**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. 3342 |  | Песчатский С. Д. |
| Преподаватель |  | Виноградова Е. В. |

Санкт-Петербург

2025

## 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.



Рисунок 1 – пример расположения квадратов для тестового случая

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

Индивидуальный вариант: 2р. Рекурсивный бэктрекинг. Исследование времени выполнения от размера квадрата.

**Описание алгоритма**

Алгоритм решает задачу разбиения квадрата размером n\*n на минимальное количество целочисленных квадратов с использованием рекурсивного бэктрекинга. Основная идея заключается в последовательном размещении квадратов, начиная с больших и постепенно уменьшая их размер. Для хранения состояния используется двумерный массив, на котором отмечаются области, занятые квадратами.

Создается пустая сетка (матрица) размером n\*n, где каждая клетка изначально равна 0 (не занята). Инициализируются переменные для хранения лучшего решения result и текущего решения current. Рекурсивная функция Req принимает текущий индекс клетки ind, размер квадрата size, размер сетки n, текущую сетку map. Если текущее решение хуже лучшего (количество квадратов больше или равно лучшему решению), функция завершает выполнение (отсечение невыгодных ветвей). Если сетка полностью заполнена, текущее решение сравнивается с лучшим, и, если оно лучше, обновляется лучшее решение.

Алгоритм ищет первую пустую клетку, начиная с индекса ind. Если все клетки заполнены, текущее решение сохраняется как лучшее (если оно оптимально). Для каждой пустой клетки перебираются возможные размеры квадратов, начиная с максимального возможного в данном месте и уменьшая до 1. Для каждого размера проверяется, можно ли разместить квадрат в текущей клетке с помощью функции IsValid. Если размещение возможно, квадрат размещается на сетке с помощью функции PlaceSquare, и выполняется рекурсивный вызов для следующей клетки. После рекурсивного вызова выполняется откат изменений с помощью функции RemoveSquare, чтобы вернуться к предыдущему состоянию и попробовать другие варианты.

Алгоритм использует несколько оптимизаций для ускорения работы. Если текущее решение уже хуже лучшего, дальнейший поиск в этой ветви прекращается. Размещение квадратов начинается с максимального возможного размера, что уменьшает количество вариантов для перебора. Для ускорения поиска первой пустой клетки используется линейный поиск, начиная с текущего индекса.

Сложность алгоритма в худшем случае экспоненциальная - O(4^(n\*n)) или даже хуже, так как перебираются все возможные комбинации размещения квадратов. Однако благодаря оптимизациям (отсечение невыгодных ветвей, приоритетное размещение крупных квадратов) сложность значительно снижается. В лучшем случае сложность может быть близка к O(n^2), но в худшем случае остается экспоненциальной, хотя и с меньшим основанием.

Потребление памяти зависит от глубины рекурсии и количества хранимых состояний. В худшем случае, если не использовать оптимизации, память может достигать O(), но благодаря отсечению невыгодных ветвей и раннему завершению, потребление памяти значительно сокращается. В среднем случае память оценивается как O(n^2), так как хранится только текущее состояние сетки и список квадратов.

**Оптимизация алгоритма**

Алгоритм разбиения квадрата n×n на минимальное количество целочисленных квадратов использует несколько оптимизаций, которые значительно сокращают количество рассматриваемых состояний и улучшают его производительность.

1. Размещение первых трех квадратов

В начале алгоритма не сложно высчитать размеры первых трех квадратов, во особенности если n является составным числом. Если n простое, то размер самого большого считается как , а для оставшихся двух . Если число составное, то используются формулы и , где t – минимальный делитель числа n, которое при этом не 1 и не n.

2. Пропуск очевидно невыгодных состояний

Если текущее состояние содержит больше квадратов, чем уже найденное лучшее решение, оно сразу отбрасывается. Это позволяет отсечь большие ветви перебора и ускорить поиск.

3. Оптимизированный поиск первой свободной клетки

Используется переменная ind, отвечающая за нахождение индекса первой пустой клетки. Она используется чтобы не рассматривать уже занятые использованные клетки, при нахождении первой пустой.

4. Запрет размещения квадратов, которые выходят за границы или перекрывают другие квадраты

Прежде чем пытаться разместить квадрат, алгоритм проверяет, можно ли его разместить. Если квадрат не помещается, он не рассматривается, что сокращает количество лишних проверок.

5. Хранение только одного состояния

Чтобы уменьшить затраты алгоритма по памяти и по времени выполнения, хранится только одно поле на которое добавляют или с которого убирают квадраты.

Благодаря этим оптимизациям алгоритм избегает полного перебора всех возможных разбиений и быстро находит решение даже для сравнительно больших значений n.

**Способ хранения частичных решений**

Для хранения частичных решений в предложенной программе используется матрица map, и два массива result и current для хранения лучшего решения и текущего решения. Map хранит текущее состояние поля и изменяется перед каждым новым вызовом рекурсивной функции, для размещения квадрата и после вызова, чтобы убрать соответствующий квадрат. Result хранит только тот набор квадратов, который получилось разместить на поле и обновляется только при условии когда ind = n\*n, в то время как current хранит текущий набор квадратов. Если длина current равна или превышает длину result, то решение сразу отсекается.

**Описание функций и структур данных**

1. void Req(int ind, int n, vector<vector<int>>& map)  
   Рекурсивная функция.
   * ind: индекс пустой клетки
   * n: размер столешницы.
   * map: матрица, хранящее текущее состояние поля.

Функция сначала проверяет длину текущего решения, если длина слишком велика, то вызывается возвращение на предыдущий шаг. Затем, используя формулы и при условии, что поле пустое, расставляются первые три квадрата в зависимости от n. Дальше ищется индекс первой свободной клетки и, если он равен n\*n записывается новый результат и происходит возвращение на предыдущий шаг. Используя найденную пустую клетку как координаты, и проверяются все возможные размеры нового квадрата, найдя подходящий, выставляется на поле, после этого снова вызывается Req. Когда происходит возврат из Req, последний поставленный квадрат убирается с поля.

1. int isComp(int number)  
   Функция для нахождения минимального делителя числа. Для простых чисел возвращает 0.  
   Параметры:
   * number: Число, для которого необходимо найти делитель.
2. bool IsValid(int x, int y, int size, int n, const vector<vector<int>>& map)  
   Функция, проверяющая, можно ли разместить квадрат размера size×size на координатах (x, y).  
   Параметры:
   * x и y: Координаты верхнего левого угла квадрата.
   * size: Размер квадрата.
   * n: Размер столешницы.
   * map: Матрица столешницы.  
     Функция проверяет, что все клетки, которые будут заняты квадратом, пусты (имеют значение 0). Если квадрат можно разместить, возвращается true, иначе — false.
3. void PlaceSquare(int x, int y, int size, const vector<vector<int>>& map)  
   Функция для размещения квадрата на столешнице.  
   Параметры:
   * x и y: Координаты верхнего левого угла квадрата.
   * size: Размер квадрата.
   * map: Матрица столешницы, которая изменяется в результате размещения квадрата.  
     Функция проходит по всем клеткам, которые занимает квадрат, и заполняет их значением current.size() + 1.
4. void RemoveSquare(int x, int y, int size, const vector<vector<int>>& map)  
   Функция для удаления квадрата со столешницы.  
   Параметры:
   * x и y: Координаты верхнего левого угла квадрата.
   * size: Размер квадрата.
   * map: Матрица столешницы, которая изменяется в результате размещения квадрата.  
     Функция проходит по всем клеткам, которые занимает квадрат, и заполняет их значением 0.

Глобальные переменные:

1. result – массив, хранящий лучшее решение.
2. current – массив, хранящий текущее решение.

**Тестирование**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 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 | Верный вывод |
|  | 100 | 4  1 1 50  51 1 50  1 51 50  51 51 50 | Верный вывод |
|  | 21 | 6  1 1 14  15 1 7  1 15 7  8 15 7  15 8 7  15 15 7 | Верный вывод |
|  | 15 | 6  1 1 10  11 1 5  1 11 5  6 11 5  11 6 5  11 11 5 | Верный вывод |

## Исследование времени выполнения от размера квадрата

Для исследования алгоритма по времени выполнения от размера квадрата запустим программу для n от 2 до 44. Также замерим время, через которое наша функция возвращает результат, с помощью clock\_t.

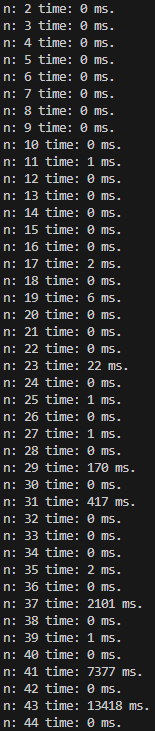
Получили результаты, представленные в рис 2.

Рисунок 2 – результаты времени выполнения алгоритма для различных n

По данным результатам видно, что различаются изменения времени выполнения для простых и составных n. Значит оптимизация с составными квадратами успешно работает.

Построим отдельно графики для составных и простых чисел, чтобы подтвердить данную гипотезу (см. рис. 3).

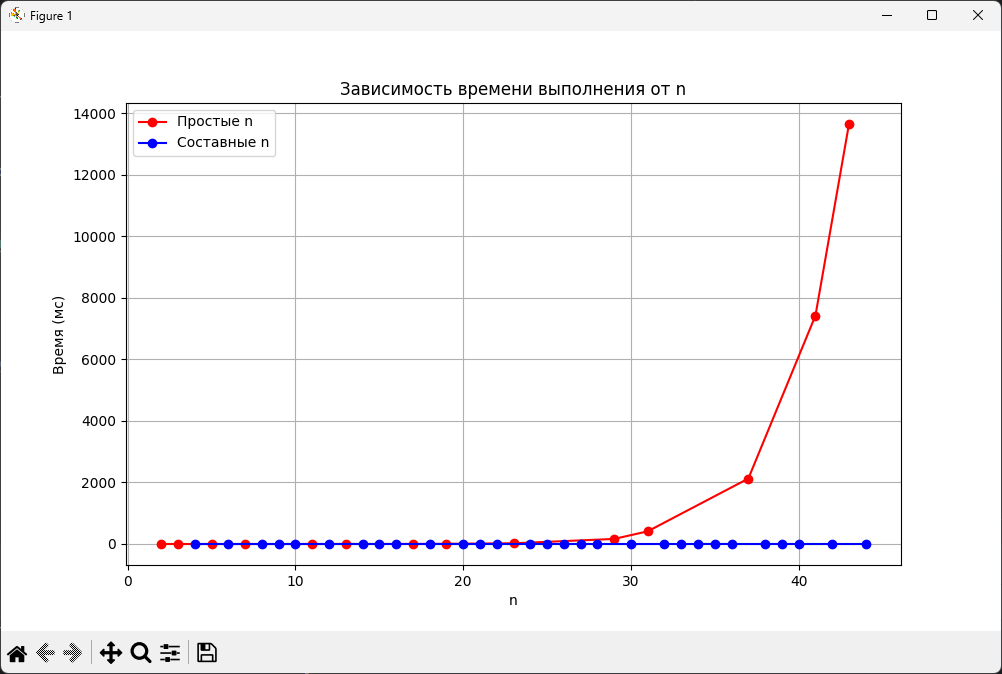


Рисунок 3 – график зависимости времени выполнения алгоритма от n

По данному графику мы заметим, что на маленьких значениях разница не особо заметна. Однако для больших n видно, что время для нечетных растет экспоненциально. Для четных n, наша оптимизация справляется за O(1), что прекрасно видно на графике.

В результате исследования мы получили, что время выполнения алгоритма для простых n будет возрастать экспоненциально, а у составных сложность не меняется.

# Приложение А Исходный код программы

Название файла: main.cpp

#include <iostream>

#include <vector>

using namespace std;

vector<vector<int>> result; // Лучшее решение

vector<vector<int>> current; // Текущее решение

int isComp(int number)

{

for(int i=2; i<number; i++){if(number%i==0)return i;}

return 0;

}

// Проверка, можно ли разместить квадрат размера size в позиции (x, y)

bool IsValid(int x, int y, int size, int n, const vector<vector<int>>& map) {

if (x + size > n || y + size > n) return false;

for (int i = x; i < x + size; ++i) {

for (int j = y; j < y + size; ++j) {

if (map[i][j] != 0) return false;

}

}

return true;

}

// Размещение квадрата размера size в позиции (x, y)

void PlaceSquare(int x, int y, int size, vector<vector<int>>& map) {

for (int i = x; i < x + size; ++i) {

for (int j = y; j < y + size; ++j) {

map[i][j] = current.size() + 1;

}

}

current.push\_back({x, y, size});

}

// Удаление квадрата размера size из позиции (x, y)

void RemoveSquare(int x, int y, int size, vector<vector<int>>& map) {

for (int i = x; i < x + size; ++i) {

for (int j = y; j < y + size; ++j) {

map[i][j] = 0;

}

}

current.pop\_back();

}

// Рекурсивная функция для поиска минимального разбиения

void Req(int ind, int n, vector<vector<int>>& map) {

// Если текущее решение хуже лучшего, прекращаем поиск

if (!result.empty() && (current.size() >= result.size())) return;

if(current.empty()){

int tmp = isComp(n);

if(!tmp){

PlaceSquare(0,0,(n+1)/2, map);

PlaceSquare((n+1)/2,0,n/2, map);

PlaceSquare(0,(n+1)/2,n/2, map);

}

else{

PlaceSquare(0,0,(n/tmp)\*((tmp+1)/2), map);

PlaceSquare((n/tmp)\*((tmp+1)/2),0,(n/tmp)\*(tmp/2), map);

PlaceSquare(0,(n/tmp)\*((tmp+1)/2),(n/tmp)\*(tmp/2), map);

}

}

// Находим первую пустую клетку

while (ind < n \* n) {

if (map[ind / n][ind % n] == 0) break;

ind++;

}

// Если все клетки заполнены, сохраняем решение

if (ind == n \* n) {

if (result.empty() || current.size() < result.size()) {

result = current;

}

return;

}

// Пробуем квадраты от максимального до минимального размера

int maxSize = min(min(n-1, n-ind/n), n-ind%n);

for (int size = maxSize; size >=1; --size) {

// Проверяем, можно ли разместить квадрат

if (IsValid(ind / n, ind % n, size, n, map)) {

// Размещаем квадрат

PlaceSquare(ind / n, ind % n, size, map);

// Рекурсивный вызов

Req(ind + size, n, map);

// Откат изменений

RemoveSquare(ind / n, ind % n, size, map);

}

}

}

int main() {

int n;

cin>>n;

// Инициализация сетки

vector<vector<int>> map(n, vector<int>(n, 0));

// Запуск рекурсивного поиска

const clock\_t c\_start = clock();

Req(0,n, map);

const clock\_t c\_end = clock();

//cout<<(c\_end-c\_start)<<" ms."<<"\n";

// Вывод результата

cout << result.size() << "\n";

for (size\_t i = 0; i < result.size(); ++i) {

cout << result[i][0] + 1 << " " << result[i][1] + 1 << " " << result[i][2] << "\n";

}

return 0;

}