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. 9383 _____ Лихашва А.Д.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

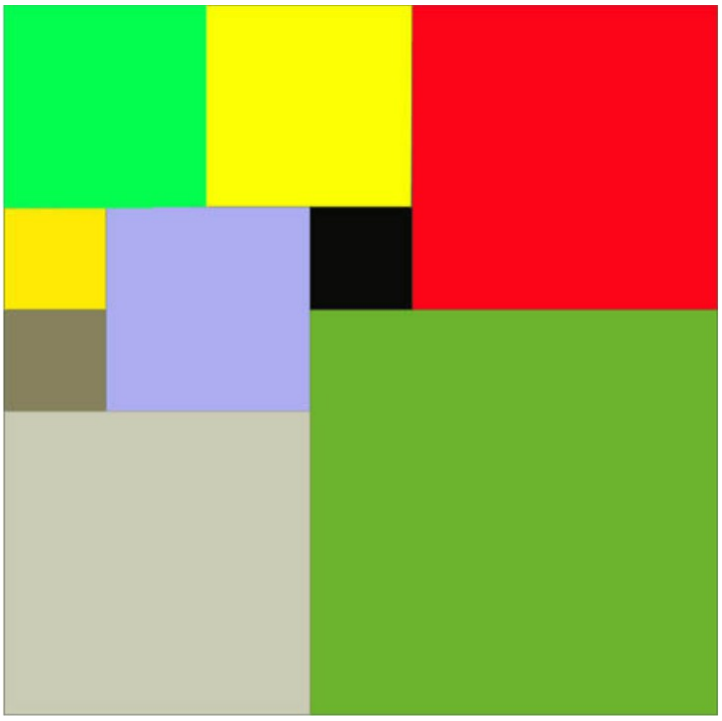
Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения исходного квадрата минимальным количеством квадратов.

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 ≤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, каж-

дая 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

Вариант 2и.

Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

Описание алгоритма:

1. Алгоритм поиска наилучшей упаковки основан на поиске с возвратом: квадраты ставятся подряд от угла, изначально выбирая наибольший возможный размер, пока поле не заполнится.
2. Если понадобилось меньше квадратов, чем на прошлых итерациях, результат сохраняется.
3. Затем алгоритм возвращается назад до тех пор, пока не встретит квадрат размера больше 1, удаляет его и ставит вместо него квадрат меньшего на 1 размера.
4. Алгоритм работает до тех пор, пока все необходимые расстановки не будут проверены.

Замечания для возможной оптимизации алгоритма:

- Карты с четной стороной всегда можно разделить на 4 квадрата без дополнительных вычислений.

- Если N составное, то разбиение будет аналогично разбиению карты с размером наименьшего простого делителя N.
- Если N простое, то в оптимальном разбиении всегда будут присутствовать следующие квадраты:

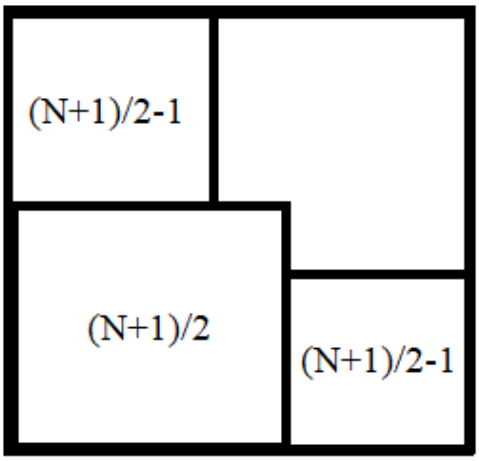


Рисунок 1 - Оптимальное разбиение

Таким образом, любой нечетный случай можно свести к простому N и рассматривать только разбиения, включающие эти 3 квадрата

- При достижении количества квадратов текущего лучшего разбиения можно вернуться назад на минимум 3 шага
- После заполнения карты помимо квадратов со стороной 1 удаляются квадраты со стороной 2, так как их разбиение и их перестановки в уже заполненной карте не изменят результат
- Проверка коллизий (выхода за границу и пересечение с другими квадратами) и поиск наибольшего возможного размера определяются за один цикл
- При определении наибольшего квадрата, который можно поместить, начиная с данной клетки, проверяются коллизии только по клеткам 2 смежных сторон, так как квадраты ставятся подряд, и остальные клетки потенциального квадрата автоматически не могут ни с чем пересечься
- После удаления квадрата запоминаются координаты его угла для использования следующим квадратом

Функции и структуры данных:

Структуры данных:

bool Field – матрица клеток

struct Square – структура хранения данных о квадрате

vector <Square> StepStack – массив квадратов, описывающий заполнение карты

vector <Square> CurrentStack – текущее заполнение

vector <Square> BestStack – заполнение с наименьшим количеством квадратов\

Функции:

void Backtracking(int size) — функция поиска с возвратом, где size – размер столешницы. Алгоритм функции:

- Сокращение размера карты, если N составное
- Расстановка первых 3 квадратов
- Пока не проверены все случаи расстановок квадратов в оставшейся области:
  - Пока карта не заполнена
    - Поставить квадрат максимально возможного размера (меньше удаленного с этого места квадрата, если такой есть) в первую свободную клетку
    - Проверить, не достигло ли текущее заполнения лучшего результата с прошлых итераций, если да, то сделать 2 шага назад и выйти из цикла
    - Найти следующую свободную клетку
  - Сравнить заполнение лучшим, обновить лучшее при необходимости
  - Сделать возврат по рекурсии до первого квадрата, который можно уменьшить
- Масштабировать лучший результат к изначальному размеру карты
- Поставить один квадрат на карту

Square Place(int size, Square Alleged) — функция, которая ставит один квадрат на столешницу, где size – размер столешницы, Alleged – первая пустая клетка или же последний удаленный квадрат. Функция уменьшает размер Alleged на 1; если же нет информации в Alleged, тогда подбирается наибольший размер квадрата. Далее заполняются соответствующие клетки столешницы. Функция возвращает Square – поставленный квадрат.

Square FreeCell(int size) — функция поиска первой свободной клетки, где size – размер столешницы.

Square DeleteSquare(const int &size) — функция возврата, удаляет квадрат, то есть отменяет шаг. Функция возвращает удаленный квадрат.

Исследование времени выполнения от размера квадрата.

Размер квадрата	Время
2	0.0011
3	0.003
7	0.005
11	0.01
13	0.011
17	0.015
19	0.021

График зависимости времени работы программы в секундах от размера квадрата:

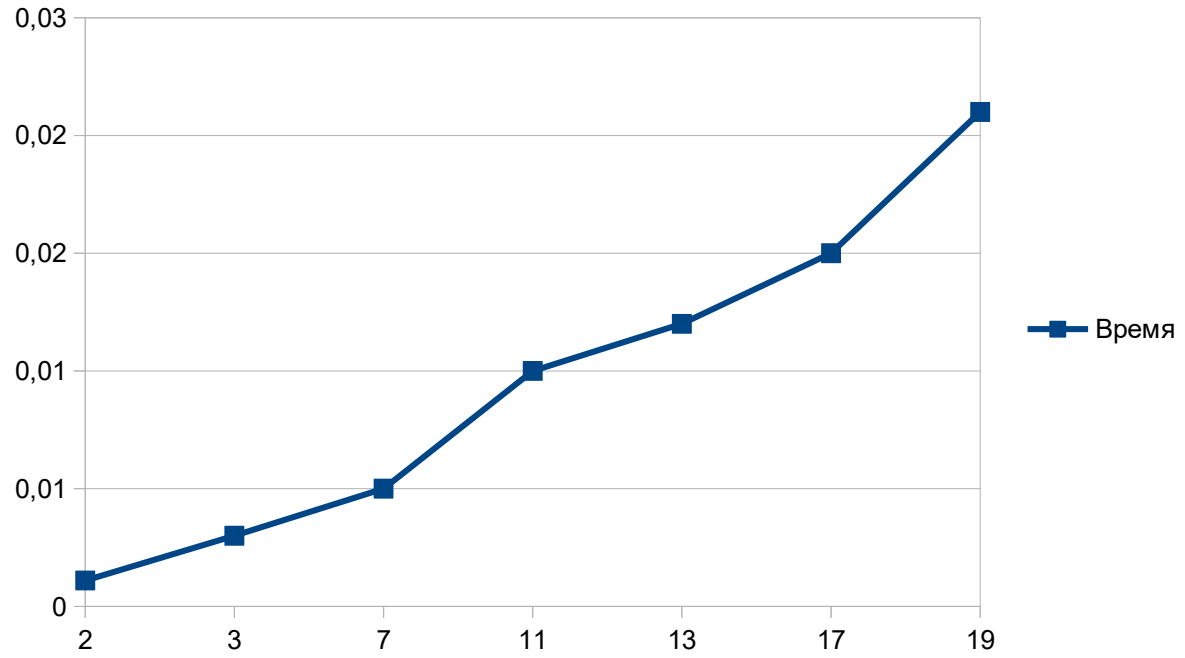


Рисунок 2 - График зависимости времени работы программы от размера квадрата

По данному графику видно, что время работы алгоритмы экспоненциально зависит от размера квадрата.

Тестирование:

Входные данные	Выходные данные
2	4 1 1 1 2 1 1 1 2 1 2 2 1 При размере квадрата 2 было затрачено времени: 0.001
5	8 1 1 3 1 4 2 4 1 2 4 3 2 3 4 1 3 5 1 4 5 1 5 5 1 При размере квадрата 5 было затрачено времени: 0.004
7	9 1 1 4 1 5 3 5 1 3 5 4 2 7 4 1 4 5 1 7 5 1 4 6 2 6 6 2 При размере квадрата 7 было затрачено времени: 0.005
23	13 1 1 12 1 13 11 13 1 11 13 12 2 15 12 5 20 12 4 12 13 1 12 14 3

	20 16 1 21 16 3 12 17 7 19 17 2 19 19 5 При размере квадрата 23 было затрачено времени: 0.056

Заключение.

В результате выполнения работы был изучен и реализован алгоритм поиска с возвратом.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;


struct Square { //Квадрат, который можно расположить на столешнице
     int size;
     int x;
     int y;
};

bool Field[50][50];   //Столешница

//Массивы квадратов:
vector <Square> StepStack;  //Описывает заполнение столешницы
vector <Square> CurrentStack; //Текущее заполнение столешницы
vector <Square> BestStack; //Лучшее заполнение столешницы (наименьшее количество квадратов)


Square DeleteSquare(const int& size) { //Функция для удаления последнего квадрата
     if (CurrentStack.size() == 0)
          return Square{ -1, -1, -1 };
     Square Last = CurrentStack.back();
     CurrentStack.pop_back();
     //cout << "Был удален квадрат: " << Last.x + 1 << ' ' << Last.y + 1 << ' ' << Last.size << '\n';
     for (int i = 0; i < Last.size; ++i) {
          for (int j = 0; j < Last.size; ++j) {
               Field[i + Last.y][j + Last.x] = 0;
               }
     }
     return Last;
}



Square FreeCell(int size) {  //Функция поиска пустой клетки на столешнице
     for (int i = 0; i < size; ++i) {
          for (int j = 0; j < size; ++j) {
               if (Field[i][j] == 0)
                    return { 0, j, i };
          }
     }
     return { -1, -1, -1 };
```

```
}


Square Place(int size, Square Alleged) { //Функция, которая по-
мещает квадрат на столешнице (использую последнюю позицию уда-
ленного квадрата)
     Square Result;
     int SizeOfRes = 1;
     if (Alleged.size > 0) { //используем информацию о последнем
удаленном квадрате
          Result = Alleged;
          Result.size--;
     }
     else {
          Result = { size - 1, 0, 0 }; //первый шаг
     }
     //Определяем размер квадрата, который будет меньше, чем
предыдущий квадрат. Также он должен помещаться на столешницу.
     while ((SizeOfRes < Result.size) && ((Result.x + SizeOfRes)
< size) && ((Result.y + SizeOfRes) < size) && (Field[Result.y]
[Result.x + SizeOfRes] == 0)) {
          SizeOfRes++;
     }
     Result.size = SizeOfRes;
     for (int i = 0; i < Result.size; ++i) { // Расположение
квадрата
          for (int j = 0; j < Result.size; ++j) {
               Field[Result.y + i][Result.x + j] = 1;
          }
     }
     return Result;
}


void Backtracking(int size) {  //Итеративный поиск с возратом
     bool back = false;
     bool full = false;
     int div = 1;
     Square Alleged{ -1, 0, 0 };
     Square ResultPlace{ 0, 0, 0 };
     for (int i = 2; i <= size; i++) { //Уменьшение размера сто-
лешницы до наименьшего простого делителя
          if (size % i == 0) {
               div = size / i;
               size = i;
          }
     }

     //cout << "Расположение первых квадратов:\n";
     //cout << "Первый квадрат: 1 1 " << (size + 1) / 2 << "\n";
     //cout << "Второй квадрат: 1 " << (size + 1) / 2 + 1 << " "
<< size - (size + 1) / 2 << "\n";
```

```
     //cout << "Третий квадрат: " << (size + 1) / 2 + 1 << " 1 "
<< size - (size + 1) / 2 << "\n";

     ResultPlace = Place(size, { (size + 1) / 2 + 1, 0, 0 }); //
Сохранение в стэк квадратов
     CurrentStack.push_back(ResultPlace);
     ResultPlace = Place(size, { size - (size + 1) / 2 + 1, 0,
(size + 1) / 2 });
     CurrentStack.push_back(ResultPlace);
     ResultPlace = Place(size, { size - (size + 1) / 2 + 1,
(size + 1) / 2, 0 });
     CurrentStack.push_back(ResultPlace);
     Alleged = FreeCell(size);
     Alleged.size = size - 1;

     //Начало цикла
     do {
          int last_size = 0;
          full = false;
          back = false;
          while (!full) {  //Пока столешница не будет полностью
заполнена
               ResultPlace = Place(size, Alleged); //Расположим
один квадрат
               CurrentStack.push_back(ResultPlace);
               if ((BestStack.size() > 0) &&
(CurrentStack.size() > 0) && (CurrentStack.size() >=
BestStack.size())) {
                    back = true;  //Если текущее расположение
хуже, то делаем шаг назад (удаляем квадрат)
                    DeleteSquare(size);
                    Alleged = DeleteSquare(size); //Квадраты
были расположены наилучшим способом
                    break;
               }
               Alleged = FreeCell(size); //Подготовка к следую-
щему размещению квадратов
               full = (Alleged.size == -1);
               Alleged.size = size;
          }
          if (!back) { //Если столешница заполнена
               if ((CurrentStack.size() < BestStack.size()) ||
(BestStack.size() == 0)) { //Сравнение текущего расположения с
лучшим расположением
                    BestStack = CurrentStack;
               }
          }

          do {
               Alleged = DeleteSquare(size); //Сделать шаг на-
зад, если размер квадрата равен 1 или 2, если столешница запол-
нена
```

```
          } while (Alleged.size == 1 || (full && Alleged.size ==
2) && CurrentStack.size() >= 3);


     } while (CurrentStack.size() >= 3);

     for (int i = 0; i < BestStack.size(); i++) {
          BestStack[i].x *= div;
          BestStack[i].y *= div;
          BestStack[i].size *= div;
     }
}



int main() {
     setlocale(LC_ALL, "Russian");
     int N;
     cout << "Введите размер столешницы:\n";
     cin >> N;
     double time = clock();

     if (N % 2 == 0) {
          cout << "\nРазмер четный, бэктрекинг не требуется.\n";
          cout << 4 << "\n";
          cout << 1 << ' ' << 1 << ' ' << N / 2 << "\n";
          cout << N / 2 + 1 << ' ' << 1 << ' ' << N / 2 << "\n";
          cout << 1 << ' ' << N / 2 + 1 << ' ' << N / 2 << "\n";
          cout << N / 2 + 1 << ' ' << N / 2 + 1 << ' ' << N / 2
<< "\n";
     }
     else {
          cout << "\nВыполнение бэктрекинга:\n";
          Backtracking(N);
          cout << BestStack.size() << "\n";
          for (auto& Res : BestStack)
               cout << Res.x + 1 << ' ' << Res.y + 1 << ' ' <<
Res.size << '\n';
     }

     cout << "\
n-------------------------------------------------------\n";
     cout << "_Исследование времени выполнения от размера квад-
рата_\n";
     cout << "При размере квадрата " << N << " было затрачено
времени: " << (clock() - time) / 1000.0 << '\n';
     return 0;
}
```