МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по курсовой работе
по дисциплине «Построение и Анализ алгоритмов»
Тема: Поиск с возвратом
Вариант 1р.

Студент гр. 3388 _____ Снигирёв А.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2024

Цель работы

Изучить концепцию и механизм работы поиска с возвратом. Реализовать программу, решающую задачу квадрирования квадратной области.

Задание

Вариант 1р

Выполнить все 3 задания курса на Stepik.

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

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 ($1 \le x,y \le N$) и длину стороны соответствующего обрезка(квадрата).

Выполнение работы

Описание алгоритма. Суть бэктрекинга в переборе всех возможных или наиболее подходящих вариантов «жадным» методом.

Алгоритм получает какую-то область, которую необходимо заполнить и начинает искать в ней пустые клетки вставлять в них квадраты максимального размера, пока пустых мест не останется. После этого идет подсчет получившегося количества квадратов и возврат на предыдущий уровень рекурсии. Последний вставленный квадрат удаляется, а вместо него вставляется квадрат поменьше, если такой есть, и идет подсчет уже этого случая.

Если решение оказывается лучшим предыдущего, то происходит замена, иначе — возврат.

Таким образом, алгоритм перебирает всевозможные варианты размещения квадратов, исследуя дерево путей на каждом квадрате. На выходе получаем минимальный набор квадратов.

Так работает алгоритм, построенный «в лоб». Однако временная сложность такого алгоритма очень велика. После вставки квадрата получаем сразу множество возможных путей. Конкретно (N-1) без оптимизаций. И так для каждой клетки. Конечно, в процессе многие варианты будут отпадать сами собой, но все же в худшем случае имеем сложность O(n^(n^2)), что не очень хорошо.

Тест 1:

N = 7, число рекурсий = 30M.

Для следующего простого числа, 11, завершения программы дождаться не удалось.

Первая оптимизация

Очевидна. Для ее выполнения нужно добавить еще одно условие возврата из рекурсии — если квадратов уже стало больше лучшего случая, а площадь еще не замостилась.

Это сразу отсекает огромное количество возможных путей и в десятки раз ускоряет программу.

Тест 2:

N	Рекурсивные вызовы
7	4368
11	262228
13	821246
17	12M
19	70M

По результатам тестов это степенная функция, немного быстрее или такая же по скорости роста, как экпонента.

Вторая оптимизация

Вторая оптимизация основана на факте, что при размещения квадрата со стороной, близкой к N, остальных маленьких квадратов будет слишком много. Поэтому считать максимальную сторону как N-1 и отталкиваться от этого факта нецелесообразно.

Нужно выбрать число, с которого начнется расчет. Чтобы это сделать, нам следует доказать один важный факт:

Разбиения составных чисел идентичны, с точностью до множителя, разбиению своего минимального простого образующего множителя, т.е кратны этому разбиению.

Так, при N=2, квадрат разделится на четыре единичных, а у любого четного числа на четыре квадрата стороны N/2. Меньшего разбиения просто нет.

При 3**N**эN похожая ситуация, только квадратов 6 штук: 1 большой и 5 поменьше вокруг него. У квадрата стороны 3, больший квадрат имеет сторону 2.

Экспериментально выяснилось, что этот факт справедлив для всех простых чисел нашего диапазона.

В большей части простых чисел, сторона максимального квадрата оказалась (N+1)/2. Она и была взята за максимум.

По результатам эксперимента, сложность алгоритма эта оптимизация не уменьшила, однако рекурсивных вызовов стало меньше на некоторый числовой коэффициент.

Третья оптимизация

Вытекает напрямую из предыдущей. Если в разбиении всегда участвует квадрат стороны (N+1)/2, то оправдано будет добавить этот квадрат заранее — до запуска рекурсии. Также добавляются два других квадрата справа и снизу со сторонами N – (N+1)/2, поскольку они оказались постоянными элементами одного из минимальных решений.

Эти манипуляции сразу же заполняют большую часть площади квадрата, оставляя «почти» квадрат в 4 раза меньший исходного. Это также дает огромный прирост производительности, однако сложность все еще примерно экспоненциальная, просто коэффициент стал намного меньше.

Тест 3:

N	Рекурсивные вызовы
7	52
11	705
13	1 603
17	9 961
19	28 263
23	105 687
29	733 266
31	1 746 937
37	8 463 487
41	28 047 082

Как видно, число рекурсий снова очень существенно сократилось.

Мелкие оптимизации

Были добавлены две небольшие оптимизации, не влияющие на число рекурсий, но благоприятно влияющие на общее время.

1) Хранение остаточной площади и ее проверка

2) Поиск пустой клетки в матрице за O(n) вместо O(n^2)

Метод хранения промежуточных решений и детали реализации

Лучшее разбиение хранится в векторе кортежей `best_solution`. Тогда как текущее разбиение лежит в векторе `squares`. Текущая матрица разбиений лежит в векторе целых чисел `grid`. Ссылки на эти объекты, а также размер стороны квадрата N и доступная площадь `remaining_area` передаются в рекурсивную функцию `void backtrack()`, в котором и прописана логика алгоритма.

`Void backtrack()` - проверяет вышеуказанные условия на каждом уровне рекурсии и отсекает неподходящие решения. Если текущее разбиение корректно, вызывает функцию `pair<int,int> find_place()`, которая возвращает координаты ближайшей пустой клетки или -1, если матрица заполнена.

Если происходит возврат -1, то происходит сравнение `squares` и `best_solution`, в случае если новое решение лучше, то совершается замена и вывод новой матрицы по желанию. Далее совершается выход на предыдущий уровень рекурсии.

Если координаты получены корректные, то происходит расчет максимальной возможной стороны квадрата, который можно вставить на данном шаге.

После этого стартует цикл от этой величины до 0. Внутри цикла делается проверка возможности вставки квадрата текущей стороны. Если можно — квадрат вставляется в `squares`, отрисовывается в `grid` и происходит

спуск на следующий уровень рекурсии, иначе — переход на следующую итерацию.

После возвращения с более низкого уровня рекурсии, что означает получение нового разбиения или сигнализирует о неоптимальности отправленного отсюда разбиения. Последний вставленный квадрат удаляется из текущего вектора квадратов и с поля. Далее происходит переход на следующую итерацию и попытка вставить квадрат с меньшей стороной.

Данный цикл в конце концов дойдет до 1 — вставки минимально возможного квадрата и завершится. По завершении цикла на самом первом уровне рекурсии получим лучшее разбиение из возможных.

Используемая память:

Используемая память состоит из памяти, занимаемой используемыми объектами и памяти, расходующейся на вызовы функций и рекурсивные вызовы.

Память объектов:

O(n^2) для матрицы и используемых двумерных векторов

O(n) для одномерных векторов

Память на вызовы: O(e^n) – рекурсии + вспомогательные функции

Итог: **O(e^n)**

Тестирование граничных случаев:

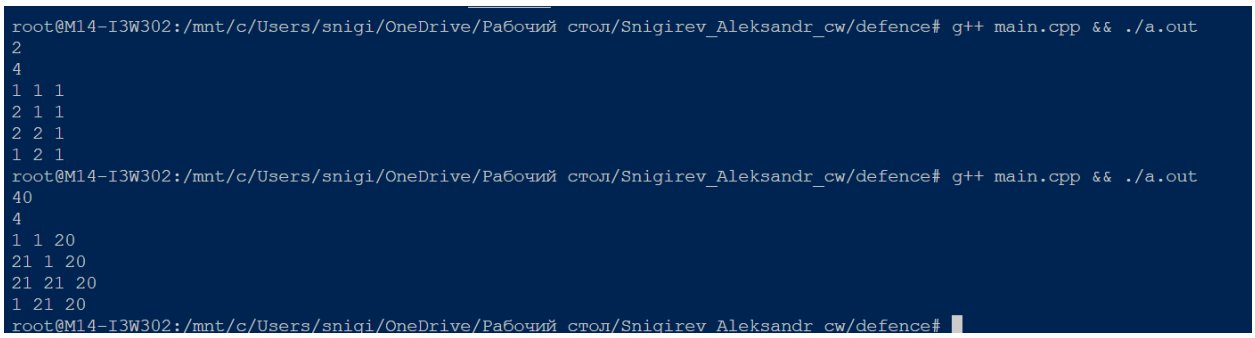

Рис. 1 — Тестирование граничных случаев

Исходный код см. в **ПРИЛОЖЕНИИ А**

Выводы

В ходе выполнения лабораторной работы был реализован и проанализирован алгоритм квадрирования квадрата, использующий поиск с возвратом или бэктрекинг.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp

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

using namespace std;

unsigned long int A;


bool is_valid(vector<vector<int>>& grid, int x, int y, int size,
int N) {
    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 (grid[x][j] != 0) return false;
    return true;
}

void place_square(vector<vector<int>>& grid, int x, int y, int
size, int label) {
    for (int i = x; i < x + size; i++)
        for (int j = y; j < y + size; j++)
            grid[i][j] = label;
}

void remove_square(vector<vector<int>>& grid, int x, int y, int
size) {
    for (int i = x; i < x + size; i++)
        for (int j = y; j < y + size; j++)
            grid[i][j] = 0;
```

```
}

pair<int,int> find_place(vector<vector<int>>& matrix){
    pair<int, int> result = {-1,-1};
    for (int x = 0; x < matrix.size(); x++) {
        for (int y = 0; y < matrix.size(); y++) {
            if (matrix[x][y] == 0) {
                result.first = x;
                result.second = y;
                break;
            }
        }
        if (result.first != -1) break;
    }
    return result;
}

void backtrack(vector<vector<int>>& grid, vector<tuple<int, int, int>>& squares, int N, vector<tuple<int, int, int>>& best_solution, int remaining_area) {
          if (!best_solution.empty() && squares.size() >= best_solution.size()) return;


    pair<int,int> coords = find_place(grid);
    int min_x = coords.first;
    int min_y = coords.second;
    if (min_x == -1) {
                if (best_solution.empty() || squares.size() < best_solution.size()) {
            best_solution = squares;
        }
        return;
    }
```

```
    int max_size = min(N - min_x, N - min_y);
    if(max_size > N - (N+1)/2){
        max_size = N-(N+1)/2;
    }
    for (int size = max_size; size > 0; size--) {
          if (size*size<=remaining_area && is_valid(grid, min_x,
min_y, size, N)) {
                        place_square(grid, min_x, min_y, size,
squares.size() + 1);
            squares.emplace_back(min_x, min_y, size);
            remaining_area-=(size*size);
            A++;
                    backtrack(grid, squares, N, best_solution,
remaining_area);
            squares.pop_back();
            remove_square(grid, min_x, min_y, size);
            remaining_area+=(size*size);
        }
    }
}


vector<tuple<int, int, int>> squaring_the_square(int N) {
    vector<vector<int>> grid(N, vector<int>(N, 0));
    vector<tuple<int, int, int>> best_solution;
    if(N%2==0){
        best_solution.emplace_back(0, 0, N/2);
        best_solution.emplace_back(N/2, 0, N/2);
        best_solution.emplace_back(N/2, N/2, N/2);
        best_solution.emplace_back(0, N/2, N/2);
        return best_solution;
    }
    if(N%3==0){
        best_solution.emplace_back(0, 0, N*2/3);
        best_solution.emplace_back(0, N*2/3, N/3);
```

```
    best_solution.emplace_back(N/3, N*2/3, N/3);
    best_solution.emplace_back(N*2/3, 0, N/3);
    best_solution.emplace_back(N*2/3, N/3, N/3);
    best_solution.emplace_back(N*2/3, N*2/3, N/3);
    return best_solution;
}
if(N%5==0){
    best_solution.emplace_back(0, 0, N*3/5);
    best_solution.emplace_back(N*3/5, N*2/5, N*2/5);
    best_solution.emplace_back(N*3/5, 0, N*2/5);
    best_solution.emplace_back(0, N*3/5, N*2/5);
    best_solution.emplace_back(N*2/5, N*3/5, N/5);
    best_solution.emplace_back(N*2/5, N*4/5, N/5);
    best_solution.emplace_back(N*3/5, N*4/5, N/5);
    best_solution.emplace_back(N*4/5, N*4/5, N/5);
    return best_solution;
}

vector<tuple<int,int,int>> squares;
int maxW = (N+1)/2;
int bigW = N-maxW;

place_square(grid, 0, 0, maxW, maxW);
tuple<int,int,int> biggest_sqr = {0,0, maxW};
squares.emplace_back(biggest_sqr);

biggest_sqr = {0, maxW, bigW};
squares.emplace_back(biggest_sqr);
place_square(grid, 0, maxW, bigW, bigW);

biggest_sqr = {maxW, 0, bigW};
squares.emplace_back(biggest_sqr);
place_square(grid, maxW, 0, bigW, bigW);
```

```
    backtrack(grid, squares, N, best_solution, N*N);
    return best_solution;
}



void print_solution_matrix(int N, const vector<tuple<int, int,
int>>& solution) {
    vector<vector<int>> matrix(N, vector<int>(N, 0));
    int index = 1;
    for (const auto& [x, y, size] : solution) {
        for (int i = x; i < x + size; i++)
            for (int j = y; j < y + size; j++)
                matrix[i][j] = index;
        index++;
    }
    cout << "Итоговая матрица разбиения:\n";
    for (const auto& row : matrix) {
         for (int cell : row) cout << (cell > 0 ? to_string(cell)
: ".") << " ";
        cout << endl;
    }
}



int main() {
    int N;

    std::cin>>N;

                vector<tuple<int,    int,    int>>    solution    =
squaring_the_square(N);
    std::cout<<solution.size()<<endl;
    for (const auto& [x, y, size] : solution)
```

```
        cout << x+1 << " " << y+1 << " " << size <<endl;
    return 0;
}
```

Литература и используемые ссылки:

https://demonstrations.wolfram.com/MrsPerkinssQuilts/ - готовая визуализация для очень похожей задачи.

https://en.wikipedia.org/wiki/Squaring_the_square