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. 9383 _____ Ноздрин В.Я.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения квадрата со стороной N минимальным количеством квадратов со сторонами размера от 1 до N-1.

Задание. Вариант – 1и(итеративный бэктрекинг).

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).

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 ($2 \le N \le 40$).

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).

Основные теоритические положения.

Backtracking – метод решения задачи полного перебора. Заключается в рекурсивном (или итеративном) переборе всевозможных расширений некоторого частичного решения до тех пор, пока не будет найдено полное решение. Основная идея метода заключается в том, что в процессе перебора, если находится частичное решение, которое не удовлетворяет необходимым условиям, происходит возврат к предыдущему частичному решению, и берется альтернативное его расщирение.

Описание алгоритма.

Для решения задачи был реализован (итеративный) алгоритм поиска с возвратом. Выполняется полный перебор заполнений квадрата, выбирая только заполнения с наименьшим количеством квадратов в заполнении. Так как перебор полный, алгоритм подходит для решения данной задачи.

Реализация следующая

1. Выполняется проход по матрице, задающей разложение квадрата. Выбирается первая свободная ячейка и в нее (и в клетки вокруг нее) помещается максимально возможный квадрат, который также записывается в стек. Стек совместно с матрицей столешницы хранят частычные решения. Шаг 1 повторяется до тех пор, пока столещница не будет полностью заполнена. Послендние свободные ячейки заполняются квадратами размера 1.
2. Выполняется проверка на минимум. Сравнивается количество квадратов текущего разложения с количеством квадратов в минимальном разложении. Если новое разложение имеет меньше квадратов, оно запоминается как новое «минимальное». Также присутствует оптимизация шага 1 – если количество квадратов текущего разложения не меньше минимального, выполняется возврат к предыдущему частичному решению.
3. Возрват к предыдущему частичному решению происходит следующим образом. У текущего решения удаляются все квадраты размера 1, далее если есть еще квадраты не единичного размера, верхний на стеке удаляется и заменяется на квадрат со стороной на единицу меньше. Далее выполняется переход к шагу 1. Выход из цикла происходит, когда после удаления квадратов размера 1, стек остается пустым. Это значит, что все возможные заполнения уже перебраны и в минимальном храниться ответ.

Некоторые частные случаи, например, когда N четно, можно решить в общем виде, вообще не пользуясь перебором.

Оценка сложности алгоритма.

Так как алгоритм переберает все возможные заполнения квадрата квадратами, сложность данного алгоритма будет **$O(e^n)$.**

Описание функций и структур данных.

struct Square – структура данных, хранящая информацию о квадрате в формате, требуемом условием задачи. Структура создана для удобства вывода ответа.

class Squaring – класс1, реализующий основной алгоритм поиска с возвратом. Поля класса содержат информацию о текущем состоянии, то есть частичном решении, а также о наилучшем решении. Состояние состоит из квадратной матрицы и стека структур Square. Класс содержит следующие методы:

void eval() – основной метод, выполняющий все работу. Загружает начальное решение и итеративно выполняет поиск с возвартом.

bool backtrack() – метод, реализующий возврат к предыдущему частичному решению. Если возврат невозможен, то все решения перебраны и метод возвращает false.

void run() – метод, являющийся оберткой для работы с классом. Вызывает метод eval для вычисления решения и печатает решение, вызывая метод printResult().

bool placeSquare(i,j,size) – метод, помещающий квадрат заданного размера в соответствующие координаты. По умолчанию, если не передавать размер, устанавливает его максимально возможным (так, чтобы квадрат поместился).

void baseCase1() – метод, задающий начальное решение.

bool complete() – метод, дополняющий частичное решение до полного решения. Если в процессе дополнения стало ясно, что текущее решение будет хуже текущего лучшего решения, метод завершает работу и возвращает false. В противном случае возвращается true.

void pop() – метод, удаляющий последний добавленный квадрат.

Также для функций написано тестирование с помощью библиотеки Catch2. Для файлов тестирования и основной программы написан Makefile.

```
ice-jack@ice-pc ~/P/p/N/lab1 (Nozdrinlab1)> ./run_tests
===============================================================================
All tests passed (6 assertions in 3 test cases)
```

Рисунок 1 – Демонстрация корректного отрабатывания тестирования.

Демонстрация работы.

```
ice-jack@ice-pc ~/P/p/N/lab1 (Nozdrinlab1)> ./lab1.out
7
9
6 6 2
6 4 2
5 7 1
5 4 1
5 1 3
4 7 1
4 5 2
1 5 3
1 1 4
ice-jack@ice-pc ~/P/p/N/lab1 (Nozdrinlab1)> ./lab1.out
13
11
12 9 2
12 7 2
11 11 3
11 10 1
9 7 3
8 7 1
8 1 6
7 10 4
7 8 2
1 8 6
1 1 7
```

Выводы.

Применен на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов. Написана программа, реализующая алгоритм заполнения квадрата минимальным кол-вом квадратов меньшей стороны с помощью поиска с возвратом.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
#include <iostream>
#include "Squaring.h"

int main() {
  int N;
  if ((std::cin >> N) && (N >=2)) {
    Squaring squaring(N);
    squaring.run();
  } else {
    std::cout << "Bad input. (enter N >= 2)\n";
  }
  return 0;
}
```

Файл Squaring.h:

```
#ifndef SQUARING_H
#define SQUARING_H

#include <iostream>
#include <vector>
#include <stack>
#include <utility>

struct Square {
    int m_x = 0;
    int m_y = 0;
    int m_size = 0;
    Square(int x, int y, int size): m_x(x), m_y(y), m_size(size) {};
};

class Squaring {
 public:
  explicit Squaring(int size);
  int getMaxSquareSize(int i, int j);
  bool placeSquare(int i, int j, int size=0);
  void baseCase1();
  void eval();
  void printResult();
  void printSquares(const std::stack<Square>&) const;
  friend std::ostream& operator<<(std::ostream& os, Squaring& squaring);
  bool complete();
  void pop();
  bool backtrack();
  void run();

  std::vector<std::vector<int>> getCurrentSquaring();
  void setCurrentSquaring(std::vector<std::vector<int>> squaring);
  [[nodiscard]] int getCounter() const;

 private:
```

```
  int                             m_size;
  std::vector<std::vector<int>> m_current;
  std::vector<std::vector<int>> m_best;
  int                             m_counter = 0;
  int                             m_minimalSquaring = 0;
  std::stack<Square>              m_squares;
  std::stack<Square>              m_squares_best;
  int                             multiplier = 1;
};

#endif // SQUARING_H
```

Файл Squaring.cpp:

```
#include "Squaring.h"

#define DEBUG 0

Squaring::Squaring(int size) {
  this->m_size = size;
  m_current.resize(m_size);
  for (int i = 0; i < m_size; i++) {
    m_current[i].resize(m_size);
    for (int j = 0; j < m_size; j++)
      m_current[i][j] = 0;
  }
}
int Squaring::getMaxSquareSize(int i, int j) {
  int maxSize;
  for (maxSize = 1; maxSize < std::min(m_size-i, m_size-j); maxSize++) {
    for (int k = 0; k < maxSize; k++)
      if (m_current[i + k][j + maxSize] != 0) {
        return maxSize;
      }
    for (int k = 0; k <= maxSize; k++)
      if (m_current[i + maxSize][j + k] != 0) {
        return maxSize;
      }
  }
  return (maxSize < m_size) ? maxSize : 0;
}
bool Squaring::placeSquare(int i, int j, int size) {
  if ((i < 0) || (j < 0) || (i >= m_size) || (j >= m_size))
    return false;
  if (m_current[i][j] != 0)
    return false;
  auto maxSize = getMaxSquareSize(i, j);
  if (size == 0) {
    size = maxSize;
//    std::cerr << "placeSquare: size = " << size << "\n";
  }
//  else if (size > maxSize) {
//    return false;
//  }
  m_counter++;
  for (int k_i = 0; k_i < size; k_i++)
```

```
    for (int k_j = 0; k_j < size; k_j++)
      m_current[i+k_i][j+k_j] = m_counter;
  m_squares.push(Square(i, j, size));
  return true;
}
std::vector<std::vector<int>> Squaring::getCurrentSquaring(){
  return m_current;
};
void Squaring::setCurrentSquaring(std::vector<std::vector<int>> squaring)
{
  m_current = std::move(squaring);
};
int Squaring::getCounter() const {
  return m_counter;
};
void Squaring::eval() {
  this->baseCase1();
  while (true) {
    if (this->complete())
      if (m_counter < m_minimalSquaring) {
        m_squares_best = m_squares;
        m_best = m_current;
        m_minimalSquaring = m_counter;
      }
    if (!backtrack())
      break;
  }
}
void Squaring::run() {
  if (m_size % 2 == 0) {
    int a = m_size / 2;
    std::cout << "4\n1 1 " << a << '\n';
    std::cout << a + 1 << ' ' << 1 << ' ' << a << '\n';
    std::cout << 1 << ' ' << a + 1 << ' ' << a << '\n';
    std::cout << a + 1 << ' ' << a + 1 << ' ' << a << std::endl;
    return;
  }
  if (m_size % 3 == 0) {
    multiplier = m_size/3;
    m_size = 3;
  } else if (m_size % 5 == 0) {
    multiplier = m_size/5;
    m_size = 5;
  }
  eval();
  printResult();
}
void Squaring::printResult() {
  std::cout << m_minimalSquaring << '\n';
  printSquares(m_squares_best);
}
void Squaring::printSquares(const std::stack<Square>& squares) const {
  auto copy = squares;
  while (!(copy.empty())) {
    std::cout << copy.top().m_x*multiplier+1 << ' ' <<
copy.top().m_y*multiplier+1 << ' '
              << copy.top().m_size*multiplier << std::endl;
    copy.pop();
```

```
  }
}
void Squaring::baseCase1() {
  m_counter = 1;
  for (int i = 0; i < m_size-1; i++)
    for (int j = 0; j < m_size-1; j++)
      m_current[i][j] = m_counter;
  m_squares.push(Square(0, 0, m_size-1));

  for (int i = 0; i < m_size-1; i++) {
    m_counter++;
    m_current[i][m_size-1] = m_counter;
    m_squares.push(Square(i, m_size - 1, 1));
  }
  for (int i = 0; i < m_size; i++) {
    m_counter++;
    m_current[m_size-1][i] = m_counter;
    m_squares.push(Square(m_size - 1, i, 1));
  }
  m_minimalSquaring = m_counter;
  m_squares_best = m_squares;
  m_best = m_current;
}
std::ostream& operator<<(std::ostream& os, Squaring& squaring) {
  for (int i = 0; i < squaring.m_size; i++) {
    for (int j = 0; j < squaring.m_size; j++)
      os << squaring.m_current[i][j] << ' ';
    os << std::endl;
  }
//  std::cout << squaring.m_minimalSquaring << '\n';
//  for (int i = 0; i < squaring.m_minimalSquaring; i++) {
//    Square tmp = squaring.m_squares_best.top();
//    squaring.pop();
//    os  << tmp.m_x * squaring.multiplier + 1 << ' '
//        << tmp.m_y * squaring.multiplier + 1 << ' '
//        << tmp.m_size * squaring.multiplier << std::endl;
//  }
  return os;
}
bool Squaring::complete() {
#if DEBUG
  std::cerr << *this;
  std::cerr << "complete...\n";
#endif // DEBUG
  for (int i = 0; i < m_size; i++)
    for (int j = 0; j < m_size; j++) {
      if (m_current[i][j] != 0)
        continue;
      this->placeSquare(i, j);
//      if (this->placeSquare(i, j))
//        j += m_squares.top().m_size;
      if (m_counter >= m_minimalSquaring) {
#if DEBUG
        std::cerr << "bad solution, (>=min)\n";
#endif // DEBUG
        return false;
      }
    }
```

```
#if DEBUG
  std::cerr << *this;
  std::cerr << "complete -> true\n";
#endif // DEBUG
  return true;
}
void Squaring::pop() {
  if (m_squares.empty())
    return;
#if DEBUG
  std::cerr << "popping...\n";
#endif // DEBUG
  Square tmp = m_squares.top();
  for (int i = tmp.m_y; i < tmp.m_y + tmp.m_size; i++)
    for (int j = tmp.m_x; j < tmp.m_x + tmp.m_size; j++)
      m_current[j][i] = 0;
  m_squares.pop();
  m_counter--;
#if DEBUG
  std::cerr << *this;
  std::cerr << "popped\n";
#endif // DEBUG
}
bool Squaring:: backtrack() {
#if DEBUG
  std::cerr << "backtrack...\n";
#endif // DEBUG
  if (m_squares.empty())
    return false;
  while (!(m_squares.empty())) {
    Square tmp = m_squares.top();
    if (tmp.m_size == 1)
      this->pop();
    else
      break;
  }
  if (m_squares.empty())
    return false;
  Square tmp = m_squares.top();
  int x = tmp.m_x;
  int y = tmp.m_y;
  int size = tmp.m_size;
  this->pop();
  placeSquare(x, y, size - 1);
#if DEBUG
  std::cerr << "backtrack->true\n";
#endif // DEBUG
  return true;
}
```

Файл tests.cpp:

```
#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>
#include "../src/Squaring.h"

TEST_CASE( "Base case is correct", "[base case]" ) {
```

```
  Squaring squaring(10);
  squaring.baseCase1();

  std::vector<std::vector<int>> testSquare = {
      {1,  1,  1,  1,  1,  1,  1,  1,  1,  2},
      {1,  1,  1,  1,  1,  1,  1,  1,  1,  3},
      {1,  1,  1,  1,  1,  1,  1,  1,  1,  4},
      {1,  1,  1,  1,  1,  1,  1,  1,  1,  5},
      {1,  1,  1,  1,  1,  1,  1,  1,  1,  6},
      {1,  1,  1,  1,  1,  1,  1,  1,  1,  7},
      {1,  1,  1,  1,  1,  1,  1,  1,  1,  8},
      {1,  1,  1,  1,  1,  1,  1,  1,  1,  9},
      {1,  1,  1,  1,  1,  1,  1,  1,  1,  10},
      {11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
  };
  REQUIRE(squaring.getCurrentSquaring() == testSquare);
}

TEST_CASE( "Backtrack is correct", "[backtrack]" ) {
  Squaring squaring(10);
  squaring.baseCase1();

  squaring.backtrack();
  std::vector<std::vector<int>> testSquare = {
      {1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  };
  REQUIRE(squaring.getCurrentSquaring() == testSquare);

  squaring.backtrack();
  testSquare = {
      {1, 1, 1, 1, 1, 1, 1, 0, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 0, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 0, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 0, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 0, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 0, 0, 0},
      {1, 1, 1, 1, 1, 1, 1, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
      {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
  };
  REQUIRE(squaring.getCurrentSquaring() == testSquare);

  squaring.backtrack();
  squaring.backtrack();
  testSquare = {
      {1, 1, 1, 1, 1, 0, 0, 0, 0, 0},
      {1, 1, 1, 1, 1, 0, 0, 0, 0, 0},
      {1, 1, 1, 1, 1, 0, 0, 0, 0, 0},
```

```
        {1, 1, 1, 1, 1, 0, 0, 0, 0, 0},
        {1, 1, 1, 1, 1, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
    REQUIRE(squaring.getCurrentSquaring() == testSquare);
}

TEST_CASE( "Pop is correct", "[pop]" ) {
    Squaring squaring(10);
    squaring.baseCase1();
    squaring.pop();

    std::vector<std::vector<int>> testSquare = {
        {1,  1,  1,  1,  1,  1,  1,  1,  1,  2},
        {1,  1,  1,  1,  1,  1,  1,  1,  1,  3},
        {1,  1,  1,  1,  1,  1,  1,  1,  1,  4},
        {1,  1,  1,  1,  1,  1,  1,  1,  1,  5},
        {1,  1,  1,  1,  1,  1,  1,  1,  1,  6},
        {1,  1,  1,  1,  1,  1,  1,  1,  1,  7},
        {1,  1,  1,  1,  1,  1,  1,  1,  1,  8},
        {1,  1,  1,  1,  1,  1,  1,  1,  1,  9},
        {1,  1,  1,  1,  1,  1,  1,  1,  1,  10},
        {11, 12, 13, 14, 15, 16, 17, 18, 19, 0},
    };
    REQUIRE(squaring.getCurrentSquaring() == testSquare);

    squaring.pop();
    testSquare = {
        {1,  1,  1,  1,  1,  1,  1,  1,  1, 2},
        {1,  1,  1,  1,  1,  1,  1,  1,  1, 3},
        {1,  1,  1,  1,  1,  1,  1,  1,  1, 4},
        {1,  1,  1,  1,  1,  1,  1,  1,  1, 5},
        {1,  1,  1,  1,  1,  1,  1,  1,  1, 6},
        {1,  1,  1,  1,  1,  1,  1,  1,  1, 7},
        {1,  1,  1,  1,  1,  1,  1,  1,  1, 8},
        {1,  1,  1,  1,  1,  1,  1,  1,  1, 9},
        {1,  1,  1,  1,  1,  1,  1,  1,  1, 10},
        {11, 12, 13, 14, 15, 16, 17, 18, 0, 0},
    };
    REQUIRE(squaring.getCurrentSquaring() == testSquare);
}
```