МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный

электротехнический университет

«ЛЭТИ» им. В.И. Ульянова (Ленина)

Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Построение и анализ алгоритмов»

Тема: Поиск с возвратом.

Студент гр. 8382 _____ Терехов А.Е.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

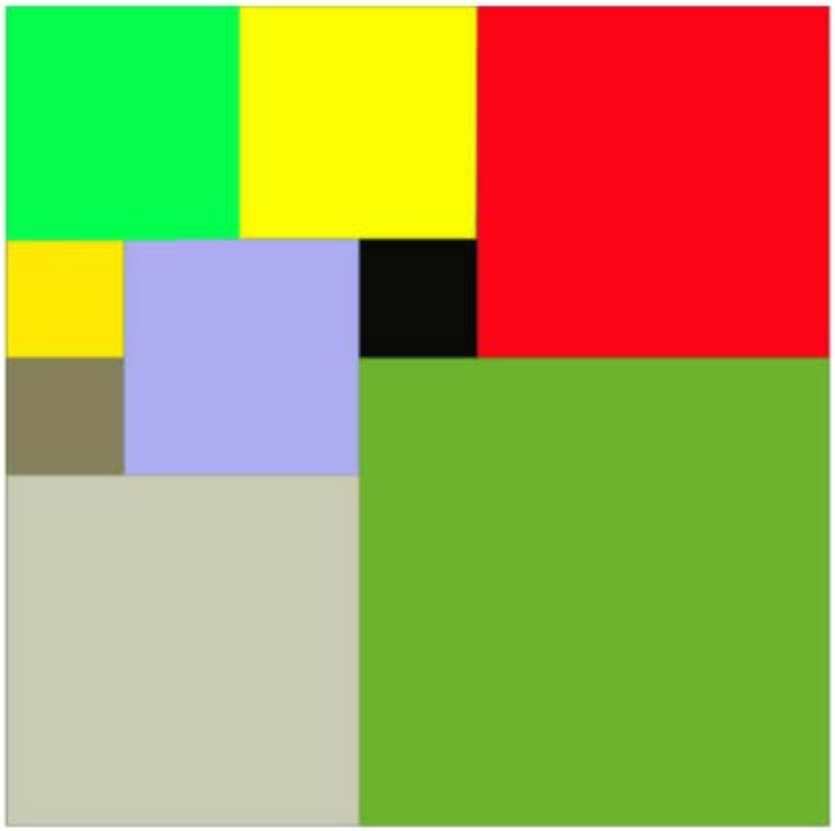
Освоить метод поиска с возвратом.

Задание.

Вариант 3Р. Рекурсивный бэктрекинг. Исследование зависимости количества операций от размера квадрата.

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 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).

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 структур.

С исходным кодом можно ознакомиться в приложении А.

В программе для хранения квадрата была реализована структура Square с полями corner, содержащим информацию о расположении левого верхнего угла и a – длина стороны. Для структуры был написан метод, определяющий пересекаются ли квадраты, а также переопределены некоторые операторы.

Для хранения массива квадратов был написан класс Squares, содержащий информацию о стороне делимого квадрата в поле a, и массив вписанных

квадратов – arr. В классе были реализованы методы для копирования массива и для добавления и удаления квадратов.

Для хранения минимального количества и соответствующей композиции были использованы глобальные переменные, также они используются для счетчиков операций.

Описание алгоритма.

Рекурсивная функция backtracking осуществляет поиск минимального количества квадратов с максимальной длиной стороны n-1, которыми можно замостить большой квадрат со стороной N. Функция принимает указатель на массив квадратов, на минимальный, размер добавляемого квадрата, и площадь, которая уже замощена. При входе в функцию первым делом, если текущее количество квадратов на один меньше минимума проверяется возможность замостить его площадью оставшуюся не замощенную площадь. Затем производится единичная вставка квадрата с переданной в функцию стороной в самое левое верхнее доступное место. Если в какой-то момент минимум оказался меньше, чем текущее количество квадратов, то очевидно дальше проверять эту композицию смысла нет. Поэтому такая проверка тоже имеет право на существование. В ней происходит удаление последнего добавленного квадрата и подъем на один уровень рекурсии. После этого происходит обновление минимума, в котором также удаляется последний квадрат и возврат на один уровень рекурсии. Затем в цикле происходит серия рекурсивных вызовов, с уменьшением стороны вставляемого квадрата.

В программе по ходу решения выводятся в консоль вставляемые и удаляемые квадраты и обновления минимума.

Были использованы следующие оптимизации:

1. Имеет смысл запускать рекурсивный поиск с возвратом только для квадратов с нечетной простой стороной, поэтому в первую очередь раскладываем длину стороны на множители – простое число, с которым будем работать, и коэффициент сжатия.

2. Очевидно, что для квадратов с четной длиной стороны ответ будет одинаков – минимальная конфигурация имеет длину 4, а квадраты расположены по углам.

3. В соответствие с оптимизацией 2, то же самое можно сказать и про длину, делящуюся на 3.

4. Эмпирическим методом было выяснено, что во всех остальных случаях в одном углу всегда стоит квадрат со стороной (n+1)//2, а два смежных с ним имеют длину стороны n//2. Доказать это можно так: увеличить ни один из квадратов нельзя, а если попробовать уменьшать какой-либо квадрат, то образуется интервал между квадратами, который необходимо заполнить относительно большим числом маленьких квадратов, что не может дать выигрыша в оптимальности.

Исследование зависимости количества операций от размера квадрата.

Как уже было сказано выше имеет смысл рассматривать только квадраты с простой стороной, так как если сторону задает составное число, то можно вычислить квадрирование для квадрата со стороной равной самому маленькому сомножителю и при выводе умножить координаты и стороны маленьких квадратов на оставшиеся сомножители – коэффициент сжатия.

Результаты исследования представлены в таблице 1.

Таблица 1. Исследование зависимости количества операций от размера квадрата. (начало)

Длина стороны	Минимальное количество квадратов	Квадраты	Количество добавлений квадратов	Количество удалений квадратов	Количество обновлений минимумов	Количество сравнений	Суммарное количество операций
2	4	1 1 1 2 1 1 1 2 1 2 2 1	4	0	1	0	5
3	6	1 1 2 3 1 1 3 2 1 1 3 1 3 3 1 2 3 1	6	0	1	0	7
5	8	1 1 3 4 1 2 1 4 2 3 4 2 4 3 1 5 3 1 5 4 1 5 5 1	21	4	1	227	253
7	9	1 1 4 5 1 3 1 5 3 4 5 2 4 7 1 5 4 1 5 7 1 6 4 2 6 6 2	58	18	2	1238	1316
11	11	1 1 6 7 1 5 1 7 5 6 7 3 6 10 2 7 6 1 8 6 1 8 10 1 8 11 1 9 6 3 9 9 3	711	330	3	32065	33109
13	11	1 1 7 8 1 6 1 8 6 7 8 2 7 10 4 8 7 1 9 7 3 11 10 1 11 11 3 12 7 2 12 9 2	1609	766	4	97409	99798

Таблица 1. Исследование зависимости количества операций от размера квадрата. (конец)

Длина стороны	Минимальное количество квадратов	Квадраты	Количество добавлений квадратов	Количество удалений квадратов	Количество обновлений минимумов	Количество сравнений	Суммарное количество операций
17	12	1 1 9 10 1 8 1 10 8 9 10 2 9 12 4 9 16 2 10 9 1 11 9 3 11 16 2 13 12 1 13 13 5 14 9 4	9967	5341	5	924601	939914
19	13	1 1 10 11 1 9 1 11 9 10 11 3 10 14 6 11 10 1 12 10 1 13 10 4 16 14 1 16 15 1 16 16 4 17 10 3 17 13 3	28269	16027	5	3087806	3132107
23	13	1 1 12 13 1 11 1 13 11 12 13 2 12 15 5 12 20 4 13 12 1 14 12 3 16 20 1 16 21 3 17 12 7 17 19 2 19 19 5	105693	60612	11	16425330	16591646
29	14	1 1 15 16 1 14 1 16 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	733272	427089	8	176977526	178137895

На рисунке 1 представлен график зависимости количества операций от длины стороны квадрата.

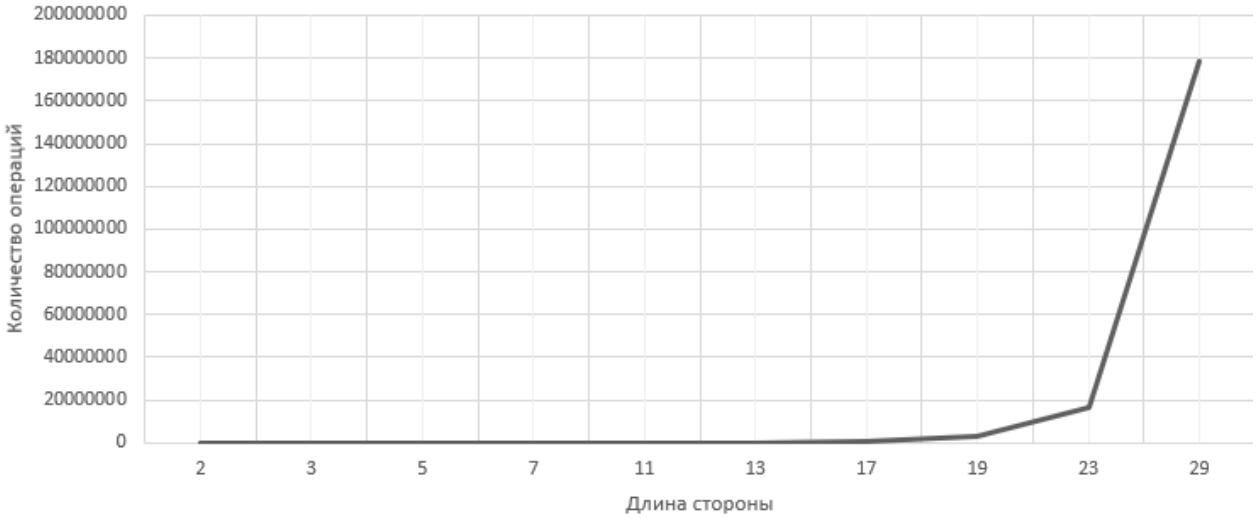


Рис. 1. Зависимость количества операций от длины стороны квадрата.

Из полученных результатов можно сделать вывод, что сложность данного алгоритма для простых значений схожа с экспоненциальной, но для четных и делящихся на 3 алгоритм работает за константу, если быть точнее, то константа умноженная на 2 в степени N. По памяти программа не столь требовательна, так как размер структуры не превышает 32 байта, прибавив к ним 2 целых числа размера 8 байт, получим 48 байт. Максимальная глубина рекурсии равна N. В итоге получим примерно 48N байт требуемой памяти.

Тестирование.

С примером работы программы при n = 7 можно ознакомиться в приложении Б.

Вывод.

В ходе работы был реализован рекурсивный алгоритм, находящий минимальное количество маленьких квадратов необходимых для квадрирования большого квадрата. Алгоритм был оптимизирован настолько, что для стороны квадрата меньше 30 можно получить ответ за разумное время. Сложность данного алгоритма – экспоненциальная. Программа потребляет примерно 48N байт.

ПРИЛОЖЕНИЕ А

Исходный код программы.

```
#include <iostream>
#include <vector>
#include <algorithm>

using std::cout;
using std::cin;
using std::endl;

const bool temp_res = true;
struct Dot {
    int x, y;

    Dot(int x, int y) : x{x}, y{y} {}
};

struct Square {
    Dot corner;
    int a;

    Square(int x, int y, int a) : corner{Dot(x, y)}, a{(a > 1) ? a : 1} {}

    Square(Square const &s) : corner{Dot(s.corner.x, s.corner.y)}, a{s.a} {}

    bool isCross(const Square &square) {
        return !(corner.x + a <= square.corner.x ||
                 corner.x >= square.corner.x + square.a ||
                 corner.y + a <= square.corner.y ||
                 corner.y >= square.corner.y + square.a);
    }

    friend bool operator==(const Square &s1, const Square &s2) {
        return s1.corner.x == s2.corner.x && s1.corner.y == s2.corner.y && s1.a
== s2.a;
    }

    friend std::ostream &operator<<(std::ostream &out, const Square &s) {
        out << s.corner.x + 1 << " " << s.corner.y + 1 << " " << s.a;
        return out;
    }

    friend Square operator*(const Square &s, int mul) {
        return Square(s.corner.x * mul, s.corner.y * mul, s.a * mul);
    }
};

class Squares {
    int a;
    std::vector<Square> arr;
public:
    Squares() : a{1} {}

    Squares(Squares &sqs) : a{sqs.a} {
        for (const auto &s : sqs.arr) {
            arr.push_back(s);
        }
    }

    void setA(int size) {
        a = size;
```

```
}

explicit Squares(int a) : a{(a > 1) ? a : 1} {}

void print(int k) {
    cout << arr.size() << endl;
    for (const auto &s : arr) {
        cout << s * k << endl;
    }
}

int getA() const {
    return a;
}

int length() const {
    return arr.size();
}

bool canAdd(Square square) {
    if (square.corner.x + square.a > a ||
        square.corner.x < 0 ||
        square.corner.y + square.a > a ||
        square.corner.y < 0)
        return false;
    for (const auto &s : arr) {
        if (square.isCross(s)) {
            return false;
        }
    }
    return true;
}

int getSquareSize(int x, int y) {
    for (auto s: arr) {
        if (s.isCross({x, y, 1}))
            return s.a;
    }
    return 0;
}

Squares &addSquare(const Square &square) {
    if (canAdd(square))
        arr.push_back(square);
    return *this;
}

std::vector<Square>::iterator begin() {
    return arr.begin();
}

std::vector<Square>::iterator end() {
    return arr.end();
}

Square delSquare() {
    Square s = *(arr.end() - 1);
    arr.pop_back();
    return s;
}

int getArea() {
    return a * a;
```

```
    }

    void assign(Squares sqs) {
        arr.assign(sqs.begin(), sqs.end());
    }
};

Squares minSqs;
int min;
int countAdd = 0;
int countDel = 0;
int countCng = 0;
int countCmp = 0;
int areaMultiplier = 1;

void backtracking(Squares sqs, int size, int area) {
    //выход если невозможно одним квадратом со стороной size замостить
оставшуюся площадь
    if (sqs.length() == min - 1 && sqs.getArea() - area > size * size) return;
    // переменная для контроля одноразовой вставки
    bool added = false;
    int i = 0;
    while (i < sqs.getA() && !added) {
        int j = 0;
        while (j < sqs.getA() && !added) {
            ++countCmp;
            if (sqs.canAdd({i, j, 1})) {
                ++countCmp;
                if (sqs.canAdd({i, j, size})) {
                    added = true;
                    sqs.addSquare({i, j, size});
                    if (temp_res)
                        cout << "Add square " << i << "\t" << j << "\t" << size
<< endl;
                    ++countAdd;
                } else return;
            } else j += sqs.getSquareSize(i, j); // перескакиваем текущий
квадрат
        }
        ++i;
    }
// выход если исчерпали лимит квадратов и удаление последнего квадрата
    if (sqs.length() + 1 == min) {
        Square s = sqs.delSquare();
        if (temp_res)
            cout << "Del square " << s.corner.x << "\t" << s.corner.y << "\t" <<
s.a << endl;
        ++countDel;
        return;
    }
// если большой квадрат заполнен и можно улучшить минимальность то обновляем
минимум
    if (sqs.getArea() - area == size * size && sqs.length() + 1 < min) {
        min = sqs.length() + 1;
        minSqs.assign(sqs);
        countCng++;
        if (temp_res){
        cout << endl << "New minimal" << endl ;
        minSqs.print(areaMultiplier);
        cout << endl;
        }
        Square s = sqs.delSquare();
        if (temp_res)
```

```
            cout << "Del square " << s.corner.x << "\t" << s.corner.y << "\t" <<
s.a << endl;
        ++countDel;
        return;
    }
// рекурсивный спуск с уменьшением стороны вставляемого квадрата
    for (int sz = sqs.getA() / 2; sz >= 1; sz--) {
        if (sz * sz <= sqs.getArea() - area)
            backtracking(sqs, sz, area + size * size);
    }
}

int main() {
    std::vector<int> primes = {29, 23, 19, 17, 13, 11, 7, 5, 3, 2};
    int n = 0;
    while (n < 2 || n > 30) {
        cout << "N (2<=N<=30): ";
        cin >> n;
    }
    int area = 0;

// для оптимизации уменьшаем квадрат до квадрата со стороной равной простому
числу, сохраняя коэффициент сжатия
    for (auto i : primes) {
        while (n % i == 0) {
            n /= i;
            areaMultiplier *= i;
            if (primes.end() != std::find(primes.begin(), primes.end(), n)) {
                break;
            }
        }
        if (primes.end() != std::find(primes.begin(), primes.end(), n)) {
            break;
        }
    }
    Squares squares(n);
    minSqs.setA(n);
    min = 2 * n;
// простые случаи рассмотрены без использования рекурсии
    if (n == 2) {
        minSqs.addSquare({0, 0, 1});
        minSqs.addSquare({1, 0, 1});
        minSqs.addSquare({0, 1, 1});
        minSqs.addSquare({1, 1, 1});
        countCng = 1;
        countAdd = 4;
    } else if (n == 3) {
        minSqs.addSquare({0, 0, 2});
        minSqs.addSquare({2, 0, 1});
        minSqs.addSquare({2, 1, 1});
        minSqs.addSquare({0, 2, 1});
        minSqs.addSquare({2, 2, 1});
        minSqs.addSquare({1, 2, 1});
        countCng = 1;
        countAdd = 6;
    }
// для больших нечетных N в углу всегда лежит квадрат со стороной (N + 1) / 2, а
правее и ниже него квадраты со сторонами N / 2 (получено эмпирическим методом)
    else {
        squares.addSquare({0, 0, (n + 1) / 2});
        minSqs.addSquare({0, 0, (n + 1) / 2});
        area += ((n + 1) / 2) * ((n + 1) / 2);
        squares.addSquare({(n + 1) / 2, 0, n / 2});
```

```
        minSqs.addSquare({(n + 1) / 2, 0, n / 2});
        area += (n / 2) * (n / 2);
        squares.addSquare({0, (n + 1) / 2, n / 2});
        minSqs.addSquare({0, (n + 1) / 2, n / 2});
        area += (n / 2) * (n / 2);
        countAdd = 3 + 3;
// запуск рекурсивной серии попыток в порядке уменьшения стороны вставляемого
квадрата
        for (int i = n / 2; i >= 1; i--) {
            backtracking(squares, i, area);
        }
    }
    cout << endl << "Answer!" << endl;
    minSqs.print(areaMultiplier);
    cout << endl;
    cout << "Count Adds: " << countAdd << endl;
    cout << "Count Dels: " << countDel << endl;
    cout << "Count Change: " << countCng << endl;
    cout << "Count Comp: " << countCmp << endl;
    cout << "Total: " << countAdd + countDel + countCng + countCmp << endl;
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

Тестирование при n=7

```
Add square 3    4       3
Add square 4    3       1
Add square 5    3       1
Add square 6    3       1
Add square 6    4       1
Add square 6    5       1
Add square 6    6       1
New minimal
10
1 1 4
5 1 3
1 5 3
4 5 3
5 4 1
6 4 1
7 4 1
7 5 1
7 6 1
7 7 1
Del square 6    6       1
Add square 3    4       2
Add square 3    6       1
Add square 4    3       1
Add square 4    6       1
Add square 5    3       2
Add square 5    5       2
New minimal
9
1 1 4
5 1 3
1 5 3
4 5 2
4 7 1
5 4 1
5 7 1
6 4 2
6 6 2
Del square 5    5       2
Add square 5    5       1
Del square 5    5       1
Add square 5    3       1
Add square 5    4       2
Del square 5    4       2
Add square 5    4       1
Del square 5    4       1
Add square 3    4       1
Add square 3    5       2
Add square 4    3       2
Add square 5    5       2
Add square 6    3       1
Add square 6    4       1
Del square 6    4       1
Add square 5    5       1
Add square 5    6       1
Add square 6    3       1
Del square 6    3       1
Add square 4    3       1
Add square 4    4       1
Add square 5    3       2
```

```
Add square 5    5       2
Del square 5    5       2
Add square 5    5       1
Del square 5    5       1
Add square 5    3       1
Add square 5    4       2
Del square 5    4       2
Add square 5    4       1
Del square 5    4       1
Add square 3    5       1
Add square 3    6       1
Add square 4    3       3
Add square 4    6       1
Add square 5    6       1
Del square 5    6       1
Add square 4    3       2
Add square 4    5       2
Add square 6    3       1
Del square 6    3       1
Add square 4    5       1
Add square 4    6       1
Del square 4    6       1
Add square 4    3       1
Add square 4    4       3
Add square 5    3       1
Del square 5    3       1
Add square 4    4       2
Add square 4    6       1
Del square 4    6       1
Add square 4    4       1
Add square 4    5       2
Del square 4    5       2
Add square 4    5       1
Del square 4    5       1
9
1 1 4
5 1 3
1 5 3
4 5 2
4 7 1
5 4 1
5 7 1
6 4 2
6 6 2
Count Adds: 58
Count Dels: 18
Count Change: 2
Total: 78
```