания квадрата размером *lengthOfSide*, цвет закрашиваемого квадрата *color*, *direction* – переменная, отвечающая либо за заливку, либо за обрезание;
* *void InitiateThreeStartSquares()* – данный метод предназначен для создания трёх стартовых квадратов, обновления информации о количестве каждого квадрата со стороной от 1 до 5 и для вызова рекурсивного метода поиска решений.
* *squareInfo GetInitiatedBlackoutSquareInfo(squareInfo sqInf, int length)* – данный метод предназначен для отправки обновлённых данных в очередную ветвь рекурсии. Данные, используемые в рекурсивном методе, хранятся в структуре *squareInfo*:

*struct squareInfo {*

*std::vector<std::vector<int>> map;*

*int x;*

*int y;*

*int squareValue;*

*int numbOfSquares;*

*int countEachSquareType[5];*

*int prevLength;*

*};*

Входные данные: структура текущего состояния поля и сторона вставляемого квадрата. Выходные данные: обновлённая структура со вставленным квадратом.

Код программы находится в [приложении А](#ПриложениеА).

## Выводы.

В ходе выполнения данной лабораторной работы был изучен принцип решения задач с помощью перебора с возвратом и метода ветвей и границ.

Была написана программа на языке С++, решающая задачу составления квадрата определённого размера из минимального количества квадратов меньшего размера. Для решения данной задачи использовался оптимизированный перебор с возвратом. Использование оптимизаций позволило решить задачу для *n <= 30* менее чем за 2 секунды, а также она имеет экспоненциальную асимптотику.

**ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ**

Название файла: main.cpp

#include <iostream>

#include <vector>

#include <cmath>

/\*

\* Данная структура хранит текущую заполненность искомого квадрата в map,

\* координаты точки (x, y), начиная с которой будет производится заливка,

\* площадь текущего набора квадратов squareValue, размер текущего набора в numbOfSquares,

\* информация о количестве квадратов размером от 1 до 4 в countEachSquareType

\* размер предыдщуего поставленного квадрата в prevLength

\*/

struct squareInfo {

std::vector<std::vector<int>> map;

int x;

int y;

int squareValue;

int numbOfSquares;

int countEachSquareType[5];

int prevLength;

};

class Squares {

public:

/\*

\* Текущий рекорд количества квадратов хранится в minNumbOfSquares должен инициализироваться числом > 40^2 + 1

\* инициализация размера квадрата занчением n

\* инициализация "базового" квадрата размером baseN (т. е. максимальное уменьшение в масштабе квадрата размером n)

\* инициализация пустого квадрата bestMap с рамкой из -1, где будет храниться лучшая комбинация меньших квадратов

\*/

Squares(int n, int baseN) {

this->minNumbOfSquares = infinity;

this->n = n;

this->baseN = baseN;

for (int i = 0; i < n + 2; i++) {

std::vector<int> row;

for (int j = 0; j < n + 2; j++) {

if ((j == 0) or (j == n + 1)) {

row.push\_back(-1); // по бокам рамка из -1

}

else {

row.push\_back(0);

}

}

bestMap.push\_back(row);

}

for (int i = 1; i < n + 1; i++) { // Сверху рамка из -1

bestMap[0][i] = -1;

}

for (int i = 1; i < n + 1; i++) {

bestMap[n + 1][i] = -1; // Снизу рамка из -1

}

}

/\*Запуск заполнения искомого квадрата и вызов метода для вывода полученного ответа\*/

void Solution() {

InitiateThreeStartSquares(); // заполнение 3-ёх квадратов

PrintBestSet();

}

private:

/\*Вывод полученного ответа на исходную задачу\*/

void PrintBestSet() {

std::cout << minNumbOfSquares << std::endl;

bool\* visited = new bool[minNumbOfSquares + 1];

for (int i = 0; i <= minNumbOfSquares; i++)

visited[i] = false;

for (int numb = 1; numb <= minNumbOfSquares; numb++) { // вывод набора квадратов

int currentSquare = 0;

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= n; j++) {

if (bestMap[i][j] == numb) {

if (!visited[bestMap[i][j]]) {

visited[bestMap[i][j]] = true;

std::cout << i << " " << j << " ";

}

currentSquare++;

}

}

}

std::cout << (int)sqrt(currentSquare) << "\n";

}

}

/\*

Входные данные: структура с необходимыми данными состояния поля

Данный метод представляет собой рекурсивный перебор с оптимизациями

\*/

void FindMinNumbOfSquares(squareInfo sqInf) {

if (sqInf.prevLength < 5) { // отсечение для квадратов размером до 5 на 5

if (sqInf.prevLength % 2 == 1) { // для 1 и 3

if (sqInf.countEachSquareType[sqInf.prevLength] > 5) {

return;

}

}

else {

if (sqInf.countEachSquareType[sqInf.prevLength] > 4) { // для 2 и 4

return;

}

}

}

if (sqInf.numbOfSquares >= minNumbOfSquares) // отсечение переполнения текущего минимума

return;

if (sqInf.squareValue == n \* n) { // площадь текущего набора квадратов равна площади искомого квадрата

if (sqInf.numbOfSquares < minNumbOfSquares) {

minNumbOfSquares = sqInf.numbOfSquares; // запись нового рекорда

bestMap = sqInf.map;

}

return;

}

while (sqInf.x <= n) {

while (sqInf.y <= n) {

if (sqInf.map[sqInf.x][sqInf.y] == 0) { // клетка свободна

int maxLengthOfCurrentSquare = MaxLengthOfSquare(sqInf.map, sqInf.x, sqInf.y);

// поиск максимального квадрата из данной клетки

for (int length = maxLengthOfCurrentSquare; length > 0; length--) {

// перебор всех возможных квадратов из данной клетки

if (length == maxLengthOfCurrentSquare) {

sqInf.map = FillingOfSquare(sqInf.map, sqInf.x, sqInf.y,

length, sqInf.numbOfSquares + 1, filling); // заполнение

}

else {

sqInf.map = FillingOfSquare(sqInf.map, sqInf.x + length, sqInf.y + length,

length + 1, sqInf.numbOfSquares + 1, trimming);

// урезание на 1

}

FindMinNumbOfSquares(GetInitiatedBlackoutSquareInfo(sqInf, length));

// запуск рекурсии для следующей \*возможно свбодной клетки

// \* - рекурсия может попасть сразу на правую границу столешницы

}

return;

}

sqInf.y++;

}

sqInf.x++;

if (sqInf.x <= bigSquareLengthOfSide) {

sqInf.y = bigSquareLengthOfSide + 1; // чтобы не попадать на заведомо заполненный большой квадрат

}

else {

sqInf.y = n - bigSquareLengthOfSide + 1; // чтобы не попадать на заведомо на заполненный квадрат,

// который меньше самого большого квадрата на минимальное количество клеток

}

}

}

/\*

\* Вход: текущее состояние искомого квадрата map, координаты стартовой точки заливки (x, y)

\* Выход: максимальный размер квадрата, который можно вставить

\* Данный метод предназначен для поиска максимального квадрата, который можно вставить таким образом,

\* чтобы его левый верхний угол находился в точке с координатами (x, y)

\*/

int MaxLengthOfSquare(std::vector<std::vector<int>> map, int x, int y) {

int maxLengthOfSide = 0;

while ((map[x][y + maxLengthOfSide] == 0) && (map[x + maxLengthOfSide][y] == 0)) {

maxLengthOfSide++;

}

return maxLengthOfSide;

}

/\*

\* Вход: состояние заполненности искомого квадрата map,

\* координаты точки (x, y), начиная с которой будет производится заливка или урезание существующего квадрата,

\* длина стороны заполняемого или урезаемого квадрата lengthOfSide,

\* заливка или урезание - переменная direction

\* Выход: новое состояние заполненности искомого квадрата

\* Данный метод либо закрашивает новую область в виде квадрата, либо урезает уже существующую

\*/

std::vector<std::vector<int>> FillingOfSquare(std::vector<std::vector<int>> map,

int x, int y, int lengthOfSide, int color, int direction) {

switch (direction)

{

case filling: // заполнение

for (int i = x; i < x + lengthOfSide; i++)

for (int j = y; j < y + lengthOfSide; j++)

map[i][j] = color;

break;

case trimming: // урезание уже нарисованной области на 1

for (int i = x; i > x - lengthOfSide; i--)

map[i][y] = 0;

for (int j = y; j > y - lengthOfSide; j--)

map[x][j] = 0;

break;

}

return map;

}

/\*

\* Данный метод оптимально устанавливает первые 3 квадрата и запускает рекурсивный перебор

\*/

void InitiateThreeStartSquares() {

bigSquareLengthOfSide = (n / baseN) \* ((baseN + 1) / 2);

bestMap = FillingOfSquare(bestMap, 1, 1, bigSquareLengthOfSide, 1, filling); // нарисовать самый большой квадрат

bestMap = FillingOfSquare(bestMap, 1, 1 + bigSquareLengthOfSide, n - bigSquareLengthOfSide, 2, filling); // нарисовать

bestMap = FillingOfSquare(bestMap, 1 + bigSquareLengthOfSide, 1, n - bigSquareLengthOfSide, 3, filling); // 2 квадрата

squareInfo sqInf;

sqInf.map = bestMap;

sqInf.x = n - bigSquareLengthOfSide + 1;

sqInf.y = bigSquareLengthOfSide + 1;

sqInf.squareValue = (int)pow(bigSquareLengthOfSide, 2) + 2 \* (int)pow(n - bigSquareLengthOfSide, 2);

sqInf.numbOfSquares = 3;

for (int i = 0; i < 5; i++) {

sqInf.countEachSquareType[i] = 0;

};

if (bigSquareLengthOfSide < 5) {

sqInf.countEachSquareType[bigSquareLengthOfSide] = 1;

}

if (n - bigSquareLengthOfSide < 5) {

sqInf.countEachSquareType[n - bigSquareLengthOfSide] = 2;

}

sqInf.prevLength = n - bigSquareLengthOfSide;

FindMinNumbOfSquares(sqInf);

// вызвать рисование остальных квадратов

}

/\*

\* Вход: структура текущего состояния поля, сторона вставляемого квадрата

\* Выход: обновлённая структура со вставленным квадратом

\* Данный метод обновляет текущее состояние поля после вставки в него квадрата размером length

\*/

squareInfo GetInitiatedBlackoutSquareInfo(squareInfo sqInf, int length) {

squareInfo sqInfBlack;

sqInfBlack.map = sqInf.map;

sqInfBlack.x = sqInf.x;

sqInfBlack.y = sqInf.y + length;

sqInfBlack.squareValue = sqInf.squareValue + (int)pow(length, 2);

sqInfBlack.numbOfSquares = sqInf.numbOfSquares + 1;

sqInfBlack.prevLength = length;

for (int i = 0; i < 5; i++)

sqInfBlack.countEachSquareType[i] = sqInf.countEachSquareType[i];

if (length < 5) {

sqInfBlack.countEachSquareType[length] = sqInfBlack.countEachSquareType[length] + 1;

}

return sqInfBlack;

}

int bigSquareLengthOfSide;

std::vector<std::vector<int>> bestMap;

int n;

int baseN;

int minNumbOfSquares;

enum directions { filling = 1, trimming };

int infinity = 10000;

};

class Factorization {

public:

/\*

\* Инициализация числа n, до которого будет производится поиск простых чисел

\* Инициализация первого простого числа p значением 2

\* Инициализация вектора чисел primeNumbers значениями true для последующего вычёркивания составных чисел

\*/

Factorization(int n) {

this->n = n;

this->p = 2;

for (int i = 0; i <= n; i++) {

primeNumbers.push\_back(true);

}

}

/\*

\* Данный метод представляет собой реализацию решета Эратосфена

\*/

void Sieve() {

while (p < n) {

for (int i = 2 \* p; i < n; i++) {

if (i % p == 0) {

primeNumbers[i] = false;

}

}

if (p + 1 < n) {

for (int i = p + 1; i <= n; i++) {

if (primeNumbers[i] == true) {

p = i;

break;

}

}

}

else {

p++;

}

}

}

/\*

\* Выход: если n - составное число, то возвращается его минимальный простой делитель, иначе - само число n

\*/

int IsComposite() { // cоставное ли число длина стороны n?

for (int i = 2; i < n; i++) {

if ((n % i == 0) && (primeNumbers[i] = true)) {

return i; // n - cоставное

}

}

return n; // n - простое

}

private:

int n;

int p;

std::vector<bool> primeNumbers;

};

int main() {

int n;

std::cin >> n;

Factorization frz(n);

frz.Sieve();

Squares sqrs(n, frz.IsComposite());

sqrs.Solution();

return 0;

}