**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: Поиск с возвратом**

Студентка гр. 7381 Машина Ю.Д.

Преподаватель Фирсов М. А.

Санкт-Петербург

2019

**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≤N≤40)***.

**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 ≤ x,y ≤ 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

**Индивидуализация.**

Вар. 5и. Итеративный 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.**

Будем заполнять столешницу квадратами по следующему алгоритму:

Шаг 1. Найти свободную клетку сетки и перейти на шаг 2. Если свободную клетку найти не удалось, перейти на шаг 4.

Шаг 2. Если текущее решение содержит число поставленных квадратов большее или равное, чем число, на единицу меньшее ранее найденного решения, перейти на шаг 6. Иначе найти максимально возможный размер квадрата, который можно поставить на эту клетку и перейти на шаг 3.

Шаг 3. Поставить этот квадрат, запомнить его как предыдущий поставленный, и перейти на шаг 1.

Шаг 4. Сохранить данное решение и число квадратов в данном решении. Перейти на шаг 5.

Шаг 5. Снять поставленный квадрат, уменьшить размер квадрата на единицу. Если его размер больше 0, перейти на шаг 3. Иначе перейти на шаг 6.

Шаг 6. Если до этого шага не было поставлено ни одного квадрата, то завершить работу алгоритма. Иначе, рассматривая предыдущий поставленный квадрат перейти на шаг 5.

Возврат в таком случае осуществляется за счет того, что мы снимаем квадраты со столешницы. Перебор значительно сокращается благодаря тому, что мы не перебираем заранее плохие варианты – когда было уже поставлено меньшее чем на единицу количество квадратов чем в ранее найденном заполнении.

Представим нашу столешницу в виде массива наполненности колонок двумерной сетки размера N. Тогда нахождение свободного места и максимального размера квадрата, который можно поставить на это место, можно выполнить за N операций. К тому-же, случай полной наполненности столешницы можно обработать за 1 операцию, введя переменную наполненности площади столешницы.

Дополнительное ограничение перебора: если размер столешницы N можно представить в виде N = p \* koef, где p – минимальный простой делитель числа N, то имеет смысл обработать столешницу размера pxp, поскольку минимальное заполнение такой столешницы можно натянуть на столешницу NxN, и оно, очевидно, будет минимальным и для нее.

Экстра ограничение перебора: эмпирическим путем было показано, что существует минимальное заполнение столешницы, содержащее квадрат в углу размером (N+1) / 2, и два квадрата размером (N-1) / 2 в прилежащих углах.

**Оценка сложности алгоритма по операциям.**

При выбранных ограничениях, перебор будет совершаться в квадрате размером , с вырезанным уголком размера 1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | … | … | … | a |
| 2 |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |
| … |  |  |  |  |  |  |  |
| … |  |  |  |  |  |  |  |
| … |  |  |  |  |  |  |  |
| a |  |  |  |  |  |  |  |

При этом перебор будет осуществляться преимущественно для наибольших квадратов, которые можно поместить в данную точку, т.е. не будут перебираться варианты вроде – один квадрат размером a – k, k > 1, в точке (2,1), а все остальное заполняется единичными квадратами, поскольку по крайней мере найдется решение для 2a - 1 квадратов.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | ... | ... | ... | a |
| 2 |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |
| ... |  |  |  |  |  |  |  |
| ... |  |  |  |  |  |  |  |
| ... |  |  |  |  |  |  |  |
| a |  |  |  |  |  |  |  |

Соответственно, алгоритм в худшем случае начнет перебирать все возможные подстановки 2a - 2 квадратов для всех возможных допустимых размеров этих квадратов. Всего таких перестановок a\*(2a - 2)!. Таким образом приходим к оценке алгоритма О(N!). Вставка и удаление квадрата выполняются за O(N), что не добавляет сложности к оценке.

**Оценка сложности алгоритма по памяти.**

В алгоритме используется массив наполненности колонок и три ограниченных списка на базе массива. Все данные структуры зависят от N линейно, т.е. сложность по памяти – O(N+k), где k - кол-во квадратов, которые обязаны участвовать в разложении.

**Описание функций и структур данных**.

|  |  |  |
| --- | --- | --- |
| **Название** | **Поле** | **Описание** |
| template <class Base, size\_t lsize> class Static\_list | Base\* mem | Список - массив, хранящий максимум lsize элементов. |
| void push\_back() | Добавляет элемент в список. |
| Base& pop\_back() | Возвращает значение элемента, находящегося первым в списке, и сдвигает указатель на последний элемент влево. |
| bool empty() | Возвращает true, если список пуст. |
| void copy(const Static\_list& other) | Создает копию списка other. |
| size\_t last | Указатель на последний элемент списка. |
| struct SolutionNode | int x, int y, int w | Тройка чисел для описания положения и размера квадрата на столешнице. |
| template <size\_t N> class Solver | void get\_solution(size\_t size) | Стартовая точка работы алгоритма. Инициализация необходимых переменных и запуск алгоритма на столешнице необходимого размера. |
| void print\_solution() | Выводит на экран найденное решение. |
| int getNumberOfOperations() | Возвращает количество операций, потребовавшееся для нахождения решения. |
| int get\_p(int N) | Возвращает наименьший простой делитель числа N. |
| void set(int column, int w) | Кладет в столешницу квадрат размером w на колонку column. |
| void unset(int column, int w) | Убирает из столешницы квадрат размером w на колонке column. |
| int get\_column() | Возвращает номер колонки, на которой может быть поставлен квадрат. Если таковой не имеется, возвращает -1. |
| int get\_max\_w(int column) | Возвращает максимальный размер квадрата, который можно поставить в столешницу на колонку column. |
| void getSubDiv() | Функция, исполняющая алгоритм поиска наименьшего заполнения столешницы. |
| int columns[N] | Массив заполнения колонок сетки на которой располагается столешница. |

**Тестирование.**

Результатом работы программы является минимальное число квадратов, необходимое для заполнения столешницы, учитывая необходимо участвующие в разложении квадраты.

|  |  |  |
| --- | --- | --- |
| № теста | Входные данные: | Выходные данные: |
| 1 | 2  1  1 | 4  2 2 1  1 2 1  2 1 1  1 1 1 |
| 2 | 5  2  3  2 | 8  5 5 1  4 5 1  3 5 1  3 4 1  1 4 2  4 3 2  4 1 2  1 1 3 |
| 3 | 7  4  5  2  2  2 | 10  7 7 1  6 7 1  5 7 1  5 6 1  3 6 2  1 6 2  6 5 2  6 3 2  6 1 2  1 1 5 |
| 4 | 15  4  10  5  5  5 | 6  11 11 5  6 11 5  1 11 5  11 6 5  11 1 5  1 1 10 |
| 5 | 18  3  15  3  3 | 12  16 16 3  13 16 3  10 16 3  7 16 3  4 16 3  1 16 3  16 13 3  16 10 3  16 7 3  16 4 3  16 1 3  1 1 15 |
| 6 | 3  2  2  2 | No solutions found. |
| 7 | 4  1  4 | No solutions found. |

**Описание теста 1.**

Ввиду слишком крупных объемов поясняющего текста в нетривиальных примерах, рассмотрим тривиальный. В столешницу 2x2 надо положить 2 квадрата со сторонами 1 и 1.

Step 1: finding the first free cell of the grid. A free cell has been found. Going to step 2

Step 2: getting the maximal size (width) of a square. It's 1. Going to step 3

Step 3: placing the square, remembering it as the previously placed. Going to step 1

Step 1: finding the first free cell of the grid. A free cell has been found. Going to step 2

Step 2: getting the maximal size (width) of a square. It's 1. Going to step 3

Step 3: placing the square, remembering it as the previously placed. Going to step 1

Step 1: finding the first free cell of the grid. A free cell has been found. Going to step 2

Step 2: getting the maximal size (width) of a square. It's 1. Going to step 3

Step 3: placing the square, remembering it as the previously placed. Going to step 1

Step 1: finding the first free cell of the grid. A free cell has been found. Going to step 2

Step 2: getting the maximal size (width) of a square. It's 1. Going to step 3

Step 3: placing the square, remembering it as the previously placed. Going to step 1

Step 1: finding the first free cell of the grid. A free cell hasn't been found. Going to step 4

Step 4: saving solution and ammount of squares in it

Step 6: Some squares have been placed already. Reviewing the previously placed square. Going to step 5

Step 5: removing the square that has just been placed. Decreasing its size (width) by 1. New size (width) equals 0. Going to step 6

Step 6: Some squares have been placed already. Reviewing the previously placed square. Going to step 5

Step 5: removing the square that has just been placed. Decreasing its size (width) by 1. New size (width) equals 0. Going to step 6

Step 6: Some squares have been placed already. Reviewing the previously placed square. Going to step 5

Step 5: removing the square that has just been placed. Decreasing its size (width) by 1. New size (width) equals 0. Going to step 6

Step 6: Some squares have been placed already. Reviewing the previously placed square. Going to step 5

Step 5: removing the square that has just been placed. Decreasing its size (width) by 1. New size (width) equals 0. Going to step 6

Step 6: No squares have been placed still. Algorithm is completed

4

2 2 1

1 2 1

2 1 1

1 1 1

**Приложение 1. Код программы без промежуточного вывода.**

#include <math.h>

#include <iostream>

#include <list>

#include <vector>

//#define stepik

struct SolutionNode {

size\_t x;

size\_t y;

size\_t w;

SolutionNode(size\_t x = 0, size\_t y = 0, size\_t w = 0) {

this->x = x;

this->y = y;

this->w = w;

}

};

template <class Base, size\_t lsize>

class Static\_list { //быстрое запоминание и удаление элементов массива mem элементов типа Base

public:

void push\_back(Base k) {

mem[last] = k;

last++;

}

Base& pop\_back() {

last--;

return mem[last];

}

bool empty() {

return !last;

}

~Static\_list() {

delete[] mem;

}

Static\_list() :

last(0)

{

mem = new Base[lsize];

}

void copy(const Static\_list& other) {

last = other.last;

if (!mem)

mem = new Base[lsize];

for (int i = 0; i <= other.last; i++) {

mem[i] = other.mem[i];

}

}

private:

size\_t last;

Base\* mem;

};

class MashinaSquaringIterative {

public:

MashinaSquaringIterative(size\_t N = 0) { // N=0 is default

init(N);

}

void init(size\_t N) {

base\_fill = 0;

#ifndef stepik// если не определен

size\_t k; // k - колво квадров которые обязаны участвовать в разложении

std::cin >> k;

size\_t wi; // размеры квадратов

for (size\_t i = 0; i < k; i++) {

std::cin >> wi;

W.push\_back(wi);

base\_fill += wi \* wi; //общая площадь которая составляется квадратами которые предлагается вставить

}

#endif

to\_fill = N \* N;

NMAX = N;

min = 40\*40+1;

columns = new size\_t[N];

for (size\_t i = 0; i < N; i++) {

columns[i] = 0;

}

#ifdef stepik // если определен

int p = get\_p(N);

int k = N / p;

if (N % 2) {

set(0, (p + 1)\*k / 2); // эмпирически можно поставить три квадрата

set(0, (p - 1)\*k / 2); //

set((p + 1)\*k / 2, (p - 1)\*k / 2);

}

else {

set(0, N / 2);

set(0, N / 2);

set(N / 2, N / 2);

}

#endif

}

int get\_p(int N) { // ищу p - минимальный простой множитель числа N

if (!(N % 2)) {

return 2;

}

for (int i = 1; (2 \* i + 1) < sqrt(N); i++) { // проверяем все нечетные числа начиная с 3 которые меньше sqrt(N)

if (!(N % (2 \* i + 1))) {

return 2 \* i + 1;

}

}

return N;

}

void print\_columns() {

for (int i = 0; i < NMAX; i++) std::cout << columns[i] << ' ';

std::cout << std::endl;

}

void set(size\_t column, size\_t w) {

to\_fill -= w \* w;

count++;

IntermediateSolution.push\_back(SolutionNode(column, columns[column], w)); // промежуточное заполнение (добавляю в конец static листа нового объекта Solution Node)

for (size\_t i = column; i < column + w; i++) {

columns[i] += w;

}

}

void unset(size\_t column, size\_t w) {

to\_fill += w \* w;

count--;

IntermediateSolution.pop\_back(); // убрать последний

for (size\_t i = column; i < column + w; i++) {

columns[i] -= w;

}

}

size\_t get\_column() { // ищу колонку куда можно вставить квадрат

int i = NMAX - 1; // NMAX-размер столешницы

if (columns[0] == 0) return 0;

//if no area to fill, conclude no empty spots

if (to\_fill == 0) {

return -1;

}

while (i != 0 && columns[i] >= columns[i - 1]) { // просмотр справа налево

i--;

}

return i;

}

size\_t get\_max\_w(size\_t column) {// ищу максимальный размер квадрата который можно вставить

size\_t max\_w = 0;

while ((column + max\_w) < NMAX && columns[column] >= columns[column + max\_w] && (NMAX - columns[column + max\_w]) > max\_w && max\_w < (NMAX - 1)) { //увеличиваем пока не уткнулись

max\_w++;

}

return max\_w;

}

void saveSolution() { //сохранить решение (удачное заполнение)

Solution.copy(IntermediateSolution);

}

void print\_solution() {

if (Solution.empty()) {

std::cout << "No solutions found.";

}

else {

std::cout << min << std::endl;

SolutionNode n;

while (!Solution.empty()) {

n = Solution.pop\_back();

std::cout << (n.x) + 1 << ' ' << (n.y) + 1 << ' ' << (n.w) << std::endl;

}

}

}

int lookInW(size\_t wi) { // посмотреть есть ли в списке размеров квадратов данный размер

int res = -1;

for (int i = 0; i < W.size(); i++) {

if (W[i] == wi) {

res = i;

break;

}

}

return res;

}

void takeFromW(size\_t index) {

size\_t wi = W[index];

W[index] = 0;

base\_fill -= wi \* wi;

}

void insertInW(size\_t index, size\_t wi) {

W[index] = wi;

base\_fill += wi \* wi;

}

/\*

Заполняем по алгоритму.

Если данный поставленный квадрат содержится в W, base\_fill -= wi\*wi, Убрать из W wi.

Доп. условие остановки перебора (бэктрекинга) - to\_fill < base\_fill

таким образом, гарантируется, что не будет найдено решение, не содержащее всех квадратов из списка

При снятии квадрата ширины wi добавить его обратно в W, если он содержался, base\_fill += wi\*wi.

\*/

void iterativeBacktracking() {

/\*

Для того, чтобы "запомнить" квадрат нужно хранить его позицию, размер, и индекс квадрата в W [ -1 -> не принадлежит W].

\*/

//списки позиций, размеров и индексов для подобия стека

std::list<int> position;

std::list<size\_t> sizew;

std::list<int> indexInW;

//bool wasInW = false;

int wIndex;

int column = 0;

int wi = 0;

size\_t step = 1;

bool ret = true;

while (ret) {

switch (step) {

case 1: {

int temp = column;

column = get\_column();

if (column == -1) { //не найдено место для вставки (столешница заполнена)

column = temp;

step = 4;

}

else {

step = 2;

}

} break;

case 2: {

if (count >= min - 1 || to\_fill < base\_fill) { // останавливаю перебор если Не удовлетворяет либо по количеству квадратов, либо по конфигурации

step = 6;

}

else {

wi = get\_max\_w(column);

step = 3;

}

} break;

case 3: {

//Проверка принадлежности wi к W

wIndex = lookInW(wi);

if (wIndex != -1) { //принадлежит

takeFromW(wIndex);

}

set(column, wi);

//запомнить квадрат

position.push\_back(column);

sizew.push\_back(wi);

indexInW.push\_back(wIndex);

step = 1;

} break;

case 4: {

saveSolution();

min = count;

step = 6;

} break;

case 5: {

unset(column, wi);

if (wIndex != -1) {

insertInW(wIndex, wi);

}

wi--;// снимаю поставленный квадрат, уменьшая размер на 1

if (wi > 0) {

step = 3;

}

else {

step = 6;

}

} break;

case 6: {

if (position.empty()) //До данного момента не было поставлено ни одного квадрата

ret = false;

else {

step = 5;

//рассматриваем предыдущий

column = position.back();

wi = sizew.back();

wIndex = indexInW.back();

position.pop\_back();

sizew.pop\_back();

indexInW.pop\_back();

}

} break;

}

}

}

~MashinaSquaringIterative() {

delete[] columns;

}

private:

//Найденное решение

Static\_list<SolutionNode, 40 \* 40> Solution;

//отслеживание текущего заполнения столешницы

Static\_list<SolutionNode, 40 \* 40> IntermediateSolution;

//список квадратов, которые обязательно должны учавствовать в разложении

std::vector<size\_t> W;

//суммарная площадь квадратов, которые обязательно должны учавствовать в разложении

size\_t base\_fill;

//площадь столешницы, которую необходимо покрыть квадратами

size\_t to\_fill;

//Размер столешницы

size\_t NMAX;

//массив заполненности колонок сетки, на которой располагается столешница

size\_t\* columns;

//текущее количество поставленных квадратов

size\_t count = 0;

//минимальное количество квадратов, которыми удалось покрыть столешницу

size\_t min;

};

int main()

{

size\_t N;

std::cin >> N;

MashinaSquaringIterative solver(N);

solver.iterativeBacktracking();

solver.print\_solution();

return 0;

}

**Приложение 2. Код программы с промежуточным выводом.**

#include "pch.h"

#include "math.h"

#include <iostream>

#include <list>

#include <vector>

//#define stepik

struct SolutionNode {

size\_t x;

size\_t y;

size\_t w;

SolutionNode(size\_t x = 0, size\_t y = 0, size\_t w = 0) {

this->x = x;

this->y = y;

this->w = w;

}

};

template <class Base, size\_t lsize>

class Static\_list { //быстрое запоминание и удаление элементов массива mem элементов типа Base

public:

void push\_back(Base k) {

mem[last] = k;

last++;

}

Base& pop\_back() {

last--;

return mem[last];

}

bool empty() {

return !last;

}

~Static\_list() {

delete[] mem;

}

Static\_list() :

last(0)

{

mem = new Base[lsize];

}

void copy(const Static\_list& other) {

last = other.last;

if (!mem)

mem = new Base[lsize];

for (int i = 0; i <= other.last; i++) {

mem[i] = other.mem[i];

}

}

private:

size\_t last;

Base\* mem;

};

class MashinaSquaringIterative {

public:

MashinaSquaringIterative(size\_t N = 0) { // N=0 is default

init(N);

}

void init(size\_t N) {

base\_fill = 0;

#ifndef stepik// если не определен

size\_t k; // k - колво квадров которые обязаны участвовать в разложении

std::cout << "Enter the ammount of squares to participate in placing:" << std::endl;

std::cin >> k;

size\_t wi; // размеры квадратов

for (size\_t i = 0; i < k; i++) {

std::cout << "Enter the size (width) of square number "<< i+1 << ":"<< std::endl;

std::cin >> wi;

W.push\_back(wi);

base\_fill += wi \* wi; //общая площадь которая составляется квадратами которые предлагается вставить

}

#endif

to\_fill = N \* N;

NMAX = N;

min = 40\*40+1;

columns = new size\_t[N];

for (size\_t i = 0; i < N; i++) {

columns[i] = 0;

}

#ifdef stepik // если определен

int p = get\_p(N);

int k = N / p;

if (N % 2) {

set(0, (p + 1)\*k / 2); // эмпирически можно поставить три квадрата

set(0, (p - 1)\*k / 2); //

set((p + 1)\*k / 2, (p - 1)\*k / 2);

}

else {

set(0, N / 2);

set(0, N / 2);

set(N / 2, N / 2);

}

#endif

}

int get\_p(int N) { // ищу p - минимальный простой множитель числа N

if (!(N % 2)) {

return 2;

}

for (int i = 1; (2 \* i + 1) < sqrt(N); i++) { // проверяем все нечетные числа начиная с 3 которые меньше sqrt(N)

if (!(N % (2 \* i + 1))) {

return 2 \* i + 1;

}

}

return N;

}

void print\_columns() {

for (int i = 0; i < NMAX; i++) std::cout << columns[i] << ' ';

std::cout << std::endl;

}

void set(size\_t column, size\_t w) {

to\_fill -= w \* w;

count++;

IntermediateSolution.push\_back(SolutionNode(column, columns[column], w)); // промежуточное заполнение (добавляю в конец static листа нового объекта Solution Node)

for (size\_t i = column; i < column + w; i++) {

columns[i] += w;

}

}

void unset(size\_t column, size\_t w) {

to\_fill += w \* w;

count--;

IntermediateSolution.pop\_back(); // убрать последний

for (size\_t i = column; i < column + w; i++) {

columns[i] -= w;

}

}

size\_t get\_column() { // ищу колонку куда можно вставить квадрат

int i = NMAX - 1; // NMAX-размер столешницы

if (columns[0] == 0) return 0;

//if no area to fill, conclude no empty spots

if (to\_fill == 0) {

return -1;

}

while (i != 0 && columns[i] >= columns[i - 1]) { // просмотр справа налево

i--;

}

return i;

}

size\_t get\_max\_w(size\_t column) {// ищу максимальный размер квадрата который можно вставить

size\_t max\_w = 0;

while ((column + max\_w) < NMAX && columns[column] >= columns[column + max\_w] && (NMAX - columns[column + max\_w]) > max\_w && max\_w < (NMAX - 1)) { //увеличиваем пока не уткнулись

max\_w++;

}

return max\_w;

}

void saveSolution() { //сохранить решение (удачное заполнение)

Solution.copy(IntermediateSolution);

}

void print\_solution() {

if (Solution.empty()) {

std::cout << "No solutions found.";

}

else {

std::cout << min << std::endl;

SolutionNode n;

while (!Solution.empty()) {

n = Solution.pop\_back();

std::cout << (n.x) + 1 << ' ' << (n.y) + 1 << ' ' << (n.w) << std::endl;

}

}

}

int lookInW(size\_t wi) { // посмотреть есть ли в списке размеров квадратов данный размер

int res = -1;

for (int i = 0; i < W.size(); i++) {

if (W[i] == wi) {

res = i;

break;

}

}

return res;

}

void takeFromW(size\_t index) {

size\_t wi = W[index];

W[index] = 0;

base\_fill -= wi \* wi;

}

void insertInW(size\_t index, size\_t wi) {

W[index] = wi;

base\_fill += wi \* wi;

}

/\*

Заполняем по алгоритму.

Если данный поставленный квадрат содержится в W, base\_fill -= wi\*wi, Убрать из W wi.

Доп. условие остановки перебора (бэктрекинга) - to\_fill < base\_fill

таким образом, гарантируется, что не будет найдено решение, не содержащее всех квадратов из списка

При снятии квадрата ширины wi добавить его обратно в W, если он содержался, base\_fill += wi\*wi.

\*/

void iterativeBacktracking() {

/\*

Для того, чтобы "запомнить" квадрат нужно хранить его позицию, размер, и индекс квадрата в W [ -1 -> не принадлежит W].

\*/

//списки позиций, размеров и индексов для подобия стека

std::list<int> position;

std::list<size\_t> sizew;

std::list<int> indexInW;

//bool wasInW = false;

int wIndex;

int column = 0;

int wi = 0;

size\_t step = 1;

bool ret = true;

while (ret) {

switch (step) {

case 1: {

std::cout << "Step 1: finding the first free cell of the grid. " ;

int temp = column;

column = get\_column();

if (column == -1) { //не найдено место для вставки (столешница заполнена)

column = temp;

step = 4;

std::cout << "A free cell hasn't been found. Going to step 4" << std::endl;

}

else {

step = 2;

std::cout << "A free cell has been found. Going to step 2" << std::endl;

}

} break;

case 2: {

std::cout << "Step 2: " ;

if (count >= min - 1 ) { // останавливаю перебор если Не удовлетворяет либо по количеству квадратов, либо по конфигурации

step = 6;

std::cout << "current ammount of squares is larger than one of the previous solutions's minimal ammount of squares - 1. No point in proceeding. Going to step 6" << std::endl;

}

else if ( to\_fill < base\_fill) { // останавливаю перебор если Не удовлетворяет либо по количеству квадратов, либо по конфигурации

step = 6;

std::cout << "ammount of cells to be filled is smaller than the ammount of cells to be placed. No point in proceeding. Going to step 6" << std::endl;

}

else {

wi = get\_max\_w(column);

step = 3;

std::cout << "getting the maximal size (width) of a square. It's " << wi << ". Going to step 3" << std::endl;

}

} break;

case 3: {

std::cout << "Step 3: placing the square, remembering it as the previously placed. Going to step 1" << std::endl;

//Проверка принадлежности wi к W

wIndex = lookInW(wi);

if (wIndex != -1) { //принадлежит

takeFromW(wIndex);

}

set(column, wi);

//запомнить квадрат

position.push\_back(column);

sizew.push\_back(wi);

indexInW.push\_back(wIndex);

step = 1;

} break;

case 4: {

std::cout << "Step 4: saving solution and ammount of squares in it" << std::endl;

saveSolution();

min = count;

step = 6;

} break;

case 5: {

std::cout << "Step 5: removing the square that has just been placed. Decreasing its size (width) by 1. " ;

unset(column, wi);

if (wIndex != -1) {

insertInW(wIndex, wi);

}

wi--;// снимаю поставленный квадрат, уменьшая размер на 1

if (wi > 0) {

std::cout << "New size (width) is larger than 0. Going to step 3" << std::endl;

step = 3;

}

else {

std::cout << "New size (width) equals 0. Going to step 6" << std::endl;

step = 6;

}

} break;

case 6: {

std::cout << "Step 6: ";

if (position.empty()) //До данного момента не было поставлено ни одного квадрата

{

std::cout << "No squares have been placed still. Algorithm is completed" << std::endl;

ret = false;

}

else {

std::cout << "Some squares have been placed already. Reviewing the previously placed square. Going to step 5" << std::endl;

step = 5;

//рассматриваем предыдущий

column = position.back();

wi = sizew.back();

wIndex = indexInW.back();

position.pop\_back();

sizew.pop\_back();

indexInW.pop\_back();

}

} break;

}

}

}

~MashinaSquaringIterative() {

delete[] columns;

}

private:

//Найденное решение

Static\_list<SolutionNode, 40 \* 40> Solution;

//отслеживание текущего заполнения столешницы

Static\_list<SolutionNode, 40 \* 40> IntermediateSolution;

//список квадратов, которые обязательно должны учавствовать в разложении

std::vector<size\_t> W;

//суммарная площадь квадратов, которые обязательно должны учавствовать в разложении

size\_t base\_fill;

//площадь столешницы, которую необходимо покрыть квадратами

size\_t to\_fill;

//Размер столешницы

size\_t NMAX;

//массив заполненности колонок сетки, на которой располагается столешница

size\_t\* columns;

//текущее количество поставленных квадратов

size\_t count = 0;

//минимальное количество квадратов, которыми удалось покрыть столешницу

size\_t min;

};

int main()

{

size\_t N;

std::cout << "Enter the size of the grid:" << std::endl;

std::cin >> N;

MashinaSquaringIterative solver(N);

solver.iterativeBacktracking();

solver.print\_solution();

return 0;

}