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: Перебор с возвратом

Студент гр. 9383 _____ Арутюнян С.Н.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритмы перебора с возвратом и применить их на примере задачи квадрирования квадрата.

Основные теоретические положения.

Поиск с возвратом или бэктрекинг — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве M.

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. 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.

Необходимо найти число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N и вывести K строк, каждая из которых должна содержать три целых числа x, y, и w, задающие координаты левого верхнего угла и длину стороны соответствующего обрезка (квадрата).

Вариант 5р. Возможность задать список квадратов (от 0 до N^2 квадратов в списке), которые обязательно должны быть использованы в покрытии квадрата со стороной N.

Ход работы:

1. Была разработана структура Square, хранящая координаты левого верхнего угла квадрата и длину его сторону.

2. Был разработан класс Field, представляющий квадрируемый квадрат.

3. Был реализован бэктрекинг-алгоритм, работающий таким образом:

1) На вход функция Backtracking принимает текущие координаты по x и y, текущее поле и минимальное поле по неконстантным ссылкам.

2) Текущие координаты сдвигаются на ближайшую (в порядке обхода поля слева направо сверху вниз) свободную клетку.

3) Запускается цикл по длине текущего квадрата от наибольшей (N — 1) до наименьшей (1).

4) В поле вставляется квадрат с координатами x, y и текущей длиной из цикла.

5) Если после вставки количество свободных клеток в поле равняется 0, то проверяется, меньше ли квадратов в текущем поле, чем в минимальном. Если меньше, то текущее поле копируется в минимальное.

6) Если же количество квадратов в квадрируемом квадрате уже больше минимального количества, то просто удаляем последний вставленный квадрат и выходим из цикла.

7) Иначе рекурсивно запускаем функцию Backtracking с аргументами x+1, y, min_field, current_field.

8) Удаляем последний вставленный квадрат. Итерируемся дальше по циклу.

4. Также, были найдены следующие оптимизации:

1) Если длина стороны квадрата четная, то по очевидным причинам минимальное количество квадратов в разбиении = 4.

2) Если длина стороны квадрата кратна 3, то минимальное количество квадратов в разбиении = 6.

3) Если длина стороны квадрата кратна 5, то минимальное количество квадратов в разбиении = 8.

Также, в остальных ситуациях (т. е. тогда, когда N — простое число, ведь все числа до N <= 40 либо просты, либо кратны 2, 3 или 5) можно применить следующую оптимизацию: в таких квадратах точно содержатся 3 квадрата. Один из них находится в левом верхнем углу и имеет длину стороны N/2 + 1. Остальные два находятся по обе стороны от первого и имеют длину стороны N/2. Это означает, что мы должны обработать с помощью перебора с возвратом лишь четверть от изначального круга.

Код этой версии приведен в приложении А.

5. Также, следующим этапом была разработана программа, принимающая на вход последовательность квадратов, которые обязательно должны присутствовать в разбиении. Код этой программы приведен в приложении Б.

6. Были разработаны тесты с использованием библиотеки Catch2.

Пункты 4 и 5 (вариант без указания обязательных квадратов и с указанием) поделены по двум разным файлам — variant1.cpp и variant2.cpp.

Сложность разработанного алгоритма — O(2^N), т. к. мы перебираем все подмножества множества клеток столешницы.

Описание функций и структур данных:

1. Структура Square представляет собой квадрат. Она хранит координаты x и y, а также длину стороны квадрата.
2. Класс Field представляет собой непосредственно столешницу. Она имеет следующие поля:

   1) std::vector<std::vector<int>> field – непосредственно “карта” столешницы. В качестве значений эта матрица хранит длину квадрата, который охватывает данную клетку.

   2) std::size_t n – длина стороны столешницы.

3) int current_area – текущее количество свободных клеток столешницы (или же просто площадь свободной части столешницы).

4) std::vector<Square> putted_squares – квадраты, учавствующие в текущем разбиении.

И следующие методы:

1) void PutSquare(const Square&) - помещает квадрат на столешницу.

2) void UnputLastSquare() - убирает со столешницы последний вставленный квадрат.

3) const std::vector<int>& operator[](index) const – возвращает строку матрицы-поля.

4) void PrintSquares() const – выводит все квадраты, учавствующие в текущем разбиении, на экран.

5) std::size_t SquaresAmount() const – возвращает размер текущего разбиения.

6) std::size_t N() const – возвращает длину стороны столешницы.

7) int Area() const – возвращает текущее количество свободных клеток.

Примеры работы программы

```
7
Введите квадраты, которые будут участвовать в разбиении:
-1
9
1 1 4
5 1 3
5 4 1
6 4 2
1 5 3
4 5 2
6 6 2
4 7 1
5 7 1
```

Рисунок 1. Пример работы программы (1).

```
19
Введите квадраты, которые будут участвовать в разбиении:
-1
13
1 1 13
14 1 6
14 7 6
14 13 2
16 13 4
1 14 6
7 14 6
13 14 1
13 15 3
16 17 1
17 17 3
13 18 2
15 18 2
```

Рисунок 2. Пример работы программы (2).

```
7
Введите квадраты, которые будут участвовать в разбиении:
2 1 3
5 5 1
-1
12
2 1 3
5 5 1
1 1 1
5 1 3
1 2 1
1 3 1
1 4 4
5 4 1
6 4 2
5 6 1
6 6 2
5 7 1
```

Рисунок 3. Пример работы программы (3).

Выводы.

В выполненной лабораторной работе был изучен и применен на практике алгоритм перебора с возвратом. Также были найдены оптимизации, позволяющие в 4 раза сократить объем работы для алгоритма, а в некоторых случаях даже сократить сложность с O(2^N) до O(1).

ПРИЛОЖЕНИЕ А

FIELD.H

```
#pragma once

#include <vector>
#include <iostream>


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


class Field {
public:
    explicit Field(std::size_t n);
    Field(const Field& other_field) = default;

    void PutSquare(const Square& square);
    void UnputLastSquare();
    bool CheckIntersection(const Square& square);

    const std::vector<int>& operator[](int index) const;

    std::size_t SquaresAmount() const;
    std::size_t N() const;
    int Area() const;
    void PrintSquares() const;

private:
    int current_area;
    std::size_t n;
    std::vector<std::vector<int>> field;
    std::vector<Square> putted_squares;
};
```

FIELD.CPP

```
#include "field.h"


Field::Field(std::size_t n)
  : n(n), current_area(n * n)
{
    field.resize(n, std::vector<int>(n, false));
    putted_squares.reserve(n*n);
}
```

```
const std::vector<int>& Field::operator[](int index) const {
    return field[index];
}

void Field::PutSquare(const Square& square) {
    for (int i = square.x; i < square.x + square.width; ++i) {
        for (int j = square.y; j < square.y + square.width; ++j) {
            field[j-1][i-1] = square.width;
        }
    }

    putted_squares.push_back(square);
    current_area -= square.width * square.width;
}

void Field::UnputLastSquare() {
    auto square = putted_squares.back();

    for (int i = square.x; i < square.x + square.width; ++i) {
        for (int j = square.y; j < square.y + square.width; ++j) {
            field[j-1][i-1] = 0;
        }
    }

    current_area += square.width * square.width;
    putted_squares.pop_back();
}

bool Field::CheckIntersection(const Square& square) {
    for (int i = square.x; i < square.x + square.width; ++i) {
        for (int j = square.y; j < square.y + square.width; ++j) {
            if (field[j-1][i-1])
                return true;
        }
```

```
    }

    RETURN FALSE;
}

STD::SIZE_T FIELD::SQUARESAMOUNT() CONST {
    RETURN PUTTED_SQUARES.SIZE();
}

STD::SIZE_T FIELD::N() CONST {
        RETURN N;
}

INT FIELD::AREA() CONST {
    RETURN CURRENT_AREA;
}

VOID FIELD::PRINTSQUARES() CONST {
    FOR (CONST AUTO& SQUARE : PUTTED_SQUARES) {
        STD::COUT << SQUARE.X << " " << SQUARE.Y << " " << SQUARE.WIDTH << STD::ENDL;
    }
}
```

BACKTRACKING.HPP

```
#pragma once

#include "field.h"

inline void Backtracking(int x, int y, Field& min_field, Field& current_field) {
   auto N = current_field.N();

   while (x <= N && y <= N && current_field[y-1][x-1]) {
      ++x;
      if (x == N + 1) {
         x = 1;
         ++y;
      }
   }

   if (y == N + 1)
      return;

   for (int width = N - 1; width >= 1; --width) {
       if (x + width > N + 1 || y + width > N + 1 || current_field.CheckIntersection({x,
y, width}))
          continue;

      current_field.PutSquare({x, y, width});

             if (current_field.Area() == 0 && current_field.SquaresAmount() <
min_field.SquaresAmount()) {
         min_field = current_field;
      }
      else if (current_field.SquaresAmount() >= min_field.SquaresAmount()) {
         current_field.UnputLastSquare();
         break;
      }
```

```
        else {
            Backtracking(x + 1, y, min_field, current_field);
        }

        current_field.UnputLastSquare();
    }
}
```

Variant1.cpp

```
#include <iostream>
#include "field.h"
#include “backtracking.hpp”

int main() {
    int N; std::cin >> N;

    Field min_field(N);
    Field current_field(N);

    if (N % 2 == 0) {
        min_field.PutSquare({1, 1, N / 2});
        min_field.PutSquare({N/2 + 1, 1, N / 2});
        min_field.PutSquare({1, N/2 + 1, N / 2});
        min_field.PutSquare({N/2 + 1, N/2 + 1, N / 2});
    }
    else if (N % 3 == 0) {
        min_field.PutSquare({1, 1, 2*N/3});
        min_field.PutSquare({1 + 2*N/3, 1, N/3});
        min_field.PutSquare({1, 2*N/3 + 1, N/3});
        min_field.PutSquare({2*N/3 + 1, N/3 + 1, N/3});
        min_field.PutSquare({N/3 + 1, 2*N/3 + 1, N/3});
        min_field.PutSquare({2*N/3 + 1, 2*N/3 + 1, N/3});
    }
    else if (N % 5 == 0) {
```

```
        MIN_FIELD.PUTSQUARE({1, 1, 3*N/5});
        MIN_FIELD.PUTSQUARE({3*N/5 + 1, 1, 2*N/5});
        MIN_FIELD.PUTSQUARE({1, 3*N/5 + 1, 2*N/5});
        MIN_FIELD.PUTSQUARE({3*N/5 + 1, 3*N/5 + 1, 2*N/5});
        MIN_FIELD.PUTSQUARE({2*N/5 + 1, 3*N/5 + 1, N/5});
        MIN_FIELD.PUTSQUARE({2*N/5 + 1, 4*N/5 + 1, N/5});
        MIN_FIELD.PUTSQUARE({3*N/5 + 1, 2*N/5 + 1, N/5});
        MIN_FIELD.PUTSQUARE({4*N/5 + 1, 2*N/5 + 1, N/5});
    }
    ELSE {
        CURRENT_FIELD.PUTSQUARE({1, 1, (N+1)/2});
        CURRENT_FIELD.PUTSQUARE({(N+3)/2, 1, N/2});
        CURRENT_FIELD.PUTSQUARE({1, (N+3)/2, N/2});



        FOR (INT X = 1; X <= N; ++X) {
            FOR (INT Y = 1; Y <= N; ++Y) {
                MIN_FIELD.PUTSQUARE({X, Y, 1});
            }
        }

        BACKTRACKING(1, 1, MIN_FIELD, CURRENT_FIELD);
    }


    STD::COUT << MIN_FIELD.SQUARESAMOUNT() << STD::ENDL;
    MIN_FIELD.PRINTSQUARES();


    RETURN 0;
}
```

ПРИЛОЖЕНИЕ Б

VARIANT2.CPP

```
#include <iostream>
#include "field.h"
#include “backtracking.hpp”


int main() {
   int N; std::cin >> N;


   Field min_field(N);
   for (int x = 1; x <= N; ++x) {
      for (int y = 1; y <= N; ++y) {
         min_field.PutSquare({x, y, 1});
      }
   }
   Field current_field(N);
   std::cout << "Введите квадраты, которые будут участвовать в разбиении:" <<
std::endl;
   for (int i = 0; i < N*N; ++i) {
      int x, y, width;
      std::cin >> x;
      if (x == -1)
         break;


      std::cin >> y >> width;
      current_field.PutSquare({x, y, width});
   }


   Backtracking(1, 1, min_field, current_field);


   std::cout << min_field.SquaresAmount() << std::endl;
   min_field.PrintSquares();
   return 0;
}
```

TESTS.CPP

```
#define CATCH_CONFIG_MAIN
#include "../../catch.hpp"
#include "../source/field.h"
#include "../source/backtracking.hpp"

TEST_CASE("UnputLastSquare tests") {
    Field field(10);

    field.PutSquare({1, 2, 3});
    for (int i = 1; i < 4; ++i) {
        for (int j = 2; j < 5; ++j) {
            REQUIRE(field[j-1][i-1] == 3);
        }
    }
    field.UnputLastSquare();
    for (int i = 1; i < 4; ++i) {
        for (int j = 2; j < 5; ++j) {
            REQUIRE(field[j-1][i-1] == 0);
        }
    }

    field.UnputLastSquare();
    for (int i = 1; i <= field.N(); ++i) {
        for (int j = 1; j <= field.N(); ++j) {
            REQUIRE(field[j-1][i-1] == 0);
        }
    }
}


TEST_CASE("PutSquares borders tests") {
    Field field(5);
```

```
    REQUIRE(FIELD.PUTSQUARE({1, 1, 100}) == FALSE);
    REQUIRE(FIELD.PUTSQUARE({5, 5, 2})   == FALSE);

    REQUIRE(FIELD.PUTSQUARE({1, 1, 1})   == TRUE);
    FIELD.UNPUTLASTSQUARE();
    REQUIRE(FIELD.PUTSQUARE({3, 3, 1})   == TRUE);
    FIELD.UNPUTLASTSQUARE();
    REQUIRE(FIELD.PUTSQUARE({1, 1, 5})   == TRUE);
    FIELD.UNPUTLASTSQUARE();
}


TEST_CASE("CHECKINTERSECTION TESTS") {
    FIELD FIELD(5);
    FIELD.PUTSQUARE({1, 1, 3});

    REQUIRE(FIELD.CHECKINTERSECTION({2, 2, 1}) == TRUE);
    REQUIRE(FIELD.CHECKINTERSECTION({4, 4, 1}) == FALSE);

    FIELD.PUTSQUARE({1, 3, 2});

    REQUIRE(FIELD.CHECKINTERSECTION({1, 3, 2}) == TRUE);
    REQUIRE(FIELD.CHECKINTERSECTION({5, 5, 1}) == FALSE);
}

TEST_CASE("ANSWERS TESTS") {
    FIELD CURRENT_FIELD1(10);
    FIELD MIN_FIELD1(10);
    FOR (INT X = 1; X <= 10; ++X) {
        FOR (INT Y = 1; Y <= 10; ++Y) {
            MIN_FIELD1.PUTSQUARE({X, Y, 1});
        }
    }
```

```
    BACKTRACKING(1, 1, MIN_FIELD1, CURRENT_FIELD1);
    REQUIRE(MIN_FIELD1.AREA() == 0);
    REQUIRE(MIN_FIELD1.SQUARESAMOUNT() == 4);



    FIELD CURRENT_FIELD2(15);
    FIELD MIN_FIELD2(15);
    FOR (INT X = 1; X <= 15; ++X) {
      FOR (INT Y = 1; Y <= 15; ++Y) {
        MIN_FIELD2.PUTSQUARE({X, Y, 1});
      }
    }

    BACKTRACKING(1, 1, MIN_FIELD2, CURRENT_FIELD2);
    REQUIRE(MIN_FIELD2.AREA() == 0);
    REQUIRE(MIN_FIELD2.SQUARESAMOUNT() == 6);



    FIELD CURRENT_FIELD3(19);
    FIELD MIN_FIELD3(19);
    FOR (INT X = 1; X <= 19; ++X) {
      FOR (INT Y = 1; Y <= 19; ++Y) {
        MIN_FIELD3.PUTSQUARE({X, Y, 1});
      }
    }

    BACKTRACKING(1, 1, MIN_FIELD3, CURRENT_FIELD3);
    REQUIRE(MIN_FIELD3.AREA() == 0);
    REQUIRE(MIN_FIELD3.SQUARESAMOUNT() == 13);
}
```