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. 9382 _____ Михайлов Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы

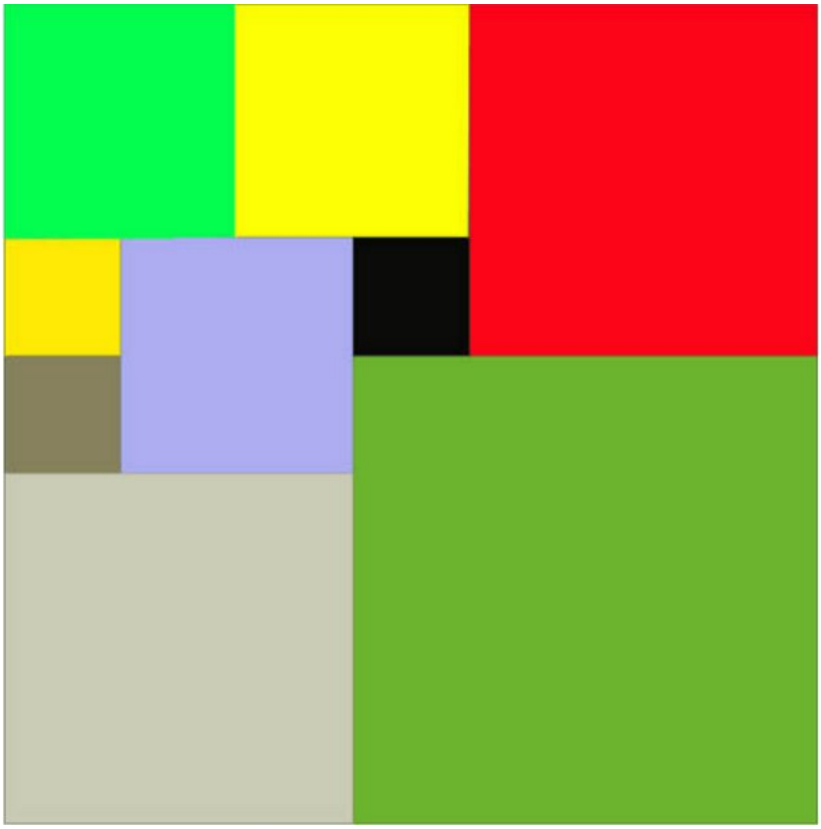
Применить на практике алгоритм поиска с возвратом для заполнения прямоугольника минимальным количеством квадратов, со сторонами, меньшими ребер прямоугольника.

Вар. 4р. Рекурсивный бэктрекинг. Расширение задачи на прямоугольные поля, рёбра квадратов меньше рёбер поля. Подсчёт количества вариантов покрытия минимальным числом квадратов.

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 x 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 KK, задающее минимальное количество обрезков (квадратов), из которых можно построить столешницу (квадрат) заданного размера NN. Далее должны идти KK строк, каждая из которых должна содержать три целых числа x, yx,y и ww, задающие координаты левого верхнего угла ($1 \le x,y \le N$) 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

Теоретические сведения

Бэктрекинг (поиск с возвратом) – это общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. 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 вычисления стараются организовать таким образом, чтобы как можно раньше отбросить заведомо не оптимальные варианты, как правило, это позволяет значительно уменьшить время нахождения решения.

Описание алгоритма

Алгоритм оптимального разбиения поля основан на поиске с возвратом: квадраты ставятся подряд от угла, при этом выбирается наибольший возможный размер, пока поле не заполнится. Если понадобилось меньше квадратов, чем на прошлых итерациях, результат сохраняется. Затем алгоритм возвращается назад до тех пор, пока не встретит квадрат размера больше 1, стирает его и ставит вместо него квадрат меньшего на один размера. Алгоритм работает до тех пор, пока все необходимые расстановки не будут проверены.

Поиск оптимального покрытия прямоугольника сводится к двум этапам: разбиению основной части – квадрата с наибольшей стороной, который можно вписать в заданный прямоугольник и разбиению оставшейся части.

На первом этапе создается массив N на N, (N - размер меньшего ребра прямоугольника), в котором содержатся данные о разбиении – в занятую квадратом ячейку помещается значение его стороны. Алгоритм находит свободное место для вставки и добавляет туда квадраты всех возможных размеров. Если поле оказывается заполненным, то количество квадратов на нем сравнивается с найденным лучшим разбиением, которое хранится в

эталонном массиве, и если новое разбиение оказалось более удачным, то оно заменяет предыдущий найденный оптимальный результат.

После того, как был разбит основной фрагмент, оставшаяся часть прямоугольника последовательно делится на квадраты с наибольшей возможной стороной (в зависимости от ширины или высоты оставшихся фрагментов). Если оставшаяся часть кратна квадрату, разбитому на первом этапе, то в результирующий массив копируются данные его квадрирования, но с соответствующим смещением координат. После второго этапа данные о получившемся оптимальном разбиении объединяются и выводятся пользователю.

Использованные оптимизации

- Массив поля изначально можно заполнить на 3/4 тремя квадратами размеров N/2 , N/2 - 1 соответственно.
- Поскольку массив заполнен на 3/4, то поиск свободной клетки, куда можно поместить квадрат, допустимо осуществлять только в оставшейся 1/4 квадрата.
- Квадрат с четной стороной имеет постоянное решение – 4 квадрата. Поэтому можно не осуществлять перебор для таких квадратов, а сразу выдать ответ.
- Сжатие квадрата. Квадрат с размером N, можно сжать до размера значения наименьшего простого делителя числа N.

Функции и структуры данных

struct fragment – структура, содержащая координаты и флаг, что данный фрагмент не занят

class SquareDivision – класс, реализующий разбиение квадратной области

Приватные поля класса

vector<vector<size_t>> squareDivArr – двумерный массив для хранения

текущих разбиений

vector<vector<size_t>> optimalSquareDivArr – двумерный массив для хранения оптимального разбиения

size_t size – размер стороны квадрата

size_t currDivCount – счетчик текущих разбиений

size_t optimalDivCount – счетчик оптимальных разбиений

size_t compression – масштаб квадрата, в случае возможности применения оптимизации

Публичные поля, содержащие выходные результаты

vector<vector<size_t>> finalData – двумерный массив с оптимальным разбиением

size_t totalDivisions – количество разбиений

Методы класса

SquareDivision(size_t size_) – конструктор класса, в котором производится инициализация массивов для хранения разбиений и оптимизация входных данных

void createOptimalDivision() – основная логика класса (публичный метод)

void insertDivision(size_t x, size_t y, size_t size_) – добавление нового квадрата с заданными координатами и размером

bool checkDivision(size_t x, size_t y, size_t size_) – проверка возможности разбить фрагмент

fragment findEmpty(size_t x, size_t y) – поиск свободного фрагмента поля

void removeDivision(size_t x, size_t y, size_t size_, bool optimal) – удаление разбиения

void prepareDivSetup() – начальное разбиение и обработка специальных случаев

void updateOptimalDivision(size_t x, size_t y, int deep = 0) – основной алгоритм квадрирования

void saveOptimalDivString() – сохранение получившегося оптимального разбиения квадрата в выходной массив

class RectangleDivision – класс, реализующий разбиение прямоугольной области

Приватные поля

size_t width, height – содержат размеры прямоугольника

Публичные методы

RectangleDivision(size_t width, size_t height) – конструктор класса

void createDivision() – главная логика класса, инициализирует первый этап разбиения основного квадрата, производит разбиение оставшейся части, объединяет и выдает получившиеся результаты.

Оценка сложности алгоритма по времени

В виду того, что в программе используется несколько оптимизаций, произведем оценку сложности алгоритма сверху. N – длина стороны квадрата. Имеется N^2 свободных клеток, также N размеров квадрата, которые требуется перебирать. Таким образом, сложность алгоритма по времени будет составлять O((N^2)! * N^N).

Оценка сложности алгоритма по памяти

Так как массив квадрата при каждом рекурсивном проходе копируется, то следует взять максимальное количество единичных квадратов в матрице, которое составляет N*N и умножить на количество рекурсивных проходов. При этом скопированные варианты удаляются, поэтому за максимум можно считать проход по матрице – N*N. Следовательно сложность алгоритма по памяти – O(N^4).

Демонстрация работы

Ввод	Вывод
5, 4	8 1 1 2 1 3 2 3 1 2 3 3 2 5 1 1 5 2 1 5 3 1 5 4 1
13, 11	18 1 1 6 1 7 5 6 7 3 6 10 2 7 1 5 7 6 1 8 6 1 8 10 1 8 11 1 9 6 3 9 9 3 12 1 2 12 3 2 12 5 2 12 7 2 12 9 2 12 11 1 13 11 1

14, 9	12 1 1 6 1 7 3 4 7 3 7 1 3 7 4 3 7 7 3 10 1 5 10 6 4 14 6 1 14 7 1 14 8 1 14 9 1
8, 19	13 1 1 4 1 5 4 5 1 4 5 5 4 1 9 4 1 13 4 5 9 4 5 13 4 1 17 3 4 17 3 7 17 2 7 19 1 8 19 1
20, 13	19 1 1 7 1 8 6 7 8 2 7 10 4 8 1 6 8 7 1 9 7 3 11 10 1 11 11 3 12 7 2 12 9 2 14 1 7

	14 8 6 20 8 1 20 9 1 20 10 1 20 11 1 20 12 1 20 13 1

Выводы

В результате выполнения работы был изучен и реализован алгоритм поиска с возвратом, на его основе была создана программа поиска оптимального заполнения прямоугольника минимальным количеством квадратов.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
/**
 * Подсчёт количества вариантов покрытия прямоугольника
 * минимальным числом квадратов
 * с использованием рекурсивного бэктрекинга.
 * Рёбра квадратов меньше рёбер поля.
 */

#include <iostream>
#include <vector>

#define DEBUG

struct fragment
{
    size_t x;
    size_t y;
    bool isEmpty;
};

class SquareDivision
{
private:
    std::vector<std::vector<size_t>> squareDivArr;
    std::vector<std::vector<size_t>> optimalSquareDivArr;
    size_t size{};
    size_t currDivCount{};
    size_t optimalDivCount{};
    size_t compression{};

public:
    std::vector<std::vector<size_t>> finalData;
    size_t totalDivisions = 0;

    explicit SquareDivision(size_t size_) : size(size_), optimalDivCount(size * size)
    {
        // Инициализация массивов для хранения разбиений
        squareDivArr.resize(size);
        // Оптимизация квадрата (если это возможно, он масштабируется)
        for (auto i = size_; i > 0; --i) {
            if (size % i == 0 && size != i) {
                size /= i;
                compression = i;
                break;
            }
        }
        for (auto &i : squareDivArr) {
            i.resize(size_);
        }
        optimalSquareDivArr = squareDivArr;
    }

    void createOptimalDivision()
```

```
    {
        prepareDivSetup();
#ifdef DEBUG
        std::cout << "=== Завершение первоначальной расстановки ===" <<
std::endl;
#endif
        updateOptimalDivision(size / 2, size / 2 + 1);
        saveOptimalDivString();
    }

private:

    // Добавление нового квадрата
    void insertDivision(size_t x, size_t y, size_t size_)
    {
        for (auto i = x; i < x + size_; ++i) {
            for (auto j = y; j < y + size_; ++j) {
                squareDivArr[i][j] = size_;
            }
        }
        currDivCount++;
#ifdef DEBUG
        std::cout << "~~~~~ Добавление квадрата № " << currDivCount << " ("
<< y + 1 << ", " << x + 1 << ", " << size_ << ") ~~~~~" << std::endl;
        if (compression > 1)
            std::cout << "Применен масштаб: " << compression << std::endl;
        for (auto & i : squareDivArr) {
            for (auto j : i) {
                std::cout << j << "  ";
            }
            std::cout << std::endl;
        }
#endif
    }

    // Проверка возможности разбить фрагмент
    bool checkDivision(size_t x, size_t y, size_t size_) {
        if (x + size_ > size || y + size_ > size)
            return false;
        for (auto i = x; i < x + size_; i++) {
            for (auto j = y; j < y + size_; j++) {
                if (squareDivArr[i][j] != 0)
                    return false;
            }
        }
        return true;
    }

    // Поиск свободного фрагмента поля
    fragment findEmpty(size_t x, size_t y) {
        while (squareDivArr[x][y] != 0) {
            if (y == size - 1) {
                if (x == size - 1) {
                    return {x, y, false};
                } else {
                    x++;
```

```
                y = size / 2;
                continue;
            }
        }
        y++;
    }
    return {x, y, true};
}

// Удаление разбиения
void removeDivision(size_t x, size_t y, size_t size_, bool optimal)
{
#ifdef DEBUG
    if (!optimal)
        std::cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << std::endl <<
"Удаляем квадрат (" << y + 1 << ", " << x + 1 << ", " << size_ << ")" <<
std::endl;
#endif
    for (auto i = x; i < x + size_; ++i)
        for (auto j = y; j < y + size_; ++j) {
            if (optimal)
                optimalSquareDivArr[i][j] = 0;
            else
                squareDivArr[i][j] = 0;
        }
    if (!optimal)
        currDivCount--;
}

// Начальное разбиение и обработка специальных случаев
void prepareDivSetup()
{
    size_t halfSize = size / 2;
    if (size % 2 == 0) {
        insertDivision(0, 0, halfSize);
        insertDivision(0, halfSize, halfSize);
        insertDivision(halfSize, 0, halfSize);
        insertDivision(halfSize, halfSize, halfSize);
        currDivCount = 4;
        optimalDivCount = currDivCount;
        optimalSquareDivArr = squareDivArr;
    } else {
        insertDivision(0, 0, halfSize + 1);
        insertDivision(0, halfSize + 1, halfSize);
        insertDivision(halfSize + 1, 0, halfSize);
    }
}

// Основной алгоритм квадрирования
void updateOptimalDivision(size_t x, size_t y, int deep = 0)
{
    fragment f;
    if (currDivCount >= optimalDivCount)
        return;
    for (auto n = size / 2; n > 0; --n) {
        if (checkDivision(x, y, n)) {
```

```
            insertDivision(x, y, n);
            f = findEmpty(x, y);
            if (f.isEmpty) {
               updateOptimalDivision(f.x, f.y, deep++);
            } else {
               if (currDivCount < optimalDivCount) {
                  optimalSquareDivArr = squareDivArr;
                  optimalDivCount = currDivCount;
               }
               removeDivision(x, y, n, false);
               return;
            }
            removeDivision(x, y, n, false);
         }
      }
   }

   // Сохранение получившегося оптимального разбиения квадрата в выходной
массив
   void saveOptimalDivString()
   {
      for (auto i = 0; i < size; ++i) {
         for (auto j = 0; j < size; ++j) {
            if (optimalSquareDivArr[i][j] != 0) {
               finalData.push_back({i * compression + 1, j * compression + 1,
optimalSquareDivArr[i][j] * compression});
               removeDivision(i, j, optimalSquareDivArr[i][j], true);
            }
         }
      }
      totalDivisions = optimalDivCount;
   }
};

class RectangleDivision
{
   size_t width, height;

public:

   RectangleDivision(size_t width, size_t height)
   {
      this->width = width;
      this->height = height;
   }

   void createDivision() const
   {
      size_t sqX = 1, sqY = 1; // координаты текущего квадрата
      size_t nWidth = width, nHeight = height; // размеры неразбитой области
      size_t totalDivisions = 0;
      std::vector<std::vector<size_t>> finalData;

      // Квадрирование основной части прямоугольника - квадрата с мак-
симально возможной стороной, который можно вписать в прямоугольник
      size_t size = nWidth < nHeight ? nWidth : nHeight;
```

```
        auto *cutter = new SquareDivision(size);
        cutter->createOptimalDivision();
        totalDivisions += cutter->totalDivisions;
        finalData.insert(finalData.end(), cutter->finalData.begin(), cutter-
>finalData.end());

        if (nWidth != nHeight) {
            // Если заданная фигура - прямоугольник, разбиваем оставшуюся часть
на квадраты с максимально возможной стороной
            while (size > 0) {
                if (nWidth > nHeight) {
                    // разбиение по горизонтали
                    sqX += size;
                    nWidth -= size;
                } else {
                    // разбиение по вертикали
                    sqY += size;
                    nHeight -= size;
                }
                if (sqX > width || sqY > height)
                    break;
                size = nWidth > nHeight ? nHeight : nWidth;
                if (size == width || size == height) {
                    // если оставшаяся часть кратна квадрату с максимально возмож-
ной стороной, копируются данные первого квадрирования со смещением ко-
ординат
                    for (auto &d: cutter->finalData) {
                        finalData.push_back({d[0] + sqX - 1, d[1] + sqY - 1, d[2]});
                    }
                    totalDivisions += cutter->totalDivisions;
                } else {
                    finalData.push_back({sqX, sqY, size});
                    totalDivisions++;
                }
            }
        }

        delete cutter;

#ifdef DEBUG
        std::cout << "~~~~~ Оптимальное покрытие прямоугольника ~~~~~"
<< std::endl;
        std::vector<std::vector<size_t>> rectArr;
        rectArr.resize(height);
        for (auto &a : rectArr) {
            a.resize(width);
        }
        for (auto &data: finalData) {
            for (auto i = data[1] - 1; i < data[1] + data[2] - 1; ++i) {
                for (auto j = data[0] - 1; j < data[0] + data[2] - 1; ++j) {
                    rectArr[i][j] = data[2];
                }
            }
        }
        for (auto & i : rectArr) {
            for (auto j : i) {
```

```
            std::cout << j << "  ";
        }
        std::cout << std::endl;
    }
#endif

    std::cout << "Количество вариантов оптимального покрытия квадратами:
" << std::endl << totalDivisions << std::endl;
    std::cout << "Финальный результат:" << std::endl;
    for (auto &data: finalData) {
        std::cout << data[0] << " " << data[1] << " " << data[2] << std::endl;
    }
  };

};

int main()
{
  size_t width, height;
  std::cout << "Введите размеры прямоугольника" << std::endl;
  std::cout << "Ширина: ";
  std::cin >> width;
  std::cout << "Высота: ";
  std::cin >> height;
  if (height < 2 || width < 2) {
    std::cout << "Размер ребра прямоугольника не может быть меньше 2 по
условиям задачи" << std::endl;
    return 0;
  }

  auto *cutter = new RectangleDivision(width, height);
  cutter->createDivision();
  delete cutter;

  return 0;
}
```