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. 8382 _____ Щемель Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

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.

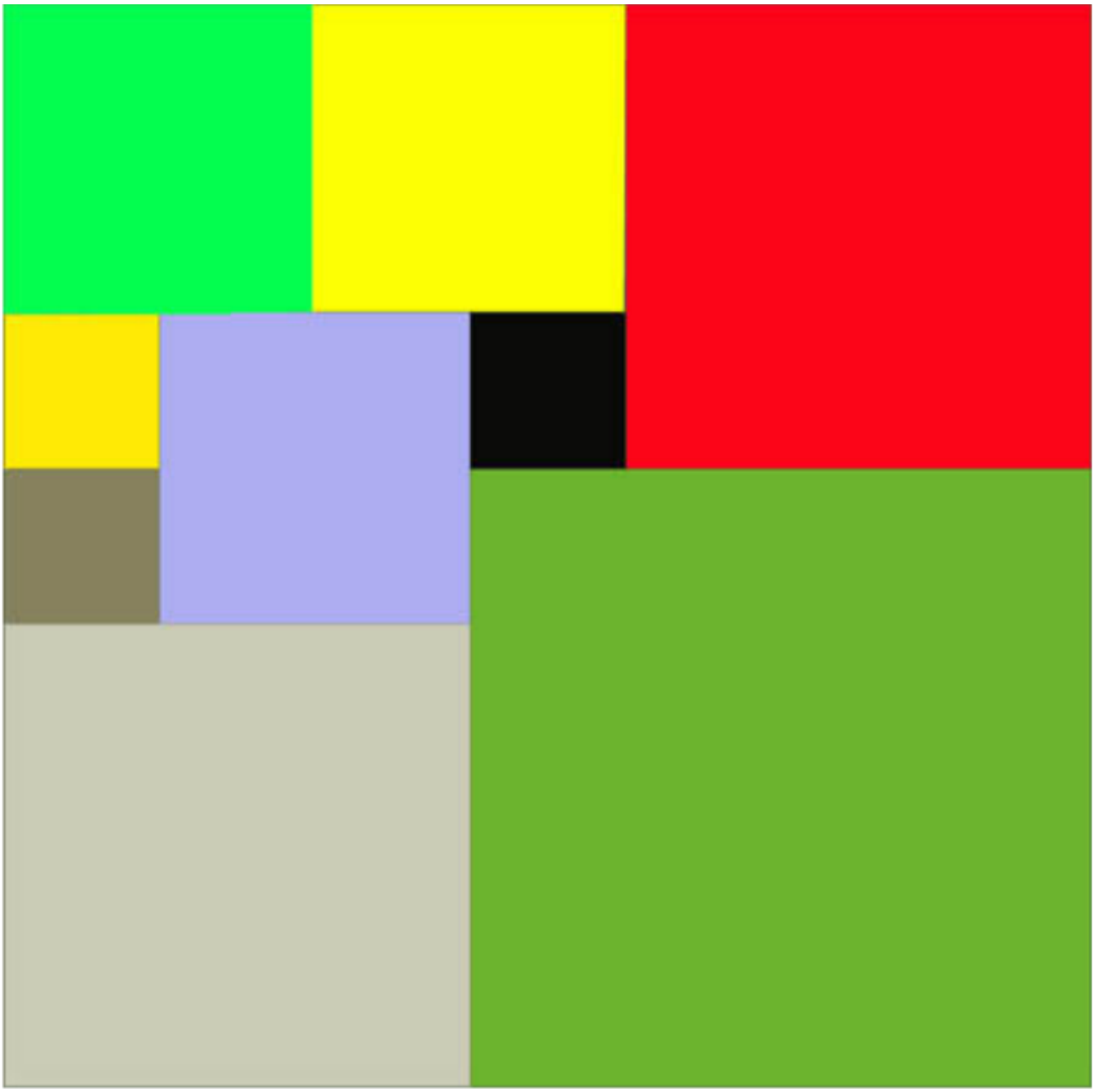

Figure 1: Разбиение квадрата

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.1

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

Вариант № 2И. Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

Описание алгоритма

Для размеров, делящихся на 2, 3 или 5 можно построить оптимальное разбиение, не прибегая к перебору.

Для 2: можно разбить на 4 равные части.

Для 3: можно оптимально разбить на 6 частей: квадрат размером 2/3 N в углу + 5 квадратов размером 1/3 N вокруг него.

Для 5: на 8 частей. 3/5N в одном углу. 2/5N в остальных углах. И остальное пространство заполняется квадратами размером 1/N.

Для других входных данных применяется алгоритм поиска с возвратом с некоторыми оптимизациями. Эмпирическим путём было установлено, что можно сразу поставить 3 квадрата, со следующими размерами по углам: *floor*(N/2)+1 и *floor*(N/2). Далее, в оставшемся углу перебираются квадраты, с размерами от 2 до *floor*(N/2)-1.

Остальное незанятое пространство заполняется(квадрат размером *floor*(N/2) с занятой вершиной в центре столешницы и занятым пространством в углу) алгоритмом перебора с возвратом: ищется свободная ячейка - туда ставится квадрат. После чего квадрат увеличивается до максимальных размеров в этой позиции. И так, пока всё свободное пространство не будет заполнено. Результат сохраняется. После этого из временного стэка с конца удаляются единичные квадраты, до тех пор, пока не будет встречен не единичный квадрат. Его размер уменьшается на 1 и итерация заполнения повторяется. Лучший результат сохраняется. Так же, итерация заполнения не продолжается, если количество квадратов в стэке превышает количество в лучшем на текущем моменте разбиении.

Сложности алгоритма:

Временная: для не простых чисел - $O(1)$. В противном случае - экспоненциальная, из-за перебора с возвратом.

Сложность по памяти: для не простых чисел - $O(1)$. Для простых - $O(N^2)$ (для карты ячеек), где N - длина столешницы.

Описание функций и структур данных

Используются 2 структуры: *Point* из двух чисел для хранения координаты и *Square* для хранения координаты угла и размера с переопределённым оператором вывода в поток.

Используемые функции:

- *std::vector makeFragmentationForDiv2(int n)* для разбиения для N, делящегося на 2
- *std::vector makeFragmentationForDiv3(int n)* для разбиения для N, делящегося на 3
- *std::vector makeFragmentationForDiv5(int n)* для разбиения для N, делящегося на 5
- *std::vector makeBaseFragmentation(int n)* для заполнения 3 большими угловыми квадратами
- *bool backtrackingIter(int n, int center, int** field, std::vector& currentStack, int bestStack-Size)* для поиска свободной ячейки и увеличения размера квадрата. bool в качестве возвращаемого типа используется для проверки успешного завершения итерации(не было ли превышено количество квадратов, чем у лучшего)
- *std::vector findNextFragmentationForDiag(int n, int center, int leftCornerShift)* для поиска лучшего разбиения для заданного размера последнего углового квадрата (размер которого варьируется)
- *std::vector makeBacktrackingFragmentation(int n)* точка входа для разбиение квадрата с простым числом *N*
- *void printDebugSquares(int n, std::vector& squares)* для вывода представления результата разбиения
- *void printSquares(int n, std::vector& squares)* для вывода результата разбиения (положения и геометрия квадратов)

Тестирование

Input	Output
2	4 1 1 1 1 2 1 2 1 1 2 2 1
3	6 1 1 2 1 3 1 2 3 1 3 3 1 3 1 1 3 2 1
5	8 1 1 3 1 4 2 4 1 2 4 4 2 3 4 1 4 3 1 3 5 1 5 3 1
7	9 1 1 4 5 1 3 1 5 3 6 6 2 4 5 2 4 7 1 5 4 1 5 7 1 6 4 2
11	11 1 1 6 7 1 5 1 7 5 10 10 2 6 7 2 6 9 2 6 11 1 7 6 1 7 11 1 8 6 4 8 10 2
13	11 1 1 7 8 1 6 1 8 6 11 11 3 7 8 2 7 10 4 8 7 1 9 7 3 11 10 1 12 7 2 12 9 2
23	13 1 1 12 13 1 11 1 13 11 19 19 5 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

Результаты тестирования приведены на собранном в *Release* бинарном файле без *дебажного* вывода, чтобы не загромождать таблицу.

Исследование

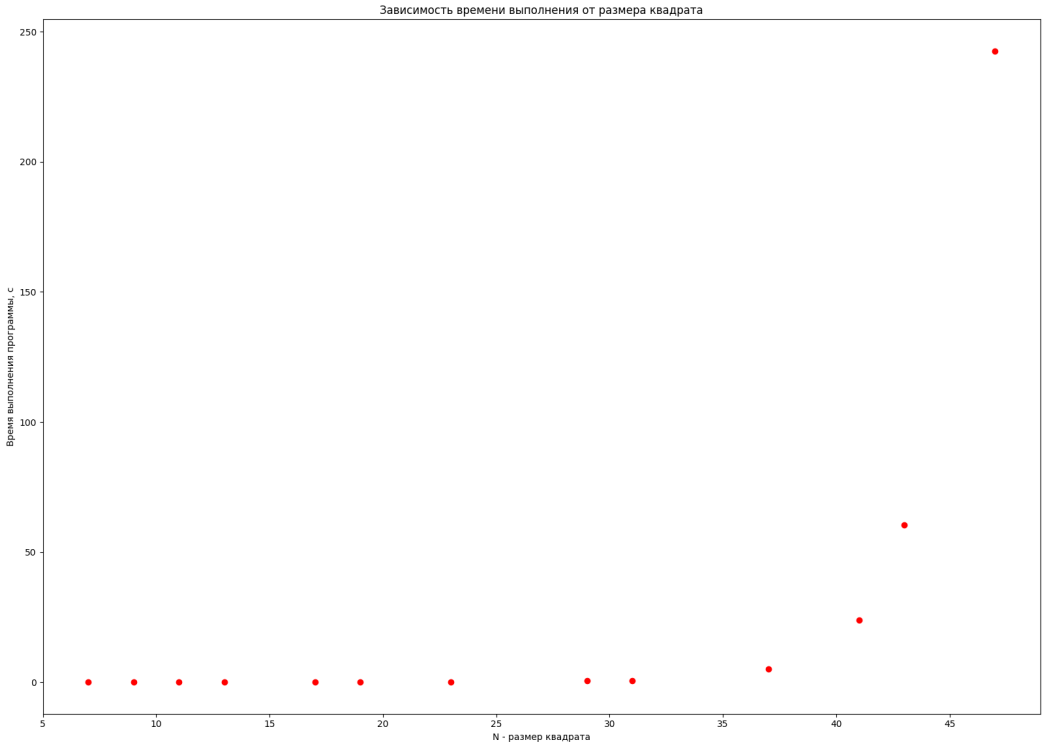

Figure 2: Граф зависимости

На представленном графе видно, что рост происходит экспоненциально.

Вывод

В ходе выполнения лабораторной работы были получены практические навыки применения алгоритма поиска с возвратом, на примере задачи с разбиение квадрата на меньшие квадраты. По результатам исследования видно, что алгоритм полного перебора - дорогой. Были применены различные оптимизации, которые позволяют для определённых размеров квадрата за O(1) получить результат. Для случаев же, когда размер квадрата - простое число, область перебора была сокращена более чем в 4 раза на основе эмпирических данных.

Приложение А. Исходный код

```
#include <iostream>
#include <vector>
#include <chrono>


struct Point
{
    int x;
    int y;
};

struct Square
{
    Point leftCorner;
    int length;

    friend std::ostream& operator<<(std::ostream& os, const Square& s)
    {
        os << s.leftCorner.x + 1 << " " << s.leftCorner.y + 1 << " " << s.length;
        return os;
    }

    Square(int x, int y, int _length)
            : leftCorner{x, y}, length(_length)
    {
    }
};

/// Make fragmentation for N divided by 2
/// \param n size of table
/// \return vector of Square
std::vector<Square> makeFragmentationForDiv2(int n)
{
    std::vector<Square> retSquares;
    int half = int(n / 2);
    for (int i = 0; i < n; i += half)
    {
        for (int j = 0; j < n; j += half)
        {
            retSquares.emplace_back(i, j, half);
        }
```

```
    }

    return retSquares;
}

/// Make fragmentation for N divided by 3
/// \param n size of table
/// \return vector of Square
std::vector<Square> makeFragmentationForDiv3(int n)
{
    std::vector<Square> retSquares;
    int smallInnerN = int(n / 3);
    int bigInnerN = smallInnerN * 2;
    retSquares.emplace_back(0, 0, bigInnerN);

    // Bottom
    for (int i = 0; i < n; i += smallInnerN)
    {
        retSquares.emplace_back(i, n - smallInnerN, smallInnerN);
    }

    // Side
    for (int i = 0; i < n - smallInnerN; i += smallInnerN)
    {
        retSquares.emplace_back(n - smallInnerN, i, smallInnerN);
    }

    return retSquares;
}

/// Make fragmentation for N divided by 5
/// \param n size of table
/// \return vector of Square
std::vector<Square> makeFragmentationForDiv5(int n)
{
    std::vector<Square> retSquares;
    int half = int(n / 5 * 2);
    int third = int(n / 5 * 3);
    int fifth = int(n / 5);
    retSquares.emplace_back(0, 0, third);

    // Big in corners
```

```
    //for a, b in [(0, 1), (1, 0), (1, 1)]:
    retSquares.emplace_back(0, third, half);
    retSquares.emplace_back(third, 0, half);
    retSquares.emplace_back(third, third, half);

    // Small in gaps
    for (int i = 0; i < 2; ++i)
    {
        retSquares.emplace_back(half, third + i * fifth, fifth);
        retSquares.emplace_back(third + i * fifth, half, fifth);
    }

    return retSquares;
}

/// Make base fragmentation for backtracking with 3 big Square
/// \param n size of table
/// \return vector of Square
std::vector<Square> makeBaseFragmentation(int n)
{
    int half = int(n / 2) + 1;
    return {Square(0, 0, half),
            Square(half, 0, half - 1),
            Square(0, half, half - 1)};
}

/// Make backtracking iteration fragmentation
/// \param n size of table
/// \param center center of table
/// \param field field of defect_square
/// \param currentStack vector of currentStack
/// \best_stack_size size of stack with the best fragmentation
bool backtrackingIter(int n, int center, int** field, std::vector<Square>& currentStack,
                      int bestStackSize)
{
    const auto canGrowth = [&](Square& square)
    {
        // Check bottom
        if (square.leftCorner.x + square.length + 1 <= n
        && square.leftCorner.y + square.length + 1 <= n)
        {
```

```
            for (int i = square.leftCorner.x; i < square.leftCorner.x + square.length + 1; ++i)
            {
                if (field[i - center][square.leftCorner.y + square.length - center] != 0)
                    return false;
            }
        }
        else
        {
            return false;
        }
        // Check side
        for (int i = square.leftCorner.y; i < square.leftCorner.y + square.length + 1; ++i)
        {
            if (field[square.leftCorner.x + square.length - center][i - center] != 0)
                return false;
        }

        return true;
    };

    const auto growth_square = [&](Square& square)
    {
        square.length += 1;
        for (int i = square.leftCorner.x; i < square.leftCorner.x + square.length; ++i)
        {
            for (int j = square.leftCorner.y; j < square.leftCorner.y + square.length; ++j)
            {
                field[i - center][j - center] = 1;
            }
        }
    };

    for (int i = center; i < n; ++i)
    {
        for (int j = center; j < n; ++j)
        {
            if (field[i - center][j - center] == 0)
            {
                auto tmpSquare = Square(i, j, 1);
                field[i - center][j - center] = 1;
                while (canGrowth(tmpSquare))
                {
```

```
                    growth_square(tmpSquare);
                }

                currentStack.push_back(tmpSquare);

                for (int i = center; i < n; ++i)
                {
                    for (int j = center; j < n; ++j)
                    {
                        std::cout << field[i-center][j-center] << " ";
                    }
                    std::cout << std::endl;
                }

                if (bestStackSize && currentStack.size() >= bestStackSize)
                    return false;
            }
        }
    }

    return true;
}

/// Find best fragmentation for specify diag
/// \param n size of table
/// \param center center of table
/// \param leftCornerShift left corner coord of diag
/// \return list of squares
std::vector<Square> findNextFragmentationForDiag(int n, int center, int leftCornerShift)
{
    int** field = new int* [n - center + 1];
    for (int i = 0; i < n - center; ++i)
    {
        field[i] = new int[n - center + 1];
    }
    const auto squize_square = [&](Square& square)
    {
        // Bottom
        for (int i = square.leftCorner.x + square.length - 1; i > square.leftCorner.x - 1; --i)
        {
            field[i - center][square.leftCorner.y + square.length - 1 - center] = 0;
        }
```

```
    // Side
    for (int j = square.leftCorner.y + square.length - 2; j > square.leftCorner.y - 1; --j)
    {
        field[square.leftCorner.x + square.length - 1 - center][j - center] = 0;
    }

    square.length -= 1;
};

for (int i = center; i < n; ++i)
{
    for (int j = center; j < n; ++j)
    {
        bool isEmpty =
                (i != j
                 || (i != center and j != center))
            && (i < leftCornerShift || j < leftCornerShift);
        field[i - center][j - center] = isEmpty ? 0 : 1;
    }
}

std::vector<Square> retStack;
std::vector<Square> tmpStack;

while (!tmpStack.empty() || retStack.empty())
{
    if (retStack.empty() || tmpStack.empty() || tmpStack.size() < retStack.size())
    {
        bool is_better = backtrackingIter(n, center, static_cast<int**>(field),
                                          tmpStack, retStack.size());
        if ((is_better && tmpStack.size() < retStack.size()) || retStack.empty())
        {
            retStack = tmpStack;
        }
    }

    while (!tmpStack.empty() && tmpStack.back().length == 1)
    {
        auto deleted = tmpStack.back();
        field[deleted.leftCorner.x - center][deleted.leftCorner.y - center] = 0;
        tmpStack.pop_back();
```

```
        }

        if (!tmpStack.empty() && tmpStack.back().length > 1)
        {
            squize_square(tmpStack.back());
        }
    }

    return retStack;
}

/// Make backtracking fragmentation
/// \param n size of table
/// \return vector of Square
std::vector<Square> makeBacktrackingFragmentation(int n)
{
    auto squares = makeBaseFragmentation(n);
    std::vector<Square> minSquares;
    int center = n / 2;
    for (int i = 2; i < center; ++i)
    {
        int left_corner_shit = n - i;
        std::vector<Square> best_fragmentation_for_diag = {Square(left_corner_shit,
                                                                  left_corner_shit, i)};

        auto backtack_fragmentation = findNextFragmentationForDiag(n, center, left_corner_shit);
        best_fragmentation_for_diag.insert(
            best_fragmentation_for_diag.begin() + 1,
            backtack_fragmentation.begin(),
                                          backtack_fragmentation.end());

        if (minSquares.empty() || best_fragmentation_for_diag.size() < minSquares.size())
        {
            minSquares = best_fragmentation_for_diag;
        }

    }

    squares.insert(squares.begin() + 3, minSquares.begin(), minSquares.end());
    return squares;
}
```

```
/// Print field state
/// \param n size of table
/// \param squares vector of squares
void printDebugSquares(int n, std::vector<Square>& squares)
{
    int field[n][n];

    int square_index = 0;
    for (auto& square : squares)
    {
        for (int y = square.leftCorner.y; y < square.leftCorner.y + square.length; ++y)
        {
            for (int x = square.leftCorner.x; x < square.leftCorner.x + square.length; ++x)
            {
                field[x][y] = square_index;
            }
        }
        ++square_index;
    }

    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            std::cout << field[i][j] << " ";
        }
        std::cout << std::endl;
    }
}

/// Print result and each Square
/// \param n size of table
/// \param squares vector of squares
void printSquares(int n, std::vector<Square>& squares)
{
    std::cout << squares.size() << std::endl;
    for (auto& square : squares)
    {
        std::cout << square << std::endl;
    }

#ifndef NDEBUG
```

```
    print_debug_squares(n, squares);
#endif
}


/// Main function for call
/// \return
int main()
{
    int n;
    std::cin >> n;
    auto fun = makeBacktrackingFragmentation;
    if (n % 2 == 0)
        fun = makeFragmentationForDiv2;
    else if (n % 3 == 0)
        fun = makeFragmentationForDiv3;
    else if (n % 5 == 0)
        fun = makeFragmentationForDiv5;
    else
        fun = makeBacktrackingFragmentation;
    auto start = std::chrono::steady_clock::now();
    auto res = fun(n);
    auto end = std::chrono::steady_clock::now();

    printSquares(n, res);
    std::cout << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).
        count() << "ms" << std::endl;
    return 0;
}
```