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

Студент гр. 1304 _____ Кривоченко Д.И.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Написать алгоритм, решающий задачу, используя перебор с возвратом. Изучить применения метода ветвей и границ на практике.

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.

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 <= 20.

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.

В ходе работы использован метод backtracking (перебор с возвратом). Это метод для решений задачи, в которой требуется перебор всех возможных вариантов. Конкретнее – использован метод ветвей и границ (заведомо неоптимальные решения отсеиваются).

Выполнение работы.

Пусть разбиение квадрата – количество обрезков, из которых можно построить столешницу заданного размера, а карта – вектор векторов целочисленных значений, являющийся иллюстрацией столешницы в определённый момент.

Благодаря эмпирическим данным (часть из которых на рисунке 1), для заданных значений N были замечены следующие факты:

1) Для чётных N можно добиться разбиения на 4 квадрата с размером стороны $\frac{N}{2}$.

2) Для N, кратных трём, можно добиться разбиения на 6 квадратов, при том, у одного из них – размер $N * \frac{2}{3}$, а у пяти остальных - $N * \frac{1}{3}$.

3) Для простых N в разбиении есть минимум один квадрат размера $\frac{N+1}{2}$, и как минимум два квадрата размера $N - \frac{N+1}{2}$, смежных с ним.

4) Для заданных N любое минимальное разбиение не превышает 13 квадратов.

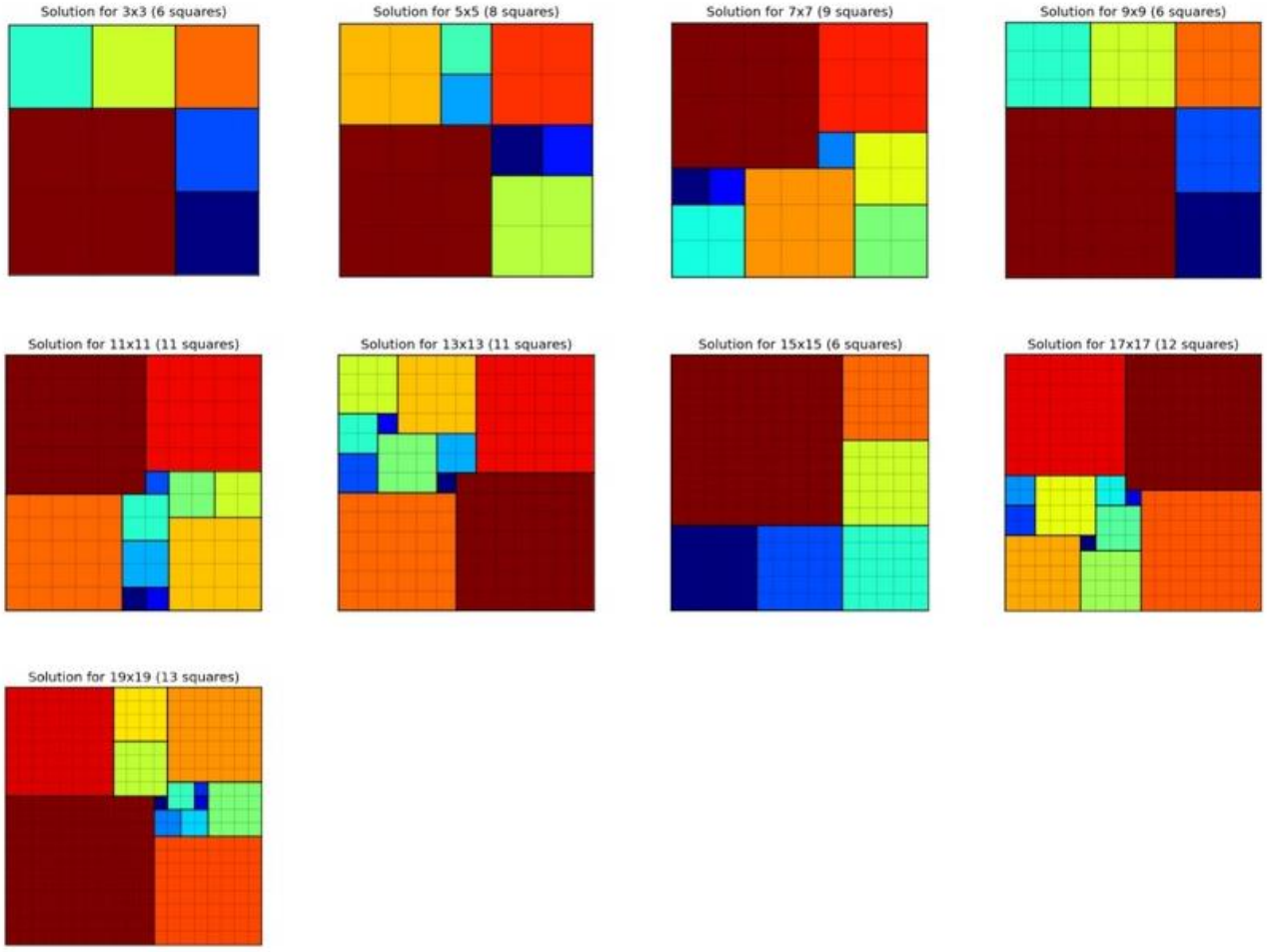

Рисунок 1 - Нечётные разбиения до N = 20. Источник - https://laurentlessard.com/bookproofs/squaring-the-square/

Эти факты были применены при написании кода. Рассмотрим функции и структуры:

1) Структура *vec3d* содержит поля *x*, *y*, *w*. Она служит вспомогательным звеном для вывода информации после разбиения (из этих структур формируется вектор, который выводится на экран после завершения перебора).

2) Функция *printAnswer* выводит ответ в консоль.

3) Функция *makeVec3D* создаёт *vec3d* согласно входным данным и возвращает её.

4) Функция *getBestVtx* создаёт ответ для квадратов с простым N по лучшему разбиению, который нужно вывести на консоль (которое имеет вид вектора векторов для удобства восприятия).

5) Функция *getBestVtxEven* создаёт ответ для квадратов с чётным N.

6) Функция *getBestVtxDivisableByThree* создаёт ответ для квадратов с N, делящимся на 3.

7) Функция *tryToPutSquare* проверяет, можно ли, начиная с данной пустой клетки, построить квадрат размера *sizeOfSquareToPut*.

8) Функция *putSquare* строит квадрат размера *sizeOfSquareToPut* начиная с данной пустой клетки (эта клетка – левый верхний угол квадрата).

9) Функция *divide* – рекурсивная функция, роль которой – нахождение минимального разбиения. Она оканчивает свою работу, если очередное переданное ей разбиение больше, чем минимальное на данном этапе, или если карта заполнена. Внутри функции происходит перебор по трём циклам: первые два отвечают за итерацию по массиву, характеризующему карту на данном этапе, а третий – за итерацию по потенциальным размерам квадратов, которые можно вставить в данную клетку. Заполнен квадрат полностью или нет определяется с помощью числа *freeArea* (площадь свободных клеток на карте). При том, после помещения очередного квадрата на карту, в случае если *freeArea* больше нуля, эта «новая» карта опять передаётся в функцию *divide*. Иначе – происходит проверка на то, что текущее разбиение минимально и если это так – происходит переприсваивание. При этом, если в третьем цикле размер квадрата-кандидата на

вставку – один, то происходит возврат из функции. Это предотвращает перебор по сути одинаковых разбиений.

10) Функция *getCleanMap* возвращает пустую карту, заполненную нулями.

11) Функция *getOptimalStartForPrimes* является применением третьего факта, рассмотренного выше. Она выставляет на карту три квадрата рассчитанного размера, чтобы уменьшить количество неоптимальных вариантов ещё до начала перебора.

12) Функция *getSolution* запускает работу алгоритма в зависимости от размера стороны карты, введённого пользователем (использует вспомогательные функции *getSolutionDevisableByThree* и *getSolutionForEven*).

13) Функция *readMapSize* – вспомогательная, отвечает за считывание размера карты с клавиатуры.

В качестве хранения ответа и промежуточных данных используются глобальные переменные *bestCounter* (целочисленное число, хранит минимальное разбиение карты), *bestMap* (вектор векторов целочисленных чисел, хранит карту), *bestVec* (вектор *vec3d*, хранит карту в формате, требуемом в задании). Исходный код приведён в приложении А.

Выводы.

Написан алгоритм, находящий минимальное разбиение квадрата с заданной стороной. Изучен метод ветвей и границ. Программой успешно пройдены тесты на платформе *Stepik* для $N \leq 20$. Были применены несколько оптимизаций для прохождения задания за указанное в *Stepik* время, а именно: генерация готового разбиения для любого чётного N; генерация разбиения чисел, кратных трём с помощью разбиения квадрата со стороной $N = 3$ и масштабирования ответа; значительное уменьшение перебора путём вставки трёх квадратов заданного (вычисляемого по N) размера для остальных квадратов со стороной N; ограничение максимально возможного количества квадратов в разбиении; предотвращение повторного перебора по сути одинаковых разбиений.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <vector>
#include <iostream>
#include <cmath>
#include <string>
#include <chrono>
#include <climits>
#define OPTIMAL_MAX_AMOUNT_OF_SQUARES 13


struct vec3d;
int bestCounter = INT_MAX;
std::vector<std::vector<int>> bestMap;
std::vector<vec3d> bestVec;


void printVec3D(vec3d dummyVec);
void printAnswer(std::vector<vec3d> outputVec, int amountOfColors);
vec3d makeVec3D(int x, int y, int w);
std::vector<vec3d> getBestVtx(std::vector<std::vector<int>>& map, int map-
Size, int amountOfColors);
std::vector<vec3d> getBestVtxEven(int mapSize);
void makeBestVtxDivisableByThree(int mapSize);
bool tryToPutSquare(int row, int col, int sizeOfSquareToPut, std::vec-
tor<std::vector<int>>& map, int mapSize);
void putSquare(int row, int col, int sizeOfSquareToPut, std::vec-
tor<std::vector<int>>& map, int mapSize, int colorCounter);
void divide(int mapSize, std::vector<std::vector<int>>& map, int freeArea,
int colorCounter);
std::vector<std::vector<int>> getCleanMap(int sizeOfMap);
void getOptimalStartForPrimes(std::vector<std::vector<int>>& map, int
sizeOfMap, int& colorCounter, int& freeArea);
void getSolutionForEven(int mapSize);
void getSolution();
void readMapSize(int& mapSize);

int main() {
	getSolution();
	return 0;
}

//Содержит три поля типа int
struct vec3d {
	int x;
	int y;
	int w;
};
//Выводит vec3d в консоль
void printVec3D(vec3d dummyVec) {
	std::cout << dummyVec.x << " " << dummyVec.y << " " << dummyVec.w <<
'\n';
```

```
}
//Выводит ответ в требуемом в задаче формате в консоль
void printAnswer(std::vector<vec3d> outputVec, int amountOfColors) {
	std::cout << amountOfColors << std::endl;
	for (int i = 0; i < outputVec.size(); i++) {
		printVec3D(outputVec[i]);
	}
}
//Создаёт и возвращает структуру vec3d по трём целочисленным значениям
vec3d makeVec3D(int x, int y, int w) {
	vec3d vecToReturn;
	vecToReturn.x = x;
	vecToReturn.y = y;
	vecToReturn.w = w;
	return vecToReturn;
}
//Создаёт и возвращает вектор vec3d, содержащий ответ в требуемом в задаче
формате по карте, её размеру и числу квадратов в разбиении.
std::vector<vec3d> getBestVtx(std::vector<std::vector<int>>& map, int map-
Size, int amountOfColors) {
	std::vector<vec3d> vecToReturn;
	std::vector<int> colorCounterArr(20, 0);
	int curColor = 0;
	std::vector<int>::iterator it;
	for (int i = 0; i < mapSize; i++) {
		for (int j = 0; j < mapSize; j++) {
			if (colorCounterArr[map[i][j]] == 0) {
				vecToReturn.push_back(makeVec3D(i, j, map[i][j]));
				curColor = map[i][j];
			}
			colorCounterArr[map[i][j]]++;
		}
	}

	for (int i = 0; i < vecToReturn.size(); i++) {
		vecToReturn[i].x++;
		vecToReturn[i].y++;
		vecToReturn[i].w	=	(int)sqrt(colorCounterArr[vecTo-
Return[i].w]);
	}
	return vecToReturn;
}
//Создаёт и возвращает вектор vec3d, содержащий ответ в требуемом в задаче
формате для чётных по размеру карты
std::vector<vec3d> getBestVtxEven(int mapSize) {
	int numForDisecting = mapSize / 2;
	std::vector<vec3d> vecToReturn;
	vecToReturn.push_back(makeVec3D(0 + 1, 0 + 1, numForDisecting));
	vecToReturn.push_back(makeVec3D(0 + 1, numForDisecting + 1, num-
ForDisecting));
	vecToReturn.push_back(makeVec3D(numForDisecting + 1, 0 + 1, num-
ForDisecting));
	vecToReturn.push_back(makeVec3D(numForDisecting + 1, numForDisecting
+ 1, numForDisecting));
	return vecToReturn;
}
```

```
//Выводит ответ в консоль, в заданном в задаче формате для кратных трём.
На вход -
void makeBestVtxDivisableByThree(int mapSize) {
	int scale = mapSize / 3;
	std::vector<vec3d> vecToReturn;
	int smallerSolutionForScale = 3;
	auto map = getCleanMap(smallerSolutionForScale);
	int freeArea = smallerSolutionForScale * smallerSolutionForScale;
	int colorCounter = 1;
	getOptimalStartForPrimes(map, smallerSolutionForScale, colorCounter,
freeArea);
	divide(smallerSolutionForScale, map, freeArea, colorCounter);
	for (auto& vec : bestVec) {
		vec.x = (vec.x - 1) * scale + 1;
		vec.y = (vec.y - 1) * scale + 1;
		vec.w = (vec.w) * scale;
	}
	printAnswer(bestVec, bestCounter);
}



/*Пробует поставить квадрат в заданную клетку на карте.Возвращает true
если удалось, false - если не удалось.
На вход - челочисленные координаты row, col, размер квадрата для вставки,
карта и её размер*/
bool tryToPutSquare(int row, int col, int sizeOfSquareToPut, std::vec-
tor<std::vector<int>>& map, int mapSize) {
	if ((row + sizeOfSquareToPut > mapSize) || (col + sizeOfSquareToPut
> mapSize)) {
		return false;
	}
	for (int i = row; i < sizeOfSquareToPut + row; i++) {
		if ((map[i][col] != 0) || (map[i][col + sizeOfSquareToPut - 1]
!= 0)) {
			return false;
		}
	}
	for (int j = col; j < sizeOfSquareToPut + col; j++) {
		if ((map[row][j] != 0) || (map[row + sizeOfSquareToPut - 1][j]
!= 0)) {
			return false;
		}
	}
	return true;
}

/*Ставит квадрат в заданную клетку на карте. На вход - челочисленные ко-
ординаты row, col, размер квадрата для вставки, карта и её размер, счётчик
количества квадратов*/
void putSquare(int row, int col, int sizeOfSquareToPut, std::vec-
tor<std::vector<int>>& map, int mapSize, int colorCounter) {
	for (int i = row; i < sizeOfSquareToPut + row; i++) {
		for (int j = col; j < sizeOfSquareToPut + col; j++) {
			map[i][j] = colorCounter;
		}
	}
```

```
}
//Рекурсивная функция, перебирает возможные разбиения. На вход - размер карты, карта, свободная на карте площадь, счётчик квадратов
void divide(int mapSize, std::vector<std::vector<int>>& map, int freeArea, int colorCounter) {
	if (colorCounter >= bestCounter || colorCounter > OPTIMAL_MAX_AMOUNT_OF_SQUARES) {
		return;
	}
	for (int i = (mapSize + 1) / 2 - 1; i < mapSize; i++) {
		for (int j = (mapSize + 1) / 2 - 1; j < mapSize; j++) {
			for (int sizeOfInputSquare = mapSize - (mapSize + 1) / 2; sizeOfInputSquare >= 1; sizeOfInputSquare--) {
				if (tryToPutSquare(i, j, sizeOfInputSquare, map, mapSize)) {
					auto newMap = map;
					putSquare(i, j, sizeOfInputSquare, newMap, mapSize, colorCounter);
					if ((freeArea - sizeOfInputSquare * sizeOfInputSquare) > 0) {
						divide(mapSize, newMap, freeArea - sizeOfInputSquare * sizeOfInputSquare, colorCounter + 1);
					}
					else {
						if (colorCounter < bestCounter) {
							bestMap = newMap;
							bestCounter = colorCounter;
							bestVec = getBestVtx(newMap, mapSize, bestCounter);
						}
						return;
					}
					if (sizeOfInputSquare == 1) {
						return;
					}
				}
			}
		}
	}
}
//Создаёт и возвращает чистую карту заданного размера (вектор векторов целочисленных значений) по размеру на входе
std::vector<std::vector<int>> getCleanMap(int sizeOfMap) {
	std::vector<std::vector<int>> map;
	std::vector<int> tmp(sizeOfMap, 0);
	for (int i = 0; i < sizeOfMap; i++) {
		map.push_back(tmp);
	}
	return map;
}
/*Заполняет чистую карту тремя квадратами, сокращающими количество вариантов, которых надо перебирать
На вход - карта, её размер, счётчик квадратов и свободная площадь*/
void getOptimalStartForPrimes(std::vector<std::vector<int>>& map, int sizeOfMap, int& colorCounter, int& freeArea) {
	int sizeOfSquareOne = (sizeOfMap + 1) / 2;
	int sizeOfSquareTwo = sizeOfMap - sizeOfSquareOne;
	putSquare(0, 0, sizeOfSquareOne, map, sizeOfMap, colorCounter++);
```

```
	putSquare(sizeOfSquareOne, 0, sizeOfSquareTwo, map, sizeOfMap, colorCounter++);
	putSquare(0, sizeOfSquareOne, sizeOfSquareTwo, map, sizeOfMap, colorCounter++);
	freeArea -= sizeOfSquareOne * sizeOfSquareOne + 2 * sizeOfSquareTwo * sizeOfSquareTwo;
}
//Вспомогательная для getSolution функция (выводит ответ в консоль по размеру карты для квадратов с чётной стороной)
void getSolutionForEven(int mapSize) {
	auto bestVec = getBestVtxEven(mapSize);
	printAnswer(bestVec, 4);
}
//Запускает решение задачи
void getSolution() {
	int mapSize;
	readMapSize(mapSize);
	if (mapSize % 2 == 0) {
		getSolutionForEven(mapSize);
	}
	else if (mapSize % 3 == 0) {
		makeBestVtxDivisableByThree(mapSize);
	}
	else {
		auto map = getCleanMap(mapSize);
		int freeArea = mapSize * mapSize;
		int colorCounter = 1;
		getOptimalStartForPrimes(map, mapSize, colorCounter, freeArea);
		divide(mapSize, map, freeArea, colorCounter);
		printAnswer(bestVec, bestCounter);
	}
}
//Считывает размер карты (получает на вход целочисленное значение по адресу)
void readMapSize(int& mapSize) {
	std::cin >> mapSize;
	}
```