# 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. 1303 |  | Герасименко Я.Д. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург 2022

# Цель работы.

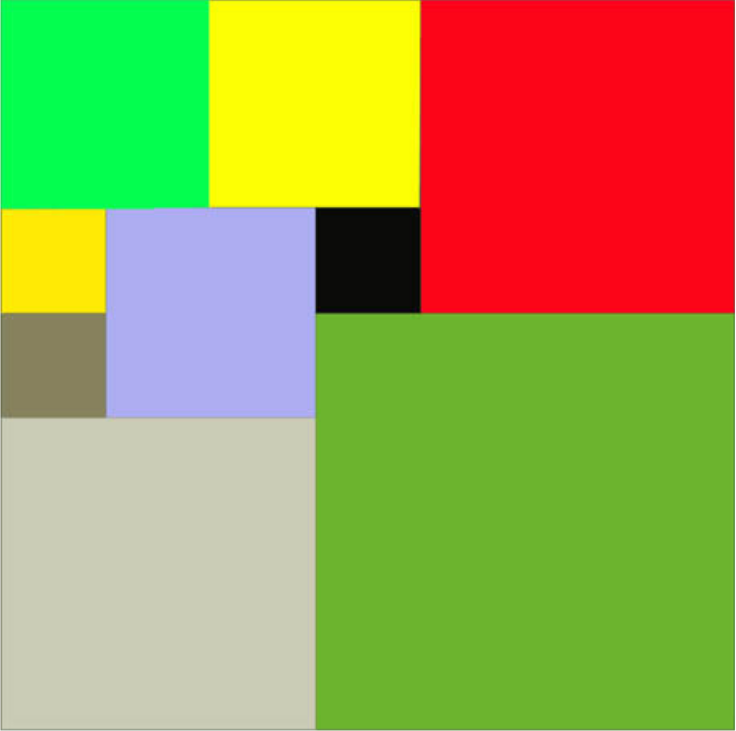
Изучить принцип работы бэктрекинга - алгоритма поиска с возвратом.

# Основные теоретические положения.

Поиск с возвратом, бэктрекинг — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

# 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, каждая из которых должна содержать три целых числа x, ,y и w, задающие координаты левого верхнего угла (1 ≤ x, y ≤ 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

**Вар. 3р**. Рекурсивный бэктрекинг. Исследование кол-ва операций о(n) от размера квадрата.

**Выполнение работы.**

Для решения задачи использован алгоритм рекурсивного бэктрекинга. При добавлении нового квадрата сохраняются его нижний левый угол и длина стороны. Функция *void backtrack(vector<Border> &map, int occupancySquare, int bordlen,int xLeftOccupancy, int yLeftOccupancy, int n, int &bestcount, vector<Border> &bestarry,int &counterOpertion)* находит самую верхнюю точку слева, которая еще не попала в квадраты. Для быстрого поиска этой точки функции передаются координаты, где уже заполнена столешница. Затем ищется максимальный радиус квадрата, нижний левый угол которого находится в этой точке, и функция void backtrack(...)

вызывается для каждого такого возможного квадрата, радиус которого меньше или равен найденному. Затем суммарная площадь квадратов сравнивается с площадью столешницы: если значения равны, а количество квадратов меньше, чем у уже найденного минимального решения, то количество квадратов записывается в переменную *bestcount.* Функция v*oid backtrack(…)* не вызывается, если длина массива на текущем шаге не меньше минимального решения, полученного на этот момент. Решение хранится в векторе v*ector<Border> bestSolve* в виде набора заполненных квадратов, где для каждого квадрата сохраняются его нижний левый угол и длина стороны.

Для оптимизации в решении участвуют только квадраты с размером,

кратным наибольшему делителю n. Также на столешнице сразу располагаются квадратные доски со стороной *(n+1)/2* в точку *{0, 0}* и два квадрата размером *n/2* в точки *{0, (n+1)/2} и {(n+1)/2, 0}.*

Был создан класс *Border*, которая хранить в себе координата верхнего левого угла обрезка *(x,y)* и размер стороны обрезка *(w)*.

*bool overlaySearch(const vector<Border> &vec, int x, int y) -bool overlaySearch(const vector<Border> &vec, int x, int y)* — функция проверяющая не принадлежит ли данной координате какой либо обрезок квадрата.

*vector<Border> initStartSqOptimization(int n)* — Функция возвращающая вектор, в котором хранятся данные исходя из нашей оптимизации.

*int maxDivision(int n, int &sqSize)* — Функция возвращающая наибольший делитель

*int OccupancySq;* — суммарная площадь размещенных досок.

*int boardLen;* — длина лучшего вектора *bestSolve.*

*int xLeftOccupancy, yLeftOccupancy;* — координаты, левее/выше которой столешница уже заполнена.

# Исследование сложности.

В данном конкретном случае, сложность алгоритма *backtracking* может быть оценена как *O(к^n)*, где *k* - константа, *n* - размер входных данных, т.е. размер доски, на которой нужно разместить квадраты.

Это связано с тем, что для каждой позиции *(x, y)* на доске мы рассматриваем до *k* возможных сторон квадрата ( *k* в моем случае это min(n-x, n-y)), что даёт нам до *O(к^n)* или  *O((min(n-x, n-y))^n)* возможных комбинаций.

Таким образом, сложность алгоритма может расти экспоненциально с увеличением размера входных данных, что делает его неэффективным для больших простых значений *n.*

# Тестирование.

|  |  |  |
| --- | --- | --- |
| Номер Теста. | Входные данные | Выходные данные |
| 1 | 2 | 4  1 1 1  1 2 1  2 1 1  2 2 1 |
| 2 | 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 |
| 3 | 9 | 6  1 1 6  1 7 3  7 1 3  4 7 3  7 4 3  7 7 3 |
| 4 | 13 | 11  1 1 7  1 8 6  8 1 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 |
| 5 | 20 | 4  1 1 10  1 11 10  11 1 10  11 11 10 |
| 6 | 23 | 13  1 1 12  1 13 11  13 1 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 |
| 7 | 25 | 1 1 15  1 16 10  16 1 10  11 16 10  16 11 5  21 11 5  21 16 5  21 21 5 |
| 8 | 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 |
| 9 | 31 | 15  1 1 16  1 17 15  17 1 15  16 17 3  16 20 6  16 26 6  17 16 1  18 16 1  19 16 4  22 20 1  22 21 1  22 22 10  23 16 6  29 16 3  29 19 3 |
| 10 | 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 |
| 11 | 40 | 1 1 20  1 21 20  21 1 20  21 21 20 |

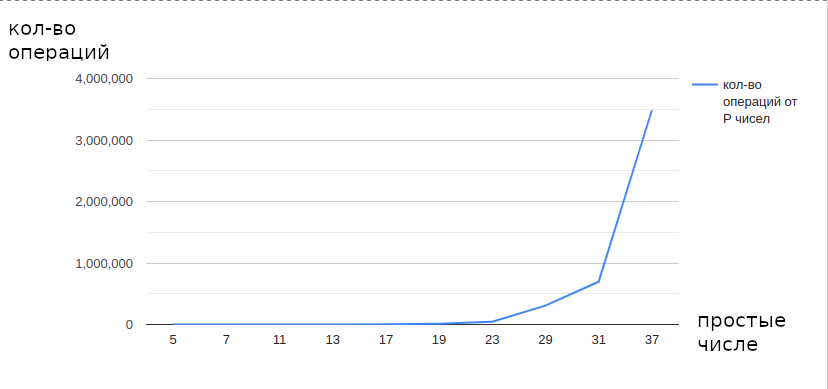
**Исследование кол-ва операций оn размераквадрата.**

Был написан код, который в функции **main()** итеративно перебрал стороны квадрата от 2 до 40. Результат времени выполнения для заданной стороны записывался в векторtimeVec<double>, так же записывали в вектор countVec<long long int> кол-во итераций для разных размеров квадрата. В итоге получили следующие данные:

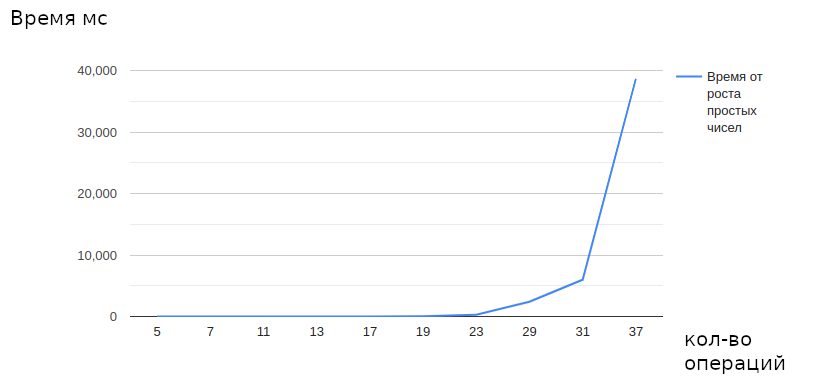
|  |  |  |
| --- | --- | --- |
| Size: 2 | Count Operation: 1 | Time: 0.001 ms |
| Size: 3 | Count Operation: 3 | Time: 0.003 ms |
| Size: 4 | Count Operation: 1 | Time: 0.001 ms |
| Size: 5 | Count Operation: 12 | Time: 0.011 ms |
| Size: 6 | Count Operation: 1 | Time: 0.001 ms |
| Size: 7 | Count Operation: 35 | Time: 0.042 ms |
| Size: 8 | Count Operation: 1 | Time: 0.001 ms |
| Size: 9 | Count Operation: 3 | Time: 0.002 ms |
| Size: 10 | Count Operation: 1 | Time: 0.001 ms |
| Size: 11 | Count Operation: 376 | Time: 0.8 ms |
| Size: 12 | Count Operation: 1 | Time: 0.001 ms |
| Size: 13 | Count Operation: 828 | Time: 2.217 ms |
| Size: 14 | Count Operation: 1 | Time: 0.001 ms |
| Size: 15 | Count Operation: 3 | Time: 0.002 ms |
| Size: 16 | Count Operation: 1 | Time: 0 ms |
| Size: 17 | Count Operation: 4621 | Time: 17.364 ms |
| Size: 18 | Count Operation: 1 | Time: 0.001 ms |
| Size: 19 | Count Operation: 12237 | Time: 54.804 ms |
| Size: 20 | Count Operation: 1 | Time: 0.001 ms |
| Size: 21 | Count Operation: 3 | Time: 0.002 ms |
| Size: 22 | Count Operation: 1 | Time: 0.001 ms |
| Size: 23 | Count Operation: 45076 | Time: 271.026 ms |
| Size: 24 | Count Operation: 1 | Time: 0.001 ms |
| Size: 25 | Count Operation: 12 | Time: 0.011 ms |
| Size: 26 | Count Operation: 1 | Time: 0.001 ms |
| Size: 27 | Count Operation: 3 | Time: 0.002 ms |
| Size: 28 | Count Operation: 1 | Time: 0.001 ms |
| Size: 29 | Count Operation: 306178 | Time: 2397.69 ms |
| Size: 30 | Count Operation: 1 | Time: 0.001 ms |
| Size: 31 | Count Operation: 695875 | Time: 5971.71 ms |
| Size: 32 | Count Operation: 1 | Time: 0.001 ms |
| Size: 33 | Count Operation: 3 | Time: 0.002 ms |
| Size: 34 | Count Operation: 1 | Time: 0.001 ms |
| Size: 35 | Count Operation: 12 | Time: 0.011 ms |
| Size: 36 | Count Operation: 1 | Time: 0.001 ms |
| Size: 37 | Count Operation: 3483061 | Time: 38652.4 ms |
| Size: 38 | Count Operation: 1 | Time: 0.001 ms |
| Size: 39 | Count Operation: 3 | Time: 0.002 ms |
| Size: 40 | Count Operation: 1 | Time: 0.001 ms |

Как можно заметить, что кол-во итераций растет с каждым последующим простым числом, так как эти числа нельзя разложить, то есть нельзя упростить задачу для вычисления алгоритмом backtracking. Давайте определим сложность роста.

Для наглядности сначала построим график зависимости количества операций от простых чисел

*график зависимости количества операций от простых чисел*

Так же построим график зависимости времени от роста простых чисел.

*график зависимости времени от роста простых чисел.*

В моем коде сложность роста *counterOpertion* будет зависеть от размера входных данных - значения *n*. В данном алгоритме используется рекурсия, и количество операций увеличивается с каждым уровнем рекурсии.

При каждом вызове функции *backtrack* мы проходим два цикла *for*, каждый из которых выполняется *n* раз. Затем мы проходим по массиву *sq*, который может иметь до *n* элементов. Затем мы выполняем проверку условий и вызываем функцию *backtrack* рекурсивно. Таким образом, время выполнения функции *backtrack* будет расти экспоненциально с увеличением *n*, что приведет к росту *counterOpertion*.

# Выводы.

В результате работы была написана программа, решающая поставленную задачу при помощи рекурсивного бэктрекинга. По результатам исследования зависимость времени работы алгоритма от размера квадрата экспоненциальная.

# **ПРИЛОЖЕНИЕ А ИСХОДНЫЙ**

# **КОД ПРОГРАММЫ**

Название файла *main.cpp*:

#include <iostream>

#include <vector>

#include <cmath>

#include <chrono>

using namespace std;

class Border {

public:

    int x;

    int y;

    int w;

};

bool overlaySearch(const vector<Border> &vec, int x, int y) {

    for (auto sq: vec) {

        if (x >= sq.x && x < sq.x + sq.w && y >= sq.y && y < sq.y + sq.w) {

            return true;

        }

    }

    return false;

}

void backtrack(vector<Border> &map, int occupancySquare, int bordlen,

               int xLeftOccupancy, int yLeftOccupancy, int n, int &bestcount, vector<Border> &bestarry,

               int &counterOpertion) {

    counterOpertion++;

    for (int x = xLeftOccupancy; x < n; ++x) {

        for (int y = yLeftOccupancy; y < n; ++y) {

            if (!overlaySearch(map, x, y)) {

                int maxside = min(n - 1, min(n - x, n - y));

                for (auto sq: map) {

                    if (sq.x + sq.w > x && sq.y > y) {

                        maxside = min(maxside, sq.y - y);

                    }

                }

                for (int ms = maxside; ms >= 1; --ms) {

                    Border sq{x, y, ms};

                    vector<Border> tmpBest = map;

                    tmpBest.push\_back(sq);

                    if (occupancySquare + pow(sq.w, 2) == n \* n) {

                        if (bordlen + 1 < bestcount) {

                            bestcount = bordlen + 1;

                            bestarry = tmpBest;

                        }

                    } else {

                        if (bordlen + 1 < bestcount) {

                            backtrack(tmpBest, occupancySquare + pow(sq.w, 2), bordlen + 1,

                                      xLeftOccupancy, yLeftOccupancy, n, bestcount, bestarry, counterOpertion);

                        } else {

                            return;

                        }

                    }

                }

                return;

            }

        }

        xLeftOccupancy = n / 2;

        yLeftOccupancy = 0;

    }

}

vector<Border> initStartSqOptimization(int n) {

    vector<Border> map;

    map.push\_back({0, 0, (n + 1) / 2});

    map.push\_back({0, (n + 1) / 2, n / 2});

    map.push\_back({(n + 1) / 2, 0, n / 2});

    return map;

}

int maxDivision(int n, int &sqSize) {

    sqSize = 2;

    for (int i = 1; i <= n / 2; ++i) {

        if (n % i == 0) {

            sqSize = i;

        }

    }

    return n / sqSize;

}

int main() {

    vector<double> timeVec;

    vector<long long int> countVec;

    int counterOpertion = 0;

    for (int i = 2, n = 2; n <= 40; ++i) {

        counterOpertion = 0;

        int multiSqSize;

        n = maxDivision(n, multiSqSize);

        int bestcount = 2 \* n + 1;

        int boardLen = 3;

        vector<Border> map = initStartSqOptimization(n);

        vector<Border> bestSolve;

        int OccupancySq =

                ((n + 1) / 2) \* ((n + 1) / 2) + 2 \* (n / 2) \* (n / 2);// площадь заполненных квадратов;

        int xLeftOccupancy = n / 2, yLeftOccupancy = (n + 1) / 2; //координаты левее которых заполнена столешница

        clock\_t start = clock();

        backtrack(map, OccupancySq, boardLen, xLeftOccupancy, yLeftOccupancy, n, bestcount, bestSolve, counterOpertion);

        clock\_t end = clock();

        double seconds = double(end - start) \* 1000 / CLOCKS\_PER\_SEC;

        countVec.push\_back(counterOpertion);

        timeVec.push\_back(seconds);

        std::cout << "Size: " << i-1  << '\n';

        std::cout << bestcount << '\n';

        for (auto i: bestSolve) {

            cout << 1 + i.x \* multiSqSize << ' ' << 1 + i.y \* multiSqSize << ' ' << i.w \* multiSqSize << '\n';

        }

        n = i;

    }

    for (auto i = 0; i < countVec.size(); i++) {

        std::cout << "| Size: " << i+1 << " | Count Operation: " << countVec[i] << " | Time: "<< timeVec[i] << "  ms | \n";

    }

}