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

ОТЧЕТ

По лабораторной работе № 1

по дисциплине «Построение и анализ алгоритмов»

Тема: Поиск с возвратом

Студент гр. 0303 _____ Болкунов В.О.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2022

Цель работы.

Изучение и практическое освоение метода поиска с возвратом (BackTracking).

Основные теоретические положения.

Поиск с возвратом, (backtracking) — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. Как правило, позволяет решать задачи, в которых ставятся вопросы типа: «Перечислите все возможные варианты …», «Сколько существует способов …», «Есть ли способ …», «Существует ли объект…» и т.п.

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.

Выполнение работы

Алгоритм: на каждом шаге рекурсии выполняется поиск первой свободной ячейки квадрата, далее в цикле перебираются размеры квадратов которые туда можно вставить, и для каждого вставленного квадрата запускается новый шаг рекурсии. Для ускорения решения эмпирически подобрана следующая оптимальная начальная конфигурация – в один угол ставится квадрат размером округлённой вверх половины размера начального полотна, а по бокам от него квадраты размером с округлением вниз (см. рис.1). Также для оптимизации решения составные числа разбиваются на минимальный простой делитель и остальную часть (число делённое на

минимальный простой делитель), и алгоритм запускает решение на этом минимальном простом делителе, далее все числа в решении масштабируются обратно (домножаются на оставшуюся от числа часть), частным случаем такой оптимизации является очевидное разбиение квадрата со стороной равной чётному число – результатом будет 4 квадрата со стороной равной половине стороны изначального.

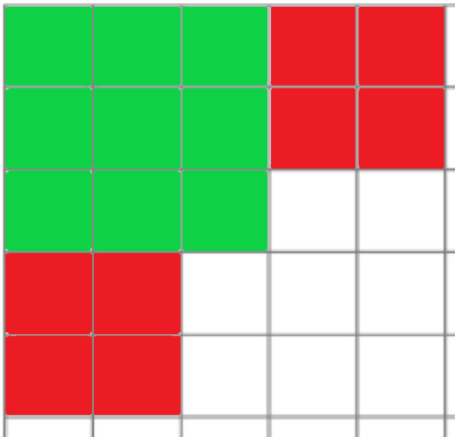

Рисунок 1: Начальная конфигурация

С прямоугольником всё аналогично, но начальная конфигурация отсутствует.

Реализованные структуры и функции

- Макрос отладочного вывода

```
#define dbgPrintf(...) if (DEBUG) printf(__VA_ARGS__);
```

- Структура хранения квадрата (координаты и размер)

```
typedef struct Square {
    int y, x, R;
} Square;
```

- Структура для хранения решения (массив квадратов)

```
typedef struct Solution {
    int mem, size;
    Square *list;
} Solution;
```

- Печать списка квадратов

```
void printSln(Solution sln);
```

- Структура для матрицы

```
typedef struct Matrix {
    int n, m;
    int **arr;
} Matrix;
```

- Печать матрицы

```
void printMat(Matrix mat);
```

- Создание нулевой матрицы n*m

```
Matrix newMat(int n, int m);
```

- Вставка квадрата в матрицу (заполнение квадратного среза числом)

```
void insert(Matrix mat, Square sq, int item);
```

- Функция поиска свободной ячейки в матрице

```
Square findFree(Matrix mat);
```

- Функция проверки возможности вставки квадрата в матрицу

  ```
  int check(Matrix mat, Square sq);
  ```

- Шаг рекурсивного алгоритма

  ```
  void recStep(int N, Solution sln, Matrix mat, Solution *best);
  ```

- Решение базовой задачи

  ```
  Solution solve(int N);
  ```

- Решение для прямоугольников

  ```
  Solution advancedSolve(int N, int M, int *count);
  ```

- Шаг рекурсии для прямоугольников

  ```
  void advancedRecStep(int N, int M, Solution sln, Matrix mat,
                       Solution *best, int *count);
  ```

Тестирование

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	23	13 0 0 12 12 0 11 0 12 11 11 12 2 11 14 5 11 19 4 12 11 1 13 11 3 15 19 1 15 20 3 16 11 7 16 18 2 18 18 5	квадрат

2.	30	4 0 0 15 0 15 15 15 0 15 15 15 15	квадрат
3.	3 4	Size: 6 0 0 2 0 2 2 2 0 1 2 1 1 2 2 1 2 3 1 Count: 4	прямоугольник, (иллюстрация на рис. 2)
4	7 9	Size: 7 0 0 5 0 5 4 4 5 1 4 6 3 5 0 2 5 2 2 5 4 2 Count: 4	прямоугольник
5	11 23	Size: 12 0 0 7 0 7 6 0 13 6 0 19 4 4 19 2 4 21 2 6 7 1 6 8 5 6 13 5 6 18 5 7 0 4 7 4 4 Count: 76	Прямоугольник

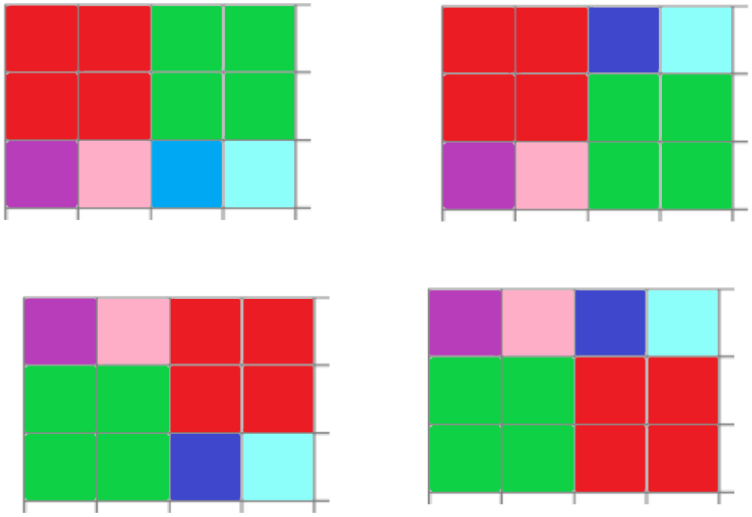

Рисунок 2: Минимальные замощения прямоугольника 3*4

Выводы:

В ходе выполнения работы был исследован переборный метод решения задач: поиск с возвратом. На основе метода был составлен и реализован алгоритм решающий задачу замощения полотна квадратными плитками.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл Backtracking.h

```
#ifndef BACKTRACKING_H
#define BACKTRACKING_H

#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <stdarg.h>

// Флаг отладки
extern int DEBUG;

// Отладочный вывод
#define dbgPrintf(...) if (DEBUG) printf(__VA_ARGS__);

// Квадрат
typedef struct Square {
    int y, x, R;
} Square;

// Список квадратов (решение)
typedef struct Solution {
    int mem, size;
    Square *list;
} Solution;

// Вывод решения
void printSln(Solution sln);

// Целочисленная матрица n на m
typedef struct Matrix {
    int n, m;
    int **arr;
} Matrix;

// Вывод матрицы
void printMat(Matrix mat);

// Конструктор матрицы
Matrix newMat(int n, int m);

// Вставка квадрата в матрицу
void insert(Matrix mat, Square sq, int item);

// Поиск свободной ячейки (сверху-слева)
Square findFree(Matrix mat);

// Проверка возможности вставить квадрат в матрицу
int check(Matrix mat, Square sq);

// Рекурсивный шаг
void recStep(int N, Solution sln, Matrix mat, Solution *best);
```

```
// Базовое решение
Solution solve(int N);

// Решение в случае прямоугольника
Solution advancedSolve(int N, int M, int *count);

// Шаг в случае прямоугольника
void advancedRecStep(int N, int M, Solution sln, Matrix mat,
Solution *best, int *count);

#endif //BACKTRACKING_H
```

Файл Backtracking.c

```
#include "BackTracking.h"

// Флаг отладки
int DEBUG = 0;

// Вывод решения
void printSln(Solution sln) {
    dbgPrintf("Size: %d {\n", sln.size);
    for (int i = 0; i < sln.size; ++i) {
        dbgPrintf("%d, %d, %d\n", sln.list[i].y, sln.list[i].x,
sln.list[i].R);
    }
    dbgPrintf("}\n");
}

// Вывод матрицы
void printMat(Matrix mat) {
    for (int i = 0; i < mat.n; ++i) {
        for (int j = 0; j < mat.m; ++j) {
            dbgPrintf("%d ", mat.arr[i][j]);
        }
        dbgPrintf("\n");
    }
}

// Конструктор матрицы
Matrix newMat(int n, int m) {
    Matrix mat = {n, m, malloc(sizeof(int *) * n)};
    for (int i = 0; i < n; ++i)
        mat.arr[i] = calloc(m, sizeof(int));
    return mat;
}

// Вставка квадрата в матрицу
void insert(Matrix mat, Square sq, int item) {
    for (int y = sq.y; y < sq.y + sq.R; ++y)
        for (int x = sq.x; x < sq.x + sq.R; ++x)
            mat.arr[y][x] = item;
}

// Поиск свободной ячейки (сверху-слева)
Square findFree(Matrix mat) {
    for (int i = 0; i < mat.n; ++i)
```

```
        for (int j = 0; j < mat.m; ++j)
            if (mat.arr[i][j] == 0)
                return (Square) {i, j, 0};
    // Если не нашлось
    return (Square) {-1, -1, 0};
}

// Проверка возможности вставить квадрат в матрицу
int check(Matrix mat, Square sq) {
    // Проверка вместимости квадрата в матрицу
    if (sq.x + sq.R > mat.m || sq.y + sq.R > mat.n)
        return 0;
    for (int i = sq.y; i < sq.y + sq.R; ++i) {
        for (int j = sq.x; j < sq.x + sq.R; ++j) {
            if (mat.arr[i][j] != 0)
                return 0;
        }
    }
    return 1;
}

// Рекурсивный шаг
void recStep(int N, Solution sln, Matrix mat, Solution *best) {
    // Завершение неоптимальной ветки решения (если решение уже
длиннее лучшего)
    if (sln.size >= best->size)
        return;
    // Свободная клетка
    Square cell = findFree(mat);
    // Если не найдена
    if (cell.y == -1) {
        // Новое кратчайшее решение
        if (sln.size < best->size) {
            dbgPrintf("New record - ");
            printSln(sln);
            memcpy(best->list, sln.list, sizeof(Square) *
sln.size);
            best->size = sln.size;
        }
    } else {
        // Перебор квадратов которые можно вставить
        for (cell.R = N / 2 + N % 2; cell.R > 0; --cell.R) {
            if (check(mat, cell)) {
                // Вставка
                insert(mat, cell, 1);
                sln.list[sln.size++] = cell;
                // Новый шаг в глубину
                recStep(N, sln, mat, best);
                // Откат к предыдущему состоянию (чтобы не
создавать новую матрицу и массив решений на каждом шаге)
                insert(mat, cell, 0);
                sln.size--;
            }
        }
    }
}
```

```
// Базовое решение
Solution solve(int N) {
    // Поиск минимального простого делителя, далее алгоритм будет
работать с ним
    // А решение просто умножится на оставшийся множитель (remain)
    int minDiv = 2;
    while (N % minDiv != 0)
        minDiv++;
    int remain = N / minDiv;
    N = minDiv;
    dbgPrintf("min divisor: %d\n", minDiv);

    // Лучшее базовое решение (все единичные квадраты)
    Solution best = {N * N, N * N, malloc(sizeof(Square) * N * N)};
    for (int i = 0; i < best.size; ++i) {
        best.list[i] = (Square) {i / N, i % N, 1};
    }

    // Буферное решение
    Solution sln = {N * N, 0, malloc(sizeof(Square) * N * N)};
    // Матрица
    Matrix mat = newMat(N, N);

    // Начальная оптимизация
    int half = N / 2 + N % 2;
    sln.list[sln.size++] = (Square) {0, 0, half};
    sln.list[sln.size++] = (Square) {half, 0, half - 1};
    sln.list[sln.size++] = (Square) {0, half, half - 1};
    for (int i = 0; i < sln.size; ++i) {
        insert(mat, sln.list[i], 1);
    }

    dbgPrintf("Initial matrix:\n");
    printMat(mat);

    // запуск рекурсии
    recStep(N, sln, mat, &best);

    // Обратное масштабирование решения
    if (remain != 1) {
        for (int i = 0; i < best.size; ++i) {
            best.list[i].x *= remain;
            best.list[i].y *= remain;
            best.list[i].R *= remain;
        }
    }
    free(sln.list);
    free(mat.arr);
    return best;
}

// Решение в случае прямоугольника
Solution advancedSolve(int N, int M, int *count) {
    // Аналогично базовое лучшее решение - все квадраты 1x1
    Solution best = {N * M, N * M, malloc(sizeof(Square) * N * M)};
    for (int i = 0; i < best.size; ++i) {
        best.list[i] = (Square) {i / N, i % N, 1};
```

```
    }

    // Буферное решение
    Solution sln = {N * M, 0, malloc(sizeof(Square) * N * M)};
    // Матрица
    Matrix mat = newMat(N, M);
    // Количество вариантов минимального замощения
    *count = 0;
    // запуск рекурсии
    advancedRecStep(N, M, sln, mat, &best, count);
    free(sln.list);
    free(mat.arr);
    return best;
}

// Шаг в случае прямоугольника
void advancedRecStep(int N, int M, Solution sln, Matrix mat,
Solution *best, int *count) {
    // Завершение неоптимальной ветки решения (если решение уже
длиннее лучшего)
    if (sln.size > best->size)
        return;
    // Свободная клетка
    Square cell = findFree(mat);
    // Если не найдена
    if (cell.y == -1) {
        // Инкремент количества минимальных замощений
        if (sln.size == best->size)
            (*count) ++;
        // Новое кратчайшее решение
        if (sln.size < best->size) {
            (*count) = 1;
            dbgPrintf("New record - ");
            printSln(sln);
            memcpy(best->list, sln.list, sizeof(Square) *
sln.size);
            best->size = sln.size;
        }
    } else {
        int min = (N > M ? M : N);
        // Перебор квадратов которые можно вставить
        for (cell.R = min - 1; cell.R > 0; --cell.R) {
            if (check(mat, cell)) {
                // Вставка
                insert(mat, cell, 1);
                sln.list[sln.size++] = cell;
                // Новый шаг в глубину
                advancedRecStep(N, M, sln, mat, best, count);
                // Откат к предыдущему состоянию (чтобы не
создавать новую матрицу и массив решений на каждом шаге)
                insert(mat, cell, 0);
                sln.size--;
            }
        }
    }
}
```

Файл main.c

```
#include "BackTracking.h"

int main() {
    int n;
    scanf("%d", &n);
    Solution sln = solve(n);
    printf("%d\n", sln.size);
    for (int i = 0; i < sln.size; ++i) {
        printf("%d %d %d\n", sln.list[i].y, sln.list[i].x,
sln.list[i].R);
    }
}
```

Файл rects.c

```
#include "BackTracking.h"

int main() {
    DEBUG = 1;
    int n, m;
    scanf("%d %d", &n, &m);
    int c;
    Solution sln = advancedSolve(n, m, &c);
    printf("\nAnswer\nSize: %d\n", sln.size);
    for (int i = 0; i < sln.size; ++i) {
        printf("%d %d %d\n", sln.list[i].y, sln.list[i].x,
sln.list[i].R);
    }
    printf("Count: %d", c);
}
```