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. 9383 _____ Рыбников Р.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

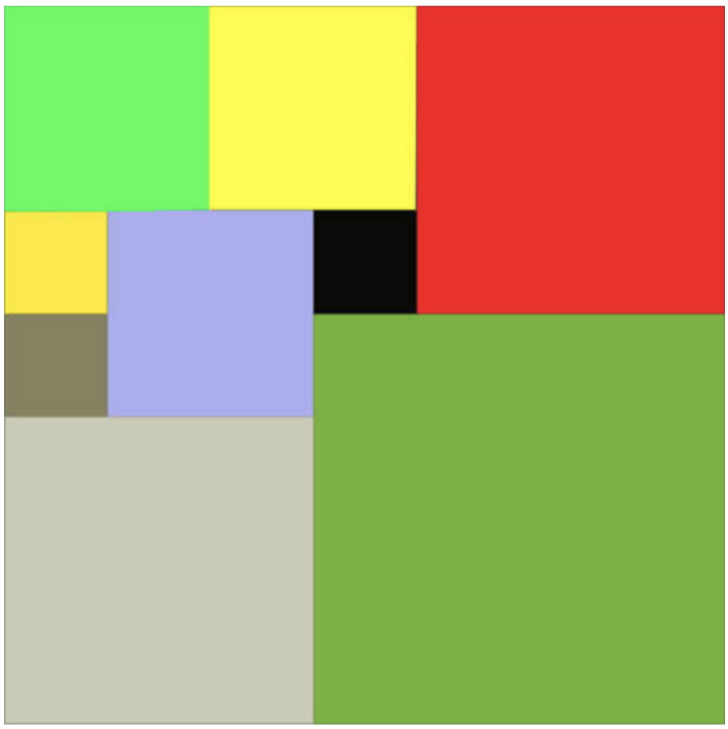
2021

Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов.

Задание. Вариант – 2р(рекурсивный бэктрекинг).

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).



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 ($2 \leq N \leq 40$).

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 ($1 \leq x,y \leq N$) и длину стороны соответствующего обрезка(квадрата).

Описание алгоритма.

Для решения задачи был реализован итеративный алгоритм бэктрекинга, который перебирает все возможные заполнения квадрата квадратами меньшей стороны:

1. Алгоритм основан на поиске с возвратом: идёт вставление всех возможных квадратов подряд
2. Если разложение, которое мы получили на первом шаге является минимальным на текущий момент, то запоминаем его.
3. После, алгоритм откатывается назад до тех пор, пока не встретит квадрат размера >1, затем удаляет его и ставит на его место квадрат размер на 1 больше.
4. Конец работы алгоритма наступит тогда, когда удалить квадрат будет невозможно.

Замечание.

- Квадраты с четной стороной можно всегда разделять на 4 квадрата.
- Если размер столешницы число *составное*, то разбиение на квадраты аналогично разбиению квадрата размером равным наименьшему простому делителю размера столешницы.
- Если размер столешницы число *простое*, то вся столешница имеет квадраты различных размеров в трех углах: в первом и втором углу квадрат размером n/2, которые касаются сторонами третьего квадрата и столешницы, в третьем углу находится квадрат размером n/2 + 1.

Описание функций и структур данных.

- struct Position -- позиция на столе.
- struct Item -- элемент столешницы.
- std::list<Item> GetRes() -- функция поиска результата.
- std::list<Item> GetResEven() -- функция поиска результата, если размер столешницы четный.
- bool PutItem() -- функция, которая пытается добавить ещё один квадрат в текущую раскладку столешницы.
- void Add() -- функция, которая добавляет конкретный квадрат в столешницу.
- void RemoveLast() -- функция, которая удаляет последний элемент из раскладки.
- bool IsFull() -- функция, которая проверяет заполненность таблицы
- bool CanAdd() -- функция, которая проверяет возможность добавить указанный квадрат в таблицу
- bool GetFirstEmpty() -- функция, которая возвращает первую свободную позицию.
- int main() -- основная функция программы, которая занимается поиском результата, а так же (исходя из задания варианта) замеряет время выполнения вычислений.

Тестирование.

```
Размер столешницы N = 40

Количество элементов столешницы = 4

Координаты элементов:
1 1 20
1 21 20
21 1 20
21 21 20

Время вычисления: 2.7263e-05 секунд.

Program ended with exit code: 0
```

Рисунок 1 -- Тестирование (чётное).

```
Размер столешницы N = 30

Количество элементов столешницы = 4

Координаты элементов:
1 1 15
1 16 15
16 1 15
16 16 15

Время вычисления: 1.0941e-05 секунд.

Program ended with exit code: 0
```

Рисунок 2 -- Тестирование (чётное).

```
Размер столешницы N = 20

Количество элементов столешницы = 4

Координаты элементов:
1 1 10
1 11 10
11 1 10
11 11 10

Время вычисления: 1.0674e-05 секунд.

Program ended with exit code: 0
```

Рисунок 3 -- Тестирование (чётное).

```
Размер столешницы N = 33

Количество элементов столешницы = 6

Координаты элементов:
1 1 22
23 1 11
1 23 11
12 23 11
23 12 11
23 23 11

Время вычисления: 1.956e-05 секунд.

Program ended with exit code: 0
```

Рисунок 4 -- Тестирование (составное).

```
Размер столешницы N = 39

Количество элементов столешницы = 6

Координаты элементов:
1 1 26
27 1 13
1 27 13
14 27 13
27 14 13
27 27 13

Время вычисления: 1.9823e-05 секунд.

Program ended with exit code: 0
```

Рисунок 5 -- Тестирование (составное).

```
Размер столешницы N = 11

Количество элементов столешницы = 11

Координаты элементов:
1 1 6
7 1 5
1 7 5
6 7 1
6 8 1
6 9 3
7 6 1
7 7 2
8 6 1
9 6 3
9 9 3

Время вычисления: 0.0835509 секунд.

Program ended with exit code: 0
```

Рисунок 6 -- Тестирование (простое).

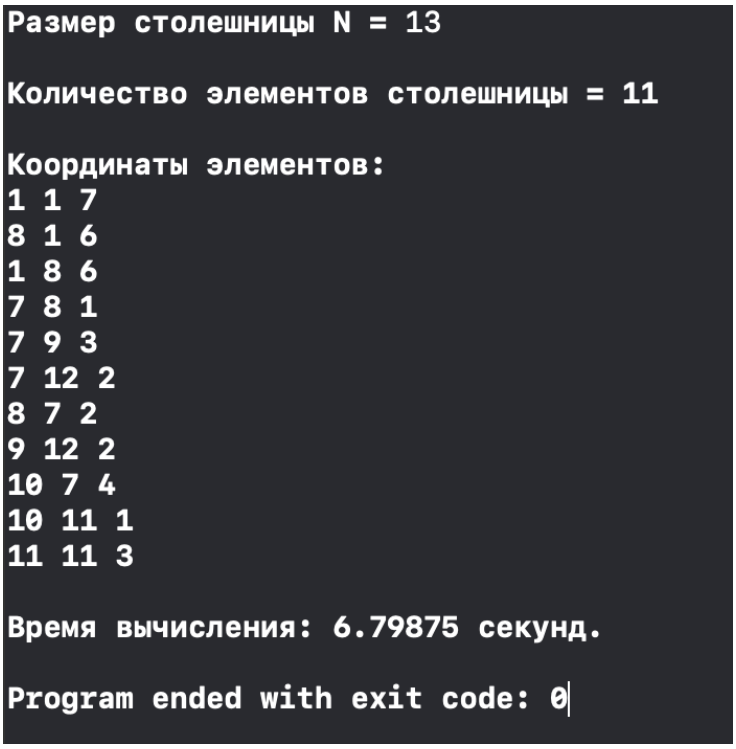


Рисунок 7 -- Тестирование (простое).

Анализ результатов.

Из результатов видно, что при чётном или составном размере столешницы, время вычислений быстрое, но при размере столешницы, равному простому числу, время увеличивается с экспоненциальной зависимостью.

Выводы.

Была реализована программа, которая выполняет поставленную задачу при помощи бэктрекинга, а так же программа была оптимизирована, что позволяет сократить количество операций.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
#include <iostream>
#include <list>
#include <chrono>
using namespace std;

struct Position { // позиция на столе
	int x;
	int y;
};

struct Item {	// один элемент столешницы
	int x;	// координата x
	int y;	// координата y
	int w;	// размер обрезка

	Item(int x, int y, int w) {
		this->x = x;
		this->y = y;
		this->w = w;
	}

	friend std::ostream& operator <<(std::ostream& os, Item item);
};

std::list<Item> GetRes(int N); // поиск результата
std::list<Item> GetResEven(int N); // возвращает результата, если размер четный
bool PutItem(Item item, bool** table, int N, std::list<Item>& list, std::list<Item>& res, int nMax);
// попытка добавить еще один итем в текущую раскладку
void Add(bool** table, std::list<Item>& list, Item item); // добавляет указанный обрезок в раскладку
void RemoveLast(bool** table, std::list<Item>& list); // удаляет последний элемент из раскладки
void ShowTable(bool** table, int N); // вывод таблицы на экран (для дебага)
bool IsFull(bool** table, int N); // заполнена ли таблица
bool CanAdd(bool** table, int N, Item item); // можно ли добавить указанный итем в таблицу
bool GetFirstEmpty(bool** table, int N, Position &position); // возвращает первую свободную позицию


// ----------------------------------------
int main() {
	// ввод данных
	int N;
	std::cout << "Размер столешницы N = ";
	std::cin >> N;
   std::cout << std::endl;
```

```
	// расчет
   chrono::steady_clock::time_point start = chrono::steady_clock::now();
	auto res = GetRes(N);
   chrono::steady_clock::time_point end = chrono::steady_clock::now();
	// вывод результата
   std::cout << "Количество элементов столешницы = " <<res.size() << std::endl << std::endl;
   std::cout << "Координаты элементов: " << std::endl;
	for (auto i = res.begin(); i!=res.end(); ++i)
		std::cout << *i << std::endl;

   cout << endl;
   cout << "Время вычисления: " << std::chrono::duration<double>(end - start).count() << "
секунд." << endl << endl;

	return 0;
}
// ----------------------------------------------------------------------------------------


std::ostream& operator <<(std::ostream& os, Item item) {
	return os << item.x + 1 << ' ' << item.y + 1 << ' ' << item.w; // +1 тк расчер ведем в
системе координат от 0 а на экран требуют от 1
}

// ----------------------------------------------------------------------------------------

std::list<Item> GetRes(int N) { // решает задачу
	// получаем вариант решения
	// если сторона четная
	if (N % 2 == 0) return GetResEven(N);
	// если сторона составная
	int div = 1;
	for (int i = 2; i <= N; ++i) {
		if (N % i == 0) {
			div = N / i;
			N = i;
	break;
		}
	}

	std::list<Item> list;	// текущий список выложенных квадратов
	std::list<Item> res;	// результат

	// создаем двумерный массив, описывающий доску
	bool** table = new bool* [N];
	for (auto i = 0; i < N; ++i) {
		table[i] = new bool[N];
		for (auto j = 0; j < N; ++j)
			table[i][j] = false;
	}
	// вывод дебага
	//std::cout << "scan N = " << N << std::endl;
```

```
	// вставка 3 изначальных квадратов
	Add(table, list, Item(0, 0, N / 2 + 1));
	Add(table, list, Item(N / 2 + 1, 0, N / 2));
	Add(table, list, Item(0, N / 2 + 1, N / 2));

	// ищем первое свободное
	Position pos;
	GetFirstEmpty(table, N, pos);

	// поиск результата (вставкой всех возможных плиток вначало)
	for (int i = 1; i < N - 1; ++i) {
		int nMax = N - i;
		PutItem(Item(pos.x, pos.y, i), table, N, list, res, nMax);
	}

	// удаляем доску
	for (auto i = 0; i < N; ++i) delete[] table[i];
	delete[] table;

	// восстановление размера
	for (auto i = res.begin(); i != res.end();++i) {
		(*i).x *= div;
		(*i).y *= div;
		(*i).w *= div;
	}

	// вывод результата
	return res;
}

// ----------------------------------------------------------------------------------

std::list<Item> GetResEven(int N) // возвращает результата, если размер четный
{
	if (N % 2 != 0) throw "N is not even";
	std::list<Item> res;
	res.push_back(Item(0, 0, N / 2));
	res.push_back(Item(0, N / 2, N / 2));
	res.push_back(Item(N / 2, 0, N / 2));
	res.push_back(Item(N / 2, N / 2, N / 2));

	return res;
}

// ----------------------------------------------------------------------------------

bool PutItem(Item item, bool** table, int N, std::list<Item>& list, std::list<Item>& res,int nMax) //
попытка добавить еще один итем в текущую раскладку
{
	// ограничитель
	if (IsFull(table, N) || !CanAdd(table, N, item)) return false;
```

```
		// вставка итема
		Add(table, list, item);

		// берем индекс первого пустого места
		Position position;
		if (GetFirstEmpty(table, N, position)) {
			// поиск макс размера
			if (N - item.w < nMax)
				nMax = N - item.w;
			// делаем вставку всех итемов, которые возможны
			for (int i = 1; i < nMax; ++i) {
				PutItem(Item(position.x, position.y, i), table, N, list, res, nMax);
			}
		}
		else { // если попали сюда, то доска стала заполненой - проверяем результат
			if (list.size() < res.size() || res.size() == 0)
				res = list;
		}

		// удаление итема
		RemoveLast(table, list);

		return true;

	//тест вставки
	/*
		Add(table, list, Item(0, 0, 2));
		Add(table, list, Item(3, 3, 3));
		Add(table, list, Item(0, 3, 1));
		ShowTable(table, N);*/
}

// ----------------------------------------------------------------------------------

bool GetFirstEmpty(bool** table, int N, Position& position) // возвращает первую свободную позицию
{
	for (int x = 0; x < N; ++x)
		for (int y = 0; y < N; ++y)
			if (!table[x][y]) {
				position.x = x;
				position.y = y;
				return true;
			}
	return false;
}

// ----------------------------------------------------------------------------------

bool CanAdd(bool** table, int N, Item item) // можно ли добавить указанный итем в таблицу
{
```

```
	if (item.x < 0 || item.x + item.w > N) return false;
	if (item.y < 0 || item.y + item.w > N) return false;

	for (int x = item.x; x < item.x + item.w; ++x)
		for (int y = item.y; y < item.y + item.w; ++y)
			if(table[x][y]) return false;
	return true;
}

// ----------------------------------------------------------------------------------------

bool IsFull(bool** table, int N) // заполнена ли таблица
{
	for (int x = 0; x < N; ++x)
		for (int y = 0; y < N; ++y)
			if (!table[x][y]) return false;
	return true;
}

// ----------------------------------------------------------------------------------------

void Add(bool** table, std::list<Item>& list, Item item) // добавляет указанный обрезок в
раскладку
{
	// добавляем в список
	list.push_back(item);

	// в таблицу
	for (int x = item.x; x < item.x + item.w; ++x)
		for (int y = item.y; y < item.y + item.w; ++y)
			table[x][y] = true;
}

// ----------------------------------------------------------------------------------------

void RemoveLast(bool** table, std::list<Item>& list) // удаляет последний элемент из раскладки
{
	// берем последний элемент
	auto item = list.back();
	list.pop_back();

	// чистим таблицу
	for (int x = item.x; x < item.x + item.w; ++x)
		for (int y = item.y; y < item.y + item.w; ++y)
			table[x][y] = false;
}

// ----------------------------------------------------------------------------------------

void ShowTable(bool** table, int N) // вывод таблицы на экран (для дебага)
{
	for (int y = 0; y < N; ++y) {
```

```
                for (int x = 0; x < N; ++x)
                        std::cout << (int)table[x][y];
                std::cout << std::endl;
        }
}
// -------------------------------------------------------------------------------
```