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. 8304 _____ Ястребов И.М.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Вариант 4р

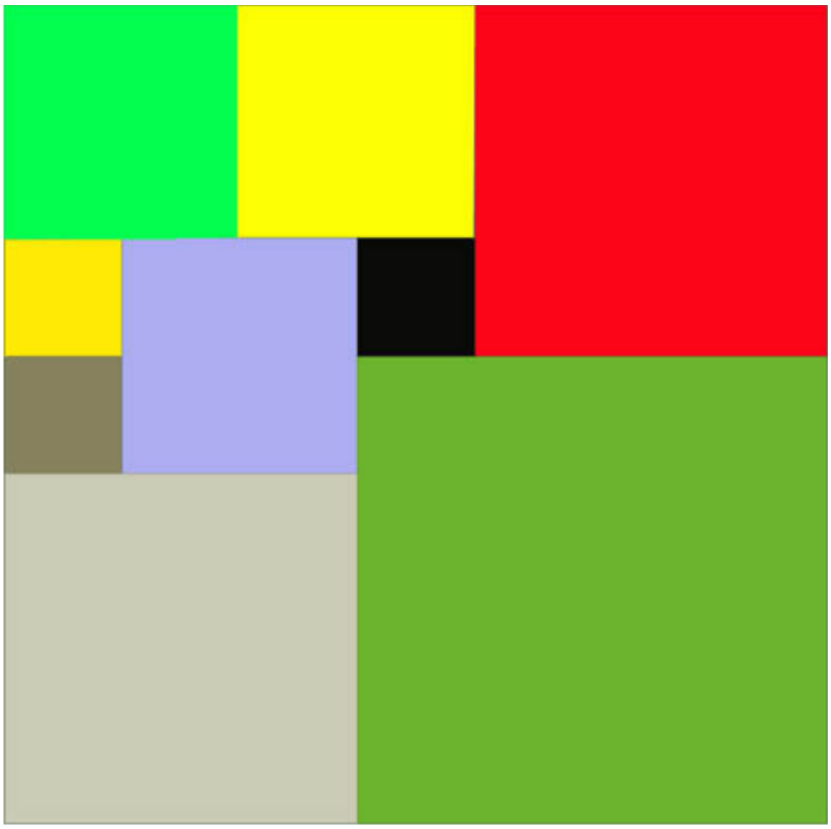
Цель работы

Ознакомиться и закрепить знания, связанные с алгоритмами поиска с возвратом.

Постановка задачи

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 ($2 \le N \le 20$).

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 ($1 \le x, y \le N$) 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

Рекурсивный алгоритм, для каждой клетки квадрата (прямоугольника), для каждого возможного размещенного в этой клетке обрезка, рассматриваются все варианты расположения отрезков в следующей клетке обрезков всех возможных размеров.

Вывод

Был получен опыт работы с алгоритмами поиска с возвратом, реализована программа, рассчитывающая минимальное разбиение квадрата (прямоугольника) на квадраты. Затраты по памяти — O(c*N^2). Затраты по времени растут экспоненциально — O(2 ^ N). Показанный график

соответствия количества вызовов функций обработки и размера входных данных наглядно это иллюстрирует.

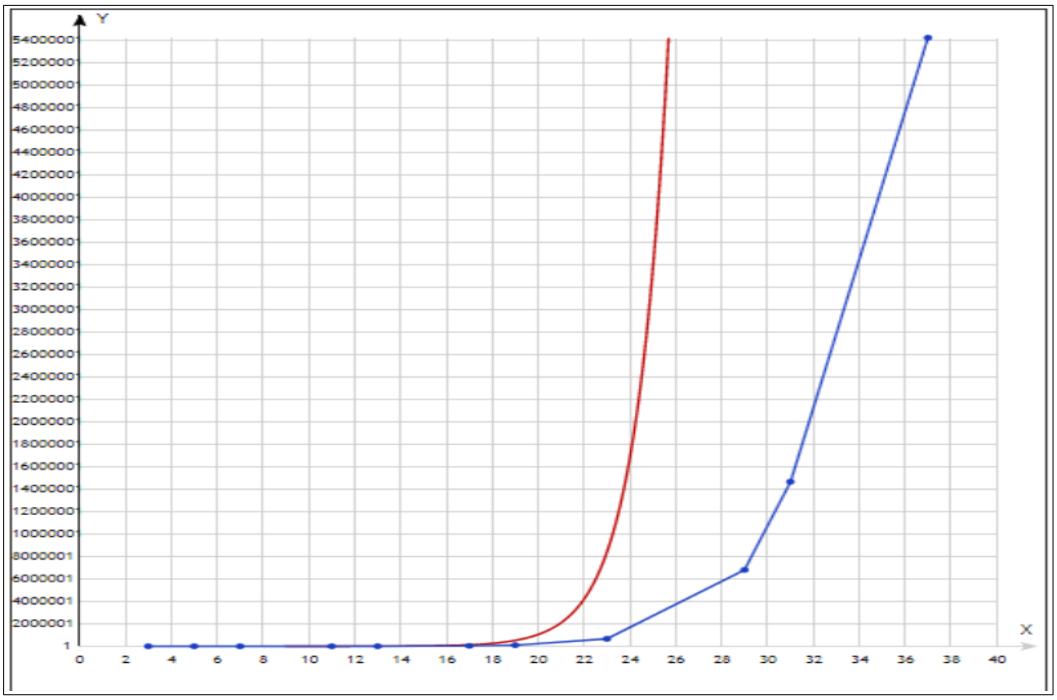
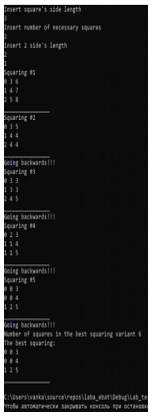


Рис.- График зависимости сложности от размера входных данных

ПРИЛОЖЕНИЕ А.

ТЕСТИРОВАНИЕ.



Здесь рассмотрены оба случая — нахождение разбиения минимального и нахождение разбиения с учетом требуемых компонент

ПРИЛОЖЕНИЕ В.

Исходный код.

```
#include "pch.h"
#include <iostream>
#include <vector>

//size, n - таблицы и подтаблицы
int size = 1;
int n;
int cnt = 0;

//Стрктура, описывающая хар-ки квадратов для в вставки
struct element {
	int x;
	int y;
	int len;
};

//Функция ввода данных
bool input(std::vector<int>& squares) {
	std::cout << "Insert square's side length\n";
	std::cin >> n;

	int quantity;
	std::cout << "Insert number of necessary squares\n";
	std::cin >> quantity;
	if (quantity < 0 || quantity > n*n) {
		std::cout << "Impossible number of squares\n";
		return false;
	}

	int side_l;
```

```
	if (quantity != 0) {
		std::cout << "Insert " << quantity << " side's length\n";
		for (int i = 0; i < quantity; ++i) {
			std::cin >> side_l;
			if (side_l > 0 && side_l <= n - 1) {
				squares.push_back(side_l);
			}
		}
	}
	return true;
}


//Функция проверки частного разбиения
bool check_variant(std::vector<int> squares, std::vector<element> tmp) {

	if (squares.empty())
		return true;

	for (int i = 0; i < tmp.size(); ++i) {
		for (int j = 0; j < squares.size(); ++j) {
			if (tmp[i].len == squares[j]) {
				tmp.erase(tmp.begin() + i);
				i -= 1;
				squares.erase(squares.begin() + j);
				break;
			}
		}
	}
	return squares.empty();
}

// Функция вывода промежуточного решения - вызывается каждый раз, когда исходный квадрат оказывается полностью заполнен
// Разные квадраты обозначаются разными числами в таблице
void print_particular_answer(std::vector<element>& tmp) {
	std::vector<std::vector<int>> view_field(n, std::vector<int>(n, 0));
	for (int i = 0; i < tmp.size(); ++i) {

		int tmp_x = tmp[i].x, tmp_y = tmp[i].y, tmp_s = tmp[i].len;

		for (int y = tmp_y; y < tmp_y + tmp_s; ++y)
			for (int x = tmp_x; x < tmp_x + tmp_s; ++x)
				view_field[y][x] = i;
	}

	for (auto& i : view_field) {
		for (auto& j : i) {
			std::cout << j << ' ';
		}
		std::cout << std::endl;
	}
	std::cout << "_________________\n";
}
```

```
//Рекурсивная функция бэктрекинга
void backtracking(int * field, std::vector<element>& tmp, std::vector<element>&
min, std::vector<int>& squares) {
	bool zero_found = false;
	element current;

	//Проверка на наличие в квадрате пустых мест
	for (int i = 0; i < size; ++i) {
		for (int j = 0; j < size; ++j)
			if (field[i * size + j] == 0) {
				zero_found = true;
				current = { i, j, 1 };
				break;
			}
		if (zero_found)
			break;
	}

	if (!zero_found) {
		std::cout << "Squaring #" << cnt + 1 << std::endl;
		print_particular_answer(tmp);
		if (cnt > 0) {
			std::cout << "Going backwards!!!\n";
		}
		cnt++;

		if (check_variant(squares, tmp))
			min = tmp;
		return;
	}

	//Заполнение поля квадратами со стороной от 1 до n-1
	//Также выполняются проверки на возвожность вставки квадрата
	for (int tmp_len = 1; tmp_len < size; ++tmp_len) {

		if (squares.empty() && tmp.size() + 1 >= min.size()) {
			break;
		}

		if (tmp_len > size - current.x || tmp_len > size - current.y)
			return;

		for (int i = current.x; i < current.x + tmp_len; ++i)
			for (int j = current.y; j < current.y + tmp_len; ++j)
				if (field[i * size + j] == 1)
					return;

		for (int i = current.x; i < current.x + tmp_len; ++i)
			for (int j = current.y; j < current.y + tmp_len; ++j)
				field[i * size + j] = 1;

		current.len = tmp_len;

		tmp.push_back(current);

		backtracking(field, tmp, min, squares);
```

```
            for (int i = current.x; i < current.x + tmp_len; ++i)
                  for (int j = current.y; j < current.y + tmp_len; ++j)
                        field[i * size + j] = 0;
            tmp.pop_back();
      }
}

int main() {

      std::vector<int> squares;

      //Ввод необходимых данных для работы программы
      if (!input(squares)) {
            return 0;
      }
      std::vector<element> tmp;

      //Случай, когда требуется найти миниимальное разложение без обязательных
квадратов
      if (squares.empty()) {

            //Находим наименьший делитель
            for (size = 2; size < n; ++size)
                  if (n % size == 0)
                        break;

            std::vector<element> min(size + 4);

            int multiplier = n / size;

            //Три квадрата для минимального разбиения
            tmp.push_back({ 0, size / 2 + size % 2, size / 2 });
            tmp.push_back({ size / 2 + size % 2, 0, size / 2 });
            tmp.push_back({ size / 2, size / 2, size / 2 + size % 2 });

            //Если сторона исходного квадрата нечётная, то с помощью алгоритма
бэктрекинга
            // находим минимальное разбиение для оставшейся области
            if (size % 2 == 1) {
                  int* field = new int[(size / 2 + 1) * (size / 2 + 1)]();
                  field[(size / 2 + 1) * (size / 2 + 1) - 1] = 1;
                  size = size / 2 + 1;

                  backtracking(field, tmp, min, squares);
            }
            //Для стороны квадрата с чётной длиной минимальное разбиение всегда
состоит из 4-ёх квадратов
            else {
                  tmp.push_back({ 0, 0, 1 });
                  min = tmp;
            }

            //Вывод найденного разбиения
            //Если сторона исходного квадрата была не простым числом, то координаты
ответа умножаются на множитель
            std::cout << "Number of squares in the best squaring variant " <<
```

```
min.size() << std::endl;

            /*for (auto i:min) {
                  std::cout << i.x * multiplier << " " << i.y * multiplier << " "
<< i.len * multiplier << std::endl;
            }*/
            std::cout << "The best squaring:\n";
            print_particular_answer(min);
      }
      //Для случая, когда заданы необходимые для разбиения квадраты
      // просматриваются все варианты разложения без оптимизаций
      else {
            size = n;
            std::vector<element> min(size*size);
            int *field = new int[size * size]();

            backtracking(field, tmp, min, squares);

            //Если ни одно разбиение не содержит необходимых квадратов, то
выводится соответсвующее сообщение
            if (!check_variant(squares, min)) {
                  std::cout << "Impossible variant!" << std::endl;
                  return 0;
            }

            //Вывод найденного минимального разбиения
            std::cout << "Number of squares in the best squaring variant " <<
min.size() << std::endl;
            /*for (auto i:min) {
                  std::cout << i.x << " " << i.y << " " << i.len << std::endl;
            }*/
            std::cout << "The best squaring:\n";
            print_particular_answer(min);
      }
      return 0;
}
```