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. 9382 _____ Павлов Р.В.

Преподаватель _____ Фирсов М.А.

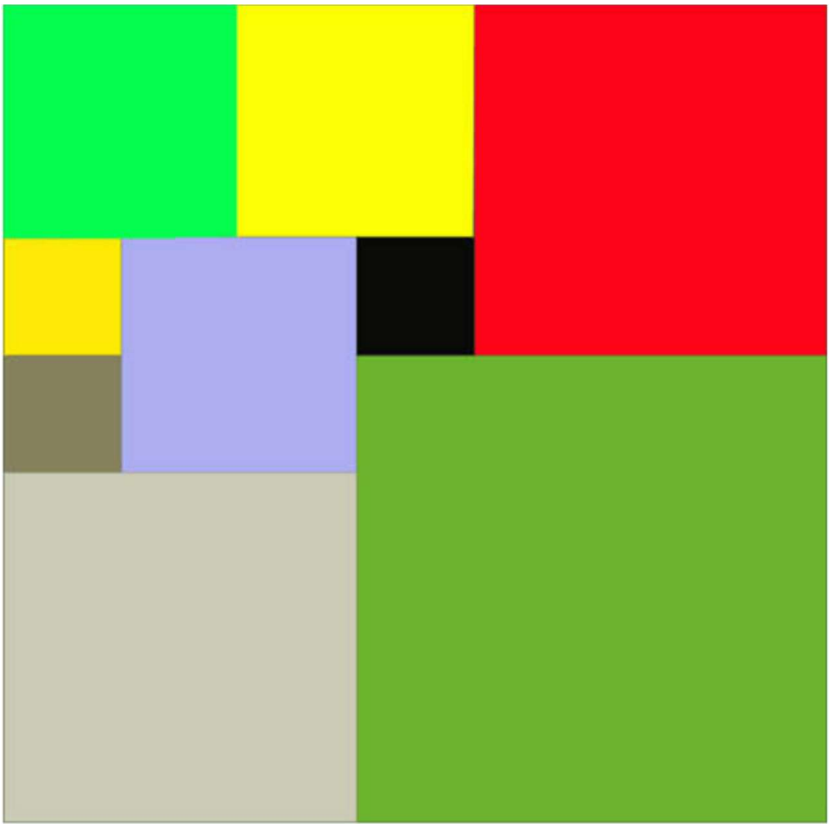
Санкт-Петербург
2021

Цель работы

Изучение и реализация алгоритма поиска с возвратом.

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

Выходные данные

Одно число , 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

1. Размещаются начальные квадраты либо сразу же записывается результат, в зависимости от входных данных. В трек заносится информация о размещённых на поле квадратах.
2. Осуществляется поиск свободного места и определение максимального размера для квадрата, верхний левый угол которого будет размещён в найденной свободной клетке.
3. Если достигнуто условие возврата (меньшее число квадратов уже было найдено либо поле заполнено), выполняется откат до последнего квадрата, размер которого при условии масштабирования не равен 1, его длина уменьшается на 1, и строятся последующие комбинации. Если поле заполнено и число квадратов меньше минимального, найденного ранее, текущее заполнение запоминается. Если условие не достигнуто, на найденную позицию помещается квадрат максимально возможного размера. Пока не произведена попытка удалить один из начальных квадратов, повторить с пункта 2.

Сложность алгоритма

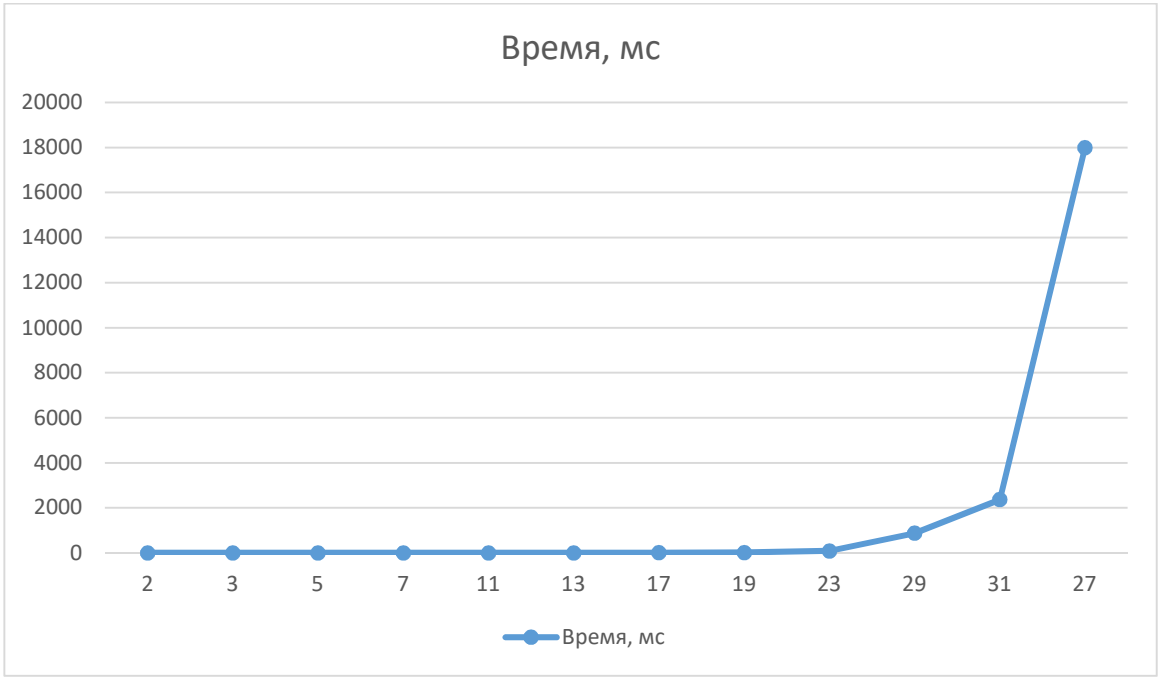
Поскольку в памяти хранится лишь двумерный массив размера N*N, а в трек может быть записано не более N*N квадратов длины 1, сложность алгоритма по памяти составляет $O(N^2)$.

Приближённо оценить сложность по операциям проблематично из-за того, что размер и размещение квадратов имеют значение, и их количество, а, следовательно, и зависимость от входных данных, трудно определить. Можно дать грубую оценку, считая за обрабатываемую область у квадратов с нечётной простой стороной ту область, которая не занята тремя начальными квадратами (в

ней всегда будет недоставать одной клетки до полного квадрата). Предположим, что в каждой клетке можно разместить n – 1 различных квадратов, где n = N/2 + 1 (квадрат со стороной n нельзя будет поместить из-за той самой недостающей клетки), а для каждого такого квадрата будет существовать ещё столько же размещений в любой незанятой клетке (не берётся во внимание остаток свободного места и положение квадратов, но на самом деле их всегда будет меньше). Всего свободных клеток $n^2 - 1$, значит приблизительная сложность по операциям составит $O((n - 1)^{n*n - 1})$ или
$O((N/2)^{(N/2 + 1)^2 - 1})$, что в итоге может быть записано как
$O(N^{N^2})$, где N – наименьший простой делитель исходного числа.

Если число чётное, то сложность составляет O(1), т.к. оптимальный вариант – разделить квадрат на 4 равных части, каким бы большим ни было число.

При исследовании зависимости времени алгоритма от объёма входных данных на основе измерений была построена следующая диаграмма:



Из графика видно, что сложность алгоритма по операциям имеет экспоненциальный порядок, хотя для первых простых чисел эксперимент дал небольшие значения временных затрат. Смысла проверять числа, кратные

простым, нет, т.к. они масштабируются и имеют сложность наименьшего простого делителя.

Оптимизации алгоритма

- Для квадратов с чётной стороной осуществляется разбиение на 4 равных квадрата.
- Для квадратов с нечётной простой стороной размещаются три начальных квадрата в трёх углах. Один из них имеет сторону N/2 + 1, другие два – N/2.
- При наличии у N простых делителей, меньших него, выбирается минимальный из них d, решение для N сводится к решению для d.
- При достижении количества квадратов, которое уже помечено, как минимальное, осуществляется возврат, даже если квадрат не заполнен.

Описание функций и СД

- **struct Square** {

int x, y;
int side;
};

- структура, описывающая квадрат с верхним левым углом в x, y и со стороной side

- **class SquareTable**

- класс, описывающий квадратное поле

Поля класса:

- **int side** – сторона поля
- **std::vector<Square> track** – трек для записи размещённых на поле квадратов
- **std::vector<Square> bestTrack** – трек для хранения наилучшего размещения
- **std::array<std::array<int, 40>, 40> surface** – двумерный массив поля
- **std::array<std::array<int, 40>, 40> bestSurface** – двумерный массив наилучшего размещения

Методы класса:

- **void printTrack(const int& scale)** – вывод записанных в трек элементов на экран; аргументы: scale – масштаб для вывода
- **void printSurface(const int& scale)** – вывод лучшего записанного размещения; аргументы: scale – масштаб для вывода

- **Square priorSquare() const** – получение координат и размера квадрата для размещения; возвращаемое значение – структура типа Square
- **void writeToTrack(const Square& pos)** – запись квадрата в трек; аргументы: pos – структура типа Square
- **Square getFromTrack()** – получение элемента из трека; возвращаемое значение – структура типа Square
- **void display(const int& scale) const** – вывод текущего размещения на экран; аргументы: scale – масштаб для вывода

- **inline int scaled(const int& n)** – возвращает масштабированное число, являющееся простым; аргументы: n – исходное число
- **void findLeastNumberOfSquares(const int& globalSide)** – основная функция, реализующая алгоритм бэктрекинга; аргументы: globalSide – исходная сторона квадратного поля
- **int main()** – пользовательский ввод, вывод времени работы

Тестирование

Результаты тестирования представлены в таблице 1.

Таблица 1

Входные данные	**Выходные данные**
2	4 1 1 1 1 2 1 2 1 1 2 2 1
14	4 1 1 7 1 8 7 8 1 7 8 8 7
3	6 1 1 2 1 3 1 3 1 1 2 3 1 3 2 1 3 3 1
5	8 1 1 3 1 4 2 4 1 2 3 4 2 4 3 1 5 3 1 5 4 1 5 5 1
7	9 1 1 4 1 5 3 5 1 3 4 5 2 4 7 1 5 4 1 5 7 1 6 4 2 6 6 2

21	6 1 1 14 1 15 7 15 1 7 8 15 7 15 8 7 15 15 7
29	14 1 1 15 1 16 14 16 1 14 15 16 2 15 18 5 15 23 7 16 15 1 17 15 3 20 15 3 20 18 3 20 21 2 22 21 1 22 22 8 23 15 7
37	15 1 1 19 1 20 18 20 1 18 19 20 2 19 22 5 19 27 11 20 19 1 21 19 3 24 19 8 30 27 3 30 30 8 32 19 6 32 25 1 32 26 1 33 25 5

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

```
#include <iostream>
#include <array>
#include <vector>
#include <cmath>
#include <climits>
#include <ctime>

#define OUTPUT

struct Square {
    int x, y;
    int side;
};

class SquareTable {
private:
    int side;
public:
    std::vector<Square> track;
    std::vector<Square> bestTrack;
    std::array<std::array<int, 40>, 40> surface = { 0 };
    std::array<std::array<int, 40>, 40> bestSurface;

    explicit SquareTable(const int n) {
        side = n;
    }

    void printTrack(const int& scale) {
        for (auto& i : bestTrack) {
            std::cout << i.x * scale + 1 << " " << i.y * scale + 1 << " " << i.side * scale
<< "\n";
        }
    }

    void printSurface(const int& scale) {
        for (int i = 0; i < side * scale; ++i) {
            for (int j = 0; j < side * scale; ++j) {
                std::cout.width(2);
                std::cout << bestSurface[i/scale][j/scale] << " ";
            }
            std::cout << "\n\n";
        }
    }

    Square priorSquare() const {
        Square pos{ -1, -1, 0 };
        for (int i = 0; i < side; ++i) {
            bool found = false;
            for (int j = 0; j < side; ++j) {
                if (surface.at(i).at(j) == 0) {
                    pos.x = i;
                    pos.y = j;
                    found = true;
                    break;
                }
            }
            if (found) {
                break;
            }
        }
```

```
        do {
            ++pos.side;
            if ((pos.x == -1) || (pos.x + pos.side - 1 >= side) || (pos.y + pos.side - 1 >=
side)) {
                pos.side = -pos.side;
            }
            for (int l = pos.y; l < pos.y + pos.side; ++l) {
                if (surface[pos.x][l] != 0) {
                    pos.side = -pos.side;
                }
            }
        } while (pos.side > 0);
        pos.side = - (++pos.side);
        return pos;
    }

    void writeToTrack(const Square& pos) {
        track.push_back(pos);
        for (int i = 0; i < pos.side; ++i) {
            for (int j = 0; j < pos.side; ++j) {
                surface.at(pos.x + i).at(pos.y + j) = track.size();
            }
        }
    }

    Square getFromTrack() {
        if (track.size() < 1) {
            return Square{ 0, 0, 0 };
        }
        Square pos = track.back();

        for (int i = 0; i < pos.side; ++i) {
            for (int j = 0; j < pos.side; ++j) {
                surface.at(pos.x + i).at(pos.y + j) = 0;
            }
        }

        track.pop_back();
        return pos;
    }

    void display(const int& scale) const {
        for (int i = 0; i < side * scale; ++i) {
            for (int j = 0; j < side * scale; ++j) {
                std::cout.width(2);
                std::cout << surface[i / scale][j / scale] << " ";
            }
            std::cout << "\n\n";
        }
    }
};

inline int scaled(const int& n) {
    for (int i = 2; i <= sqrt(n); ++i) {
        if (!(n % i)) {
            return i;
        }
    }
    return n;
}

void findLeastNumberOfSquares(const int& globalSide) {
    int scaledSide = scaled(globalSide);
```

```
    SquareTable* table = new SquareTable(scaledSide);
    int leastNum = INT_MAX;
    Square squareFragment;

    if (scaledSide == 2) {
        leastNum = 4;
#ifdef OUTPUT
        std::cout << "Число чётное, выполнить разбиение столешницы на 4 квадрата со стороной
" << globalSide / 2 << ".\n\n";
#endif
        squareFragment = { 0, 0, scaledSide / 2 };
        table->writeToTrack(squareFragment);
        squareFragment = { 0, scaledSide / 2, scaledSide / 2 };
        table->writeToTrack(squareFragment);
        squareFragment = { scaledSide / 2, 0, scaledSide / 2 };
        table->writeToTrack(squareFragment);
        squareFragment = { scaledSide / 2, scaledSide / 2, scaledSide / 2 };
        table->writeToTrack(squareFragment);
        table->bestTrack = table->track;
        table->bestSurface = table->surface;
    }
    else {
        int curNum = 3;
#ifdef OUTPUT
        std::cout << "Обнаружено простое нечётное число.\nЗапись трёх начальных квадра-
тов.\n\n";
#endif
        squareFragment = { 0, 0, scaledSide / 2 + 1 };
        table->writeToTrack(squareFragment);

        squareFragment = { 0, scaledSide / 2 + 1, scaledSide / 2 };
        table->writeToTrack(squareFragment);

        squareFragment = { scaledSide / 2 + 1, 0, scaledSide / 2 };
        table->writeToTrack(squareFragment);
#ifdef OUTPUT
        table->bestSurface = table->surface;
        std::cout << "\n\nНачальный вид столешницы:\n\n";
        table->printSurface(globalSide / scaledSide);
#endif

        while (true) {
            squareFragment = table->priorSquare();
            if (squareFragment.x < 0 || curNum >= leastNum) {

                if (curNum < leastNum) {
#ifdef OUTPUT
                    std::cout << "\nОбнаружено лучшее заполнение.\n\nПредыдущий набор (" <<
                        (leastNum==INT_MAX ? 0 : leastNum) << " квадратов):\n";
                    table->printTrack(1);
#endif
                    table->bestTrack = table->track;
                    leastNum = curNum;
#ifdef OUTPUT
                    std::cout << "\nНовый набор (" << leastNum << " квадратов):\n";
                    table->printTrack(1);
                    std::cout << "\n\nЛучшее заполнение столешницы:\n\n\tДо:\n\n";
                    table->printSurface(globalSide / scaledSide);
#endif
                    table->bestSurface = table->surface;
#ifdef OUTPUT
                    std::cout << "\n\tПосле:\n";
                    table->printSurface(globalSide / scaledSide);
#endif
```

```
                }
#ifdef OUTPUT
                else {
                    std::cout << "\nТекущее число квадратов достигло минимального.\nВыполнить возврат.\n";
                }
                std::cout << "\nИзъятие из трека всех квадратов вплоть до первого с длиной, отличной от 1 (с учётом масштаба)...\n";
#endif
                do {
                    squareFragment = table->getFromTrack();
                    --curNum;
                } while (squareFragment.side == 1);
#ifdef OUTPUT
                std::cout << "Столешница после возврата:\n\n";
                table->display(globalSide / scaledSide);
#endif
                if (++curNum < 4) {
#ifdef OUTPUT
                    std::cout << "\nПопытка удалить один начальный квадрат или более.\nПрекратить поиск.\n\n";
#endif
                    break;
                }
                --squareFragment.side;
                table->writeToTrack(squareFragment);
#ifdef OUTPUT
                std::cout << "\nДекремент длины последнего изъятого квадрата и запись на исходные координаты.\nТекущий вид столешницы:\n\n";
                table->display(globalSide / scaledSide);
#endif
            }
            else {
                ++curNum;
                table->writeToTrack(squareFragment);
#ifdef OUTPUT
                std::cout << "\nЗапись квадрата максимально допустимого размера с приоритетом (верх, лево). Текущий вид столешницы:\n\n";
                table->display(globalSide / scaledSide);
#endif
            }
        }
    }
    std::cout << "\nВ итоге " << leastNum << " квадратов. Финальный набор:\n";
    table->printTrack(globalSide / scaledSide);
    std::cout << "\n\nНаилучшее заполнение столешницы:\n\n";
    table->printSurface(globalSide / scaledSide);
}

int main()
{
    setlocale(LC_ALL, "Rus");
    int n;
    std::cout << "Пожалуйста, введите длину стороны квадрата: ";
    std::cin >> n;
    clock_t begin = clock();
    findLeastNumberOfSquares(n);
    clock_t end = clock();
    std::cout << "Время выполнения: " << end - begin << "\n";
    system("pause");
    return 0;
}
```