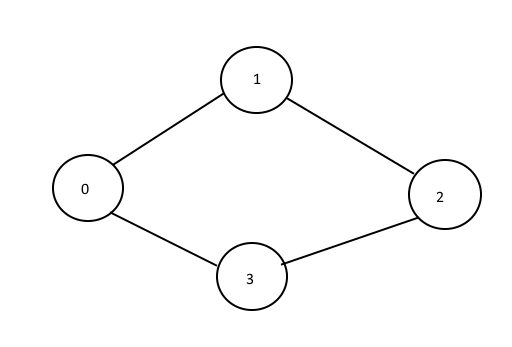
Отчет

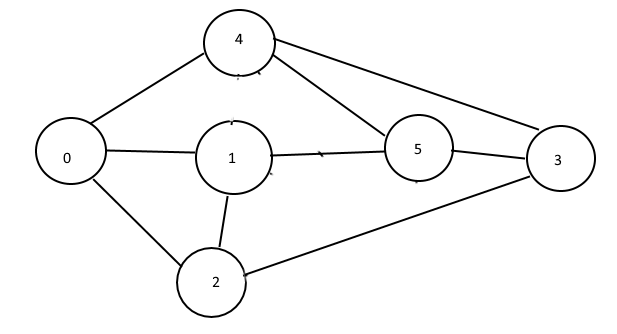
Контрольная работа 1

Задача 12.  
Пусть группа состоит из N человек. В ней каждый имеет (N/2) друзей и не больше K врагов. У одного из них есть книга, которую  
все хотели бы прочитать и потом обсудить с некоторыми из остальных.  
Написать программу, которая:  
1. Находит способ передачи книги таким образом, чтобы она побывала у каждого в точности один раз,переходя только от друга к  
другу и наконец возвратилась к своему владельцу.  
2.Разбивает людей на S групп, где будет обсуждаться книга, таким образом, чтобы вместе с каждым человеком в ту же самую группу вошло не более P его врагов. Примечание: предполагается, что S\*P>=K.

Если кол-во 4, тоже самое и для 5



Для 6 :



Пусть 6 человек. Значит 6/2 = 3 друга у каждого. И не больше K врагов. В данном случае 6-(6/2+1). K=2 – врага. Решено было реализовывать через гамильтонов цикл (задача коммивояжера), когда ищем самый оптимальный цикл, проходя через все вершины графа, не повторяясь. Количество всех связей (рёбер \) определяется как (N/2)^2. Тут 3^2=9

**package** sample;  
  
**public class** Main {  
   
 **public static void** main(String[] args) {  
 *// n - количество узлов, т. е. V* **int** n = 4;  
 **int**[][] graph = {{0, 10, 15, 20},  
 {10, 0, 35, 25},  
 {15, 35, 0, 30},  
 {20, 25, 30, 0}};  
*// int n = 6;  
// int[][] graph = {{0, 1, 1,1, 0, 0},  
// {1, 0,1, 0, 1, 0},  
// {1,1, 0, 0, 0, 1},  
// {1,0, 0, 0, 1, 1},  
// {0, 1, 0, 1, 0, 1},  
// {0, 0, 1, 1, 1, 0}};  
  
 // логический массив для проверки наличия узла  
 // посещали или нет* **boolean**[] wasInNode = **new boolean**[n];  
 *// Пометить 0-й узел как посещенный* wasInNode[0] = **true**;  
 **int** maxValue = Integer.***MAX\_VALUE***;  
 *// Находим минимальный вес гамильтонова цикла* **int** result = *findCycle*(graph, wasInNode, 0, n, 1, 0, maxValue);  
 *// result - результат пути гамильтонова цикла* System.***out***.println(result);  
 }  
  
 *// Функция поиска минимального веса  
 // Гамильтонов цикл* **static int** findCycle(**int**[][] graph, **boolean**[] wasInNode, **int** currPos, **int** n, **int** count, **int** cost, **int** maxValue) {  
  
 *// Если достигнут последний узел и есть ссылка к начальному узлу, т.е. к источнику  
 // сохранить минимальное значение от общей стоимости прохождения и результат  
 // Наконец возвращаемся, чтобы проверить больше возможных значений* **if** (count == n && graph[currPos][0] > 0) {  
 maxValue = Math.*min*(maxValue, cost + graph[currPos][0]);  
 **return** maxValue;  
 }  
 *// BACKTRACKING STEP  
 // Цикл для обхода списка смежности узла currPos и увеличение количества  
 // на 1 и стоимость на график [currPos, i] значение* **for** (**int** i = 0; i < n; i++) {  
 **if** (wasInNode[i] == **false** && graph[currPos][i] > 0) {  
 *// Пометить как посещенные* wasInNode[i] = **true**;  
 maxValue = *findCycle*(graph, wasInNode, i, n, count + 1,  
 cost + graph[currPos][i], maxValue);  
 *// Пометить i-й узел как не посещенный* wasInNode[i] = **false**;  
 }  
 }  
 **return** maxValue;  
 }  
}

Контрольная работа 2

Задача 3.  
Имеется N городов. Для каждой пары городов (I,J) можно построить дорогу, соединяющую эти два города и не заходящие в другие города. Стоимость такой дороги A(I,J). Вне городов дороги не пересекаются.  
Написать алгоритм для нахождения самой дешевой системы дорог, позволяющей попасть из любого города в любой другой. Результаты задавать таблицей B[1:N,1:N], где B[I,J]=1 тогда и только тогда, когда дорогу, соединяющую города I и J, следует строить.

Для реализации примера возьмём такой граф и такие числа

6 9 – узлов и рёбер

0 1 5 // из 0 узла в 1 узел, стоимость 5

0 3 19

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Узел  /узел | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 5 |  | 19 | 6 |  |
| 1 | 5 | 0 | 10 |  | 1 | 9 |
| 2 |  | 10 | 0 | 13 |  |  |
| 3 | 19 |  | 13 | 0 |  | 8 |
| 4 | 6 | 1 |  |  | 0 | 2 |
| 5 |  | 9 |  | 8 | 2 | 0 |

0 4 6

1 2 10

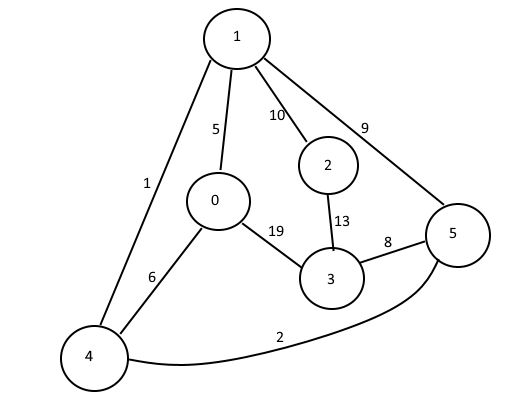
2 3 13

1 4 1

1 5 9

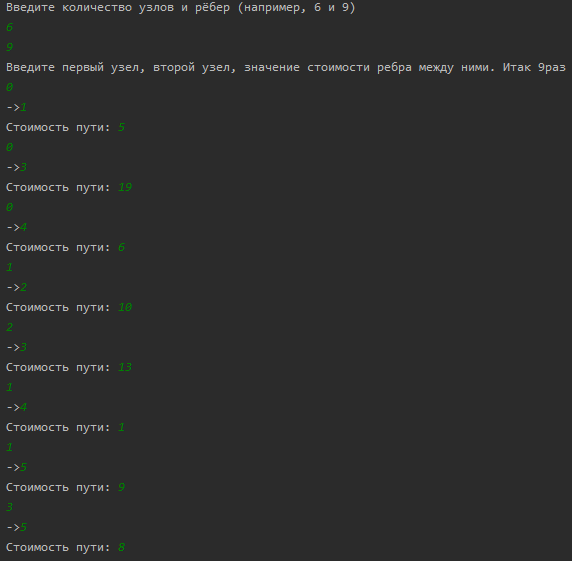
3 5 8

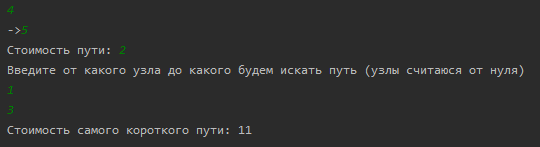
4 5 2



Было принято решение использовать алгоритм Дейкстры для нахождения самого оптимального пути.

Ниже предоставлена работа программы:





**package** sample;  
  
**import** java.io.IOException;  
  
**public class** Main {  
  
 **public static void** main(String[] args) {  
 **try** {  
 Menu.*startGraph*();  
 } **catch** (IOException e) {  
 System.***err***.println(**"Видимо что-то не то ввели: "** + e.getMessage());  
 }  
 }  
}

**package** sample;  
  
**import** java.io.BufferedReader;  
**import** java.io.IOException;  
**import** java.io.InputStreamReader;  
  
**public class** Menu {  
  
 **public static void** startGraph() **throws** IOException {  
 **int** max\_v = 100000;  
  
 *printStartMenu*();  
 BufferedReader reader = *getBufferedReader*();  
 **int** nodes = Integer.*parseInt*(reader.readLine()); *//кол-во узлов* **int** ribs = Integer.*parseInt*(reader.readLine()); *//кол-во рёбер* CityGraph cityGraph = **new** CityGraph(nodes, ribs); *//инициализация графа города* **int**[][] mass = **new int**[nodes][ribs]; *// матрица смежностей* **for** (**int** i = 0; i < nodes; i++) { *// присваиваем элементам матрицы максимальные значения* **for** (**int** j = 0; j < nodes; j++) {  
 mass[i][j] = max\_v;  
 mass[j][i] = max\_v;  
 }  
 }  
  
 *printInputValues*(ribs);  
 **for** (**int** i = 0; i < ribs; i++) { *// заполняем матрицу значениями* **int** firstNodeA = Integer.*parseInt*(reader.readLine());  
 System.***out***.print(**"->"**);  
 **int** secondNodeB = Integer.*parseInt*(reader.readLine());  
 System.***out***.print(**"Стоимость пути: "**);  
 **int** cost = Integer.*parseInt*(reader.readLine());  
  
 mass[firstNodeA][secondNodeB] = cost;  
 mass[secondNodeB][firstNodeA] = cost;  
 }  
   
 *printEnterStartAndEndNodes*();  
 **int** start = Integer.*parseInt*(reader.readLine());  
 **int** end = Integer.*parseInt*(reader.readLine());  
  
 CityGraphService cityGraphService = **new** CityGraphService(mass, cityGraph);  
 System.***out***.println(**"Стоимость самого короткого пути: "** + cityGraphService.findShortWay(start, end));  
 }  
  
 **private static void** printEnterStartAndEndNodes() { *// пример : " 1 3 " : 1 - начало , 3 - конец пути* System.***out***.println(**"Введите от какого узла до какого будем искать путь (узлы считаюся от нуля)"**);  
 }  
  
 **private static void** printInputValues(**int** ribs) {  
 System.***out***.println(**"Введите первый узел, второй узел, значение стоимости ребра между ними. Итак "** + ribs + **"раз"**);  
 }  
  
 **public static void** printStartMenu() {  
 System.***out***.println(**"Введите количество узлов и рёбер (например, 6 и 9)"**);  
 }  
  
 **private static** BufferedReader getBufferedReader() {  
 **return new** BufferedReader(**new** InputStreamReader(System.***in***));  
 }  
  
}

**package** sample;  
  
**public class** CityGraphService {  
  
 **private int**[][] **mass**;  
 **private** CityGraph **cityGraph**;  
  
 **public** CityGraphService() {  
 }  
  
 **public** CityGraphService(**int**[][] mass, CityGraph cityGraph) {  
 **this**.**mass** = mass;  
 **this**.**cityGraph** = cityGraph;  
 }  
  
 **public int** findShortWay(**int** start, **int** end) {  
 **int** W = 99;  
 **int**[] D = **new int**[100];  
 **int**[] used = **new int**[100];  
  
 **if** (start == end) *// если начальный узел является конечным,* **return** 0; *// то путь равен 0  
  
 /\* используем алгоритм Дейкстры \*/* **for** (**int** i = 0; i < **cityGraph**.getNode(); i++) *// заполняем массив, который отвечает за проход узла (если побывали в узле, то присваиваем 1, иначе 0)* used[i] = 0;  
  
 **for** (**int** i = 0; i < **cityGraph**.getNode(); i++)  
 D[i] = **mass**[start][i]; *// массив, который содержит кратчайший путь из заданной вершины в вершину с номером i* used[start] = 1; *// побывали в начальном узле* **for** (**int** i = 0; i < **cityGraph**.getNode() - 2; i++) {  
 **int** min\_v = 1000000;  
 **for** (**int** j = 0; j < **cityGraph**.getNode(); j++)  
 **if** (used[j] == 0 && D[j] < min\_v) { *// если еще не побывали в вершине j, и значение в вершине с номером j меньше, чем предыдущее значение,* min\_v = D[j]; *// min\_v присваивам минимальное значение.* W = j; *// W - номер узла с наименьшим значением пути* }  
 used[W] = 1;  
  
 **for** (**int** j = 0; j < **cityGraph**.getNode(); j++)  
 **if** (used[j] == 0)  
 D[j] = Math.*min*(D[j], D[W] + **mass**[W][j]); *// выбираем минимальный путь* }  
  
 **return** D[end];  
 }  
}

**package** sample;  
  
**import** java.util.Objects;  
  
**public class** CityGraph {  
 **private int node**;  
 **private int rib**;  
  
 **public** CityGraph() {  
 }  
  
 **public** CityGraph(**int** node, **int** rib) {  
 **this**.**node** = node;  
 **this**.**rib** = rib;  
 }  
  
 **public int** getNode() {  
 **return node**;  
 }  
  
 **public void** setNode(**int** node) {  
 **this**.**node** = node;  
 }  
  
 **public int** getRib() {  
 **return rib**;  
 }  
  
 **public void** setRib(**int** rib) {  
 **this**.**rib** = rib;  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
 CityGraph cityGraph = (CityGraph) o;  
 **return node** == cityGraph.**node** &&  
 **rib** == cityGraph.**rib**;  
 }  
  
 @Override  
 **public int** hashCode() {  
 **return** Objects.*hash*(**node**, **rib**);  
 }  
}

Лабораторная работа №1

ИЗУЧЕНИЕ МЕТОДОВ ОЦЕНКИ АЛГОРИТМОВ

Цель работы: Изучение методов оценки алгоритмов и программ и определение временной и емкостной сложности типовых алгоритмов и программ

Вариант 1

Составить программу, которая формирует одномерный массив из *n* случайных чисел. Определить среднее арифметическое этих чисел. Значение *n* меняется в пределах от 10 до 50 миллионов.

/Верхняя оценка - О(n), так как выполнение алгоритма зависит от количества n

Асимптотическая оценка - количественно-зависимый (выполняется одинаковое кол-во раз для фиксированного значения n)

О(n) =1+3\**n*+*n*\**f*цикла = 1+3n + (1+1+3n) = 3+6n

|  |  |  |
| --- | --- | --- |
| попытка | объем исходных данных | время работы программы |
| 1 | 10000000 | 6 мс |
| 2 | 20000000 | 9 мс |
| 3 | 30000000 | 10 мс |
| 4 | 40000000 | 14 мс |
| 5 | 50000000 | 15 мс |

**package** by.nik.lab1;  
  
**import** java.util.Random;  
  
**public class** Main {  
 **public static void** main(String[] args) {  
 Random random = **new** Random();  
  
 **for** (**int** n = 10000000; n <= 50000000; n += 10000000) {  
 **int**[] arr = **new int**[n];  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 arr[i] = random.nextInt();  
 }  
  
 **long** start = System.*currentTimeMillis*();  
  
 *getArray*(arr);  
  
 **long** finish = System.*currentTimeMillis*();  
 **long** timeConsumedMillis = finish - start;  
 System.***out***.printf(**"При n = %d алгоритм выполняется %d мс \n"**, n, timeConsumedMillis);  
 }  
 }  
  
 **public static void** getArray(**int**[] arr) {  
 **int** sumArray = 0;  
  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 sumArray+=arr[i];  
 }  
 **int** average = sumArray/arr.**length**;  
 System.***out***.println(**"Среднее = "** + average);  
 }  
}

Лабораторная работа №2

ИССЛЕДОВАНИЕ И ОЦЕНКА АЛГОРИТМОВ ПОИСКА

Цель работы. Разработка программ, реализующих различные алгоритмы поиска, и оценка их временной и пространственной сложности.

Вариант 4

Разработать алгоритм и программу дихотомического поиска. В качестве исходных данных использовать массив целых чисел, который формируется с помощью датчика случайных чисел с диапазоном от 0 до 100. Аргумент поиска – число.

Возьмем средний элемент отсортированного массива и сравним его c искомым. Если элемент меньше – продолжим поиск в левой части массива, если больше в правой, пока не останется нужный элемент. Таким образом нам понадобится число операций равное тому, сколько раз нам нужно поделить массив размером n пополам. Например, для массива в 16 элементов мы сначала поделим его на два по 8, потом 8 на два по 4, потом 4 на два по 2 и на конец 2 пополам, те всего 4 операции в худшем случае. Такое число равно двоичному логарифму: число можно столько раз разделить на 2, сколько нужно умножать 2 на 2(возводить в степень) чтобы получить это число.

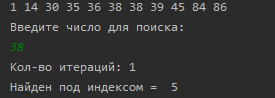
16 = 24.

А логарифм - это функция обратная возведению в степень, показывающая сколько же раз нужно умножать 2(возводить в степень), чтобы получить 16.

log2 16 = 4.

|  |  |  |
| --- | --- | --- |
| попытка | объем исходных данных | время выполнения |
| 1 | 10 | 0 мс |
| 2 | 100 | 0 мс |
| 3 | 1000 | 1 мс |
| 4 | 10000 | 2 мс |
| 5 | 100000 | 3 мс |
| 6 | 1000000 | 4 мс |
| 7 | 10000000 | 5,6 мс |

**Поэтому сложность О(log(n))**



**package** by.nik.lab2;  
  
**import** java.io.BufferedReader;  
**import** java.io.IOException;  
**import** java.io.InputStreamReader;  
**import** java.util.Arrays;  
**import** java.util.Random;  
  
**public class** Main {  
 **public static void** main(String[] args) {  
 Random random = **new** Random();  
 **int**[] arr = **new int**[10];  
  
 *// Заполнение массива* **for** (**int** i = 0; i < arr.**length**; i++) {  
 arr[i] = random.nextInt(100) + 1;  
 }  
  
 *// Сортировка* Arrays.*sort*(arr);  
 **for**(**int** value : arr) {  
 System.***out***.print(value + **" "**);  
 }  
  
 *// Ввод числа для поиска* System.***out***.println(**"\nВведите число для поиска: "**);  
 BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(System.***in***));  
 **int** number = 0;  
 **try** {  
 number = Integer.*parseInt*(bufferedReader.readLine());  
 } **catch** (IOException e) {  
 System.***out***.println(**"Вы ввели не число"**);  
 }  
  
 **long** start = System.*nanoTime*();  
  
 Binary binary = *binarySearch*(arr, number);  
  
 **long** finish = System.*nanoTime*();  
  
 System.***out***.println(**"Кол-во итераций: "** + binary.getCount());  
  
 **if** (binary.getResult() != -1)  
 System.***out***.println(**"Найден под индексом = "** + binary.getResult());  
 **else** System.***out***.println(**"Такое число не найдено"**);  
  
 **long** timeConsumedMillis = finish - start;  
 System.***out***.printf(**"алгоритм выполняется %d нс \n"**, timeConsumedMillis);  
 }  
  
 *// Метод дихотомии* **public static** Binary binarySearch(**int**[] sortedArray, **int** key) {  
 **int** index = -1; *//если не найдёт элемент, то вернёт его* **int** count = 0; *// кол-во итераций поиска* **int** firstIndex = 0; *//первый элемент массива* **int** lastIndex = sortedArray.**length** - 1; *//последний элемент массива* **while** (firstIndex <= lastIndex) {  
 count++;  
 **int** mid = (firstIndex + lastIndex) / 2;  
 **if** (sortedArray[mid] < key) {  
 firstIndex = mid + 1;  
 } **else if** (sortedArray[mid] > key) {  
 lastIndex = mid - 1;  
 } **else if** (sortedArray[mid] == key) {  
 index = mid;  
 **break**;  
 }  
 }  
 **return new** Binary(count, index);  
 }  
}

**package** by.nik.lab2;  
  
**import** java.util.Objects;  
  
**public class** Binary {  
 **private int count**;  
 **private int index**;  
  
 **public** Binary(**int** count, **int** index) {  
 **this**.**count** = count;  
 **this**.**index** = index;  
 }  
  
 **public int** getCount() {  
 **return count**;  
 }  
  
 **public void** setCount(**int** count) {  
 **this**.**count** = count;  
 }  
  
 **public int** getResult() {  
 **return index**;  
 }  
  
 **public void** setResult(**int** index) {  
 **this**.**index** = index;  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
 Binary binary = (Binary) o;  
 **return count** == binary.**count** &&  
 **index** == binary.**index**;  
 }  
  
 @Override  
 **public int** hashCode() {  
 **return** Objects.*hash*(**count**, **index**);  
 }  
}

Лабораторная работа № 3

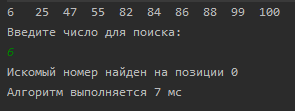
РАЗРАБОТКА РЕКУРСИВНЫХ АЛГОРИТМОВ

Цель работы. Разработка программ, реализующих различные рекурсивные алгоритмы, и оценка их временной и пространственной сложности.

Вариант 12

Ввода одномерного массива и интерполирующего поиска целочисленного значения ключа в нем.

Верхняя оценка - от log(log(n)) до О(n), в зависимости от нахождения элементов



**package** by.nik.lab3;  
  
**import** java.io.BufferedReader;  
**import** java.io.IOException;  
**import** java.io.InputStreamReader;  
**import** java.util.Arrays;  
**import** java.util.Random;  
  
**public class** Main {  
 **public static void** main(String[] args) {  
 Random random = **new** Random();  
 **int**[] arr = **new int**[100000];  
 *// Заполнение массива* **for** (**int** i = 0; i < arr.**length**; i++) {  
 arr[i] = random.nextInt(1000000)+1;  
  
 }  
  
 Arrays.*sort*(arr);  
  
 **for** (**int** value : arr) {  
 System.***out***.print(value + **"\t"**);  
 }  
  
 *// Ввод числа для поиска* System.***out***.println(**"\nВведите число для поиска: "**);  
 BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(System.***in***));  
 **int** number = 0;  
 **try** {  
 number = Integer.*parseInt*(bufferedReader.readLine());  
 } **catch** (IOException e) {  
 System.***out***.println(**"Вы ввели не число"**);  
 }  
  
 **long** start = System.*currentTimeMillis*();  
  
 *interSearch*(arr, 0, arr.**length** - 1, number);  
  
 **long** finish = System.*currentTimeMillis*();  
 **long** timeConsumedMillis = finish - start;  
 System.***out***.printf(**"Алгоритм выполняется %d мс \n"**, timeConsumedMillis);  
 }  
  
 **public static void** interSearch(**int**[] arr, **int** low, **int** high, **int** k) {  
 **if** (k >= 0) {  
 **while** (arr[low] < k && arr[high] >= k) {  
 **int** mid = low + (k - arr[low]) \* (high - low) / (arr[high] - arr[low]);  
 **if** (arr[mid] < k) {  
 low = mid + 1;  
 *interSearch*(arr, low, high, k);  
 } **else if** (arr[mid] > k) {  
 high = mid - 1;  
 *interSearch*(arr, low, high, k);  
 } **else** {  
 System.***out***.printf(**"Искомый номер найден на позиции %d\n"**, mid);  
 **return**;  
 }  
 }  
 **if** (arr[low] == k) {  
 System.***out***.printf(**"Искомый номер найден на позиции %d\n"**, low);  
 } **else if** (arr[high] == k) {  
 System.***out***.printf(**"Искомый номер найден на позиции %d\n"**, high);  
 } **else** {  
 System.***out***.println(**"Искомый номер не найден\n"**);  
 }  
 }  
 }  
  
}

Лабораторная работа №4

ИССЛЕДОВАНИЕ И ОЦЕНКА АЛГОРИТМОВ СОРТИРОВКИ

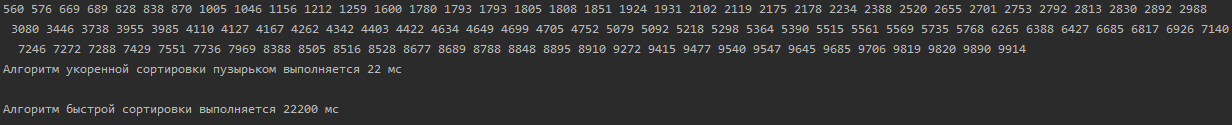
Цель работы. Разработка программ, реализующих различные алгоритмы сортировки, и оценка их временной и пространственной сложности.

Вариант 12

Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и быстрой сортировки. Исходные данные задавать с помощью датчика случайных чисел.

Верхняя оценка - O(n^2) для пузырьковой

Верхняя оценка - O(n\*log2\*n) для быстрой



**package** by.nik.lab4;  
  
**import** java.util.\*;  
  
**public class** Main {  
 **public static void** main(String[] args) {  
 Random random = **new** Random();  
 **int**[] arrBubble = **new int**[100000];  
 **for** (**int** i = 0; i < arrBubble.**length**; i++) {  
 arrBubble[i] = random.nextInt(1000000) + 1;  
 }  
  
 **int**[] arrQuick = arrBubble.clone();  
 **for** (**int** i = 0; i < arrQuick.**length**; i++) {  
 arrQuick[i] = random.nextInt(10000) + 1;  
 }  
  
 **long** startBubble = System.*currentTimeMillis*();  
 *fastBubbleSort*(arrBubble);  
 **long** finishBubble = System.*currentTimeMillis*();  
 **long** timeConsumedMillisBubble = finishBubble - startBubble;  
  
 **long** start = System.*nanoTime*();  
 *quickSort*(arrQuick, 0, arrQuick.**length**-1);  
 **long** finish = System.*nanoTime*();  
 **long** timeConsumedMillisQuickSort = finish - start;  
  
 System.***out***.printf(**"\nАлгоритм укоренной сортировки пузырьком выполняется %d мс \n"**, timeConsumedMillisBubble);  
 System.***out***.printf(**"\nАлгоритм быстрой сортировки выполняется %d нс \n"**, timeConsumedMillisQuickSort);  
 }  
  
 **public static void** fastBubbleSort(**int**[] arr) {  
 **int** k = 1;  
 **int** temp;  
 **boolean** trp;  
  
 **do** {  
 trp = **false**;  
 **for** (**int** i = 0; i < arr.**length** - k; i++) {  
 **if** (arr[i] > arr[i + 1]) {  
 temp = arr[i];  
 arr[i] = arr[i + 1];  
 arr[i + 1] = temp;  
 trp = **true**;  
 }  
 }  
 k++;  
 } **while** (trp);  
  
 **for** (**int** i : arr) {  
 System.***out***.printf(**"%d "**, i);  
 }  
 }  
  
 **public static void** quickSort(**int**[] arr, **int** low, **int** high) {  
 **if** (arr.**length** == 0)  
 **return**;*//завершить выполнение если длина массива равна 0* **if** (low >= high)  
 **return**;*//завершить выполнение если уже нечего делить  
  
 // выбрать опорный элемент* **int** middle = low + (high - low) / 2;  
 **int** opora = arr[middle];  
 *// разделить на подмассивы, который больше и меньше опорного элемента* **int** i = low, j = high;  
 **while** (i <= j) {  
 **while** (arr[i] < opora) {  
 i++;  
 }  
 **while** (arr[j] > opora) {  
 j--;  
 }  
  
 **if** (i <= j) {*//меняем местами* **int** temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 i++;  
 j--;  
 }  
 }  
 *// вызов рекурсии для сортировки левой и правой части* **if** (low < j)  
 *quickSort*(arr, low, j);  
  
 **if** (high > i)  
 *quickSort*(arr, i, high);  
 }  
  
}

Лабораторная работа №5

ИССЛЕДОВАНИЕ И ОЦЕНКА АЛГОРИТМОВ ПОИСКА НА ДЕРЕВЬЯХ

Цель работы. Разработка программ, реализующих алгоритмы формирования и обхода двоичных и В+ деревьев, а также поиска элементов в них, и оценка их временной и пространственной сложности.

Вариант 12

1. Построить двоичное дерево, содержащее *n* = 16 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном *D* от 0 до 120.

2. Построить В+ дерево, содержащее *n* = 16 узлов и имеющее степень *m* = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном *D* от 0 до 120.

3. Обеспечить симметричный обход деревьев.

4. Выполнить поиск значения ключа по близости сверху.

*Построение бинарного дерева*

Алгоритм построения упорядоченного бинарного дерева может быть сформулирован так: 1.Первый элемент с ключом р[1] становится корнем дерева.

2.Значение ключа второго узла р[2] сравнивается с р[1] (корня дерева). Если р[2] < p[1], то второй элемент помещается на левой от корня ветви, в противном случае – на правой. В результате будет получено упорядоченное дерево из первых двух узлов.

3. Далее на каждом шаге создается упорядоченное дерево из первых i элементов. Выбор i-го узла производится следующим образом. Ключ р[i] сравнивается с корневым значением и выполняется переход по левому адресу (если р[1]>р[i]), в противном случае (при р[1] <= р [i]) – по правому адресу. Далее ключ достигнутого узла также сравнивается с р[i], и снова организуется 57 переход по левому и правому адресу и т.д. При нахождении незаполненного адреса связи ему присваивается ключ р[i].

Пункт 3 повторяется до тех пор, пока не будут включены в дерево все элементы исходного массива.

Поиск узла в бинарном дереве

Искомое значение ключа q сравнивается со значением корня р[1]. Если р[1]>q, просмотр дерева продолжается по левой от корня ветви, иначе (если р[1] <= q) – по правой. Для произвольного узла с ключом р[i] могут быть получены следующие результаты сравнения:

a) р[i] = q – элемент, удовлетворяющий условию поиска найден, и поиск заканчивается по правой ветви.

b) р[i] >q – производится переход к элементу, расположенному на левой ветви р(i).

c) р[i] < q – производится переход к элементу, расположенному на правой ветви р[i].

Поиск заканчивается, когда у какого-либо узла отсутствует адрес связи, необходимый для дальнейшего продолжения поиска.

Алгоритм включения узла в дерево

Если дерево пусто, заменить его на дерево с одним корневым узлом и остановиться. Иначе сравнить ключ К с ключом корневого узла X: если K>X, рекурсивно добавить К в правое поддерево, иначе рекурсивно добавить К в левое поддерево.

Алгоритм удаления узла из дерева

1. Если дерево T пусто, остановиться Иначе сравнить K с ключом X корневого узла n.

2.1. Если K>X, рекурсивно удалить K из правого поддерева Т

2.2. Иначе если K<X, рекурсивно удалить K из левого поддерева Т.

2.3. Если K=X, то необходимо рассмотреть два случая.

2.3.1.Если одного из детей нет, то значения второго ребѐнка m ставим вместо соответствующих значений корневого узла, затирая его старые значения, и освобождаем память, занимаемую узлом m.

2.3.2. Если оба потомка присутствуют, то

2.3.2.1. найдѐм узел m, являющийся самым левым узлом правого поддерева;

2.3.2.2. скопируем значения полей (ключ, значение) узла m в соответствующие поля узла n.

2.3.2.3. у предка узла m заменим ссылку на узел m ссылкой на правого потомка узла m (который может быть равен нулевому указателю).

2.3.2.4. освободим память, занимаемую узлом m (на него теперь никто не указывает, а его данные были перенесены в узел n).

|  |  |  |
| --- | --- | --- |
| **Сложность** | **В среднем** | **В худшем случае** |
| **Расход памяти** | O(n) | O(n) |
| **Поиск** | O(log n) | O(n) |
| **Вставка** | O(log n) | O(n) |
| **Удаление** | O(log n) | O(n) |

**package by.nik.lab5**;  
  
**import java.io.BufferedReader**;  
**import java.io.IOException**;  
**import java.io.InputStreamReader**;  
**import java.util.Random**;  
  
**public class** Main {  
 **private static final int** MAX\_VALUE **= *120***;  
  
 **public static void** main(**String**[] args) {  
 Random rnd **= new** Random();  
 **int** n **= *16***;  
 **int**[] array **= new int**[n];  
 **for** (**int** i **= *0***; i **<** n; i**++**) {  
 array[i] **=** rnd.nextInt(MAX\_VALUE) **+ *1***;  
 }  
 BinaryTree tree **= new** BinaryTree(array[***0***], ***null***);  
 System.out.println(**"Рандомный массив: "**);  
 **for** (**int** i **= *1***; i **<** n; i**++**) {  
 tree.add(array[i]);  
 System.out.printf(**"%d "**, array[i]);  
 }  
  
 System.out.println(**"*\n*Отсортированный массив:"**);  
 tree.printInorder(tree);  
  
 System.out.println(**"*\n*Введите число для поиска: "**);  
 BufferedReader bufferedReader **= new** BufferedReader(**new** InputStreamReader(System.in));  
 **int** number **= *0***;  
 **try** {  
 number **=** Integer.parseInt(bufferedReader.readLine());  
 } **catch** (**IOException** e) {  
 System.out.println(**"Вы ввели не число"**);  
 }  
 BinaryTree binaryTree **=** tree.search(number);  
 **if** (binaryTree **== *null***) {  
 System.out.println(**"Не найдено"**);  
 } **else** System.out.println(binaryTree.getValue());  
  
 }  
}

**package by.nik.lab5**;  
  
**import java.util.Objects**;  
  
**public class** BinaryTree {  
 **private** BinaryTree parent;  
 **private** BinaryTree left;  
 **private** BinaryTree right;  
 **private int** value;  
  
 **public** BinaryTree(**int** val, BinaryTree Parent) {  
 this.value **=** val;  
 this.parent **=** Parent;  
 }  
  
 **public void** add(**int** val) {  
 **if** (val **<** this.value) {  
 **if** (this.left **== *null***) {  
 this.left **= new** BinaryTree(val, this);  
 } **else** this.left.add(val);  
 } **else if** (val **>** this.value) {  
 **if** (this.right **== *null***) {  
 this.right **= new** BinaryTree(val, this);  
 } **else** this.right.add(val);  
 }  
 }  
  
 **private** BinaryTree search(BinaryTree tree, **int** val) {  
 **if** (tree **== *null***) **return *null***;  
 **switch** (Integer.compare(val, tree.value)) {  
 **case *1*:  
 return** search(tree.right, val);  
 **case -*1*:  
 return** search(tree.left, val);  
 **case *0*:  
 return** tree;  
 **default:  
 return *null***;  
 }  
 }  
  
 **public** BinaryTree search(**int** val) {  
 **return** search(this, val);  
 }  
  
 **public void** printInorder(BinaryTree root) {  
 **if** (root **== *null***) {  
 **return**;  
 }  
 printInorder(root.left); *//рекурсивный вызов левого поддерева* System.out.println(root.value);  
 printInorder(root.right); *//рекурсивный вызов правого поддерева* }  
  
 @**Override  
 public** String toString() {  
 **return "BinaryTree{" +  
 "Parent=" +** parent **+  
 ", Left=" +** left **+  
 ", Right=" +** right **+  
 ", Value=" +** value **+  
 '}'**;  
 }  
  
 @**Override  
 public boolean** equals(Object o) {  
 **if** (this **==** o) **return *true***;  
 **if** (o **== *null* ||** getClass() **!=** o.getClass()) **return *false***;  
 BinaryTree that **=** (BinaryTree) o;  
 **return** value **==** that.value **&&** Objects.equals(parent, that.parent) **&&** Objects.equals(left, that.left) **&&** Objects.equals(right, that.right);  
 }  
  
 @**Override  
 public int** hashCode() {  
 **return** Objects.hash(parent, left, right, value);  
 }  
  
 **public** BinaryTree getParent() {  
 **return** parent;  
 }  
  
 **public void** setParent(BinaryTree parent) {  
 this.parent **=** parent;  
 }  
  
 **public** BinaryTree getLeft() {  
 **return** left;  
 }  
  
 **public void** setLeft(BinaryTree left) {  
 this.left **=** left;  
 }  
  
 **public** BinaryTree getRight() {  
 **return** right;  
 }  
  
 **public void** setRight(BinaryTree right) {  
 this.right **=** right;  
 }  
  
 **public int** getValue() {  
 **return** value;  
 }  
  
 **public void** setValue(**int** value) {  
 this.value **=** value;  
 }  
  
 **public boolean** remove(**int** val) {  
 *//Проверяем, существует ли данный узел* BinaryTree tree **=** search(val);  
 **if** (tree **== *null***) {  
 *//Если узла не существует, вернем false* **return *false***;  
 }  
 BinaryTree curTree;  
 *//Если удаляем корень* **if** (tree **==** this) {  
 **if** (tree.right **!= *null***) {  
 curTree **=** tree.right;  
 } **else** curTree **=** tree.left;  
  
 **while** (curTree.left **!= *null***) {  
 curTree **=** curTree.left;  
 }  
 **int** temp **=** curTree.value;  
 this.remove(temp);  
 tree.value **=** temp;  
  
 **return *true***;  
 }  
  
 *//Удаление листьев* **if** (tree.left **== *null* &&** tree.right **== *null* &&** tree.parent **!= *null***) {  
 **if** (tree **==** tree.parent.left)  
 tree.parent.left **= *null***;  
 **else** {  
 tree.parent.right **= *null***;  
 }  
 **return *true***;  
 }  
 *//Удаление узла, имеющего левое поддерево, но не имеющее правого поддерева* **if** (tree.left **!= *null* &&** tree.right **== *null***) {  
 *//Меняем родителя* tree.left.parent **=** tree.parent;  
 **if** (tree **==** tree.parent.left) {  
 tree.parent.left **=** tree.left;  
 } **else if** (tree **==** tree.parent.right) {  
 tree.parent.right **=** tree.left;  
 }  
 **return *true***;  
 }  
 *//Удаление узла, имеющего правое поддерево, но не имеющее левого поддерева* **if** (tree.left **== *null* &&** tree.right **!= *null***) {  
 *//Меняем родителя* tree.right.parent **=** tree.parent;  
 **if** (tree **==** tree.parent.left) {  
 tree.parent.left **=** tree.right;  
 } **else if** (tree **==** tree.parent.right) {  
 tree.parent.right **=** tree.right;  
 }  
 **return *true***;  
 }  
 *//Удаляем узел, имеющий поддеревья с обеих сторон* **if** (tree.right **!= *null* &&** tree.left **!= *null***) {  
 curTree **=** tree.right;  
 **while** (curTree.left **!= *null***) {  
 curTree **=** curTree.left;  
 }  
 *//Если самый левый элемент является первым потомком* **if** (curTree.parent **==** tree) {  
 curTree.left **=** tree.left;  
 tree.left.parent **=** curTree;  
 curTree.parent **=** tree.parent;  
 **if** (tree **==** tree.parent.left) {  
 tree.parent.left **=** curTree;  
 } **else if** (tree **==** tree.parent.right) {  
 tree.parent.right **=** curTree;  
 }  
 **return *true***;  
 }  
 *//Если самый левый элемент НЕ является первым потомком* **else** {  
 **if** (curTree.right **!= *null***) {  
 curTree.right.parent **=** curTree.parent;  
 }  
 curTree.parent.left **=** curTree.right;  
 curTree.right **=** tree.right;  
 curTree.left **=** tree.left;  
 tree.left.parent **=** curTree;  
 tree.right.parent **=** curTree;  
 curTree.parent **=** tree.parent;  
 **if** (tree **==** tree.parent.left) {  
 tree.parent.left **=** curTree;  
 } **else if** (tree **==** tree.parent.right) {  
 tree.parent.right **=** curTree;  
 }  
 **return *true***;  
 }  
 }  
 **return *false***;  
 }  
}

Лабораторная работа №6

**Представление графов в памяти ЭВМ.**

Разработать класс «Граф», для работы с графом, представленным

* матрицей смежности,
* матрицей инцидентности,
* списками смежности,
* списками дуг.

Класс «Граф» должен содержать следующие методы:

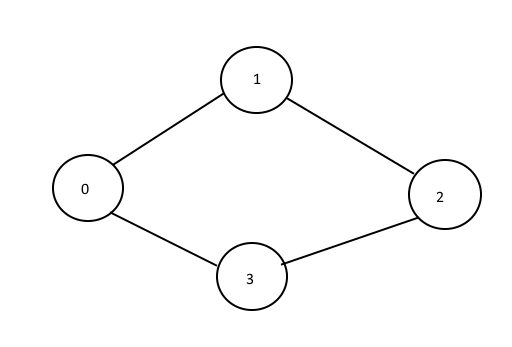
- конструктор/ деструктор;

- конструктор копирования;

- добавление/удаление вершины;

- добавление/удаление дуги;

- печать;



Матрица смежности – из какой вершины в какую

{{0, 1, 0, 1},  
 {1, 0, 1, 0},  
 {0, 1, 0, 1},  
 {1, 0, 1, 0}}

Матрица инцидентности

строки соответствуют вершинам, а столбцы рёбрам

{{1, 0, 0, 1},  
 {1, 1, 0, 0},  
 {0, 1, 1, 0},  
 {0, 0, 1, 1}}

Списки смежности – из какой вершины в какую

{{0, 1}, {1, 2}, {2, 3}, {3, 0}}

Список дуг – из одной вершины в другую + вес ребра

{{0, 1, 1},  
 {0, 3, 1},  
 {1, 2, 1},  
 {1, 0, 1},  
 {2, 1, 1},  
 {2, 3, 1},  
 {3, 2, 1},  
 {3, 0, 1}}

Лабораторная работа №7

**Обходы графов в глубину и в ширину**

Расширить класс «Граф» методом, реализующим один из алгоритмов обхода графа:

4. Алгоритм обхода графа в ширину с циклическими списками дуг.

Лабораторная работа №8

**Основные задачи теории графов, алгоритмы их решения и области их приложения**

Расширить класс «Граф» методом, решающим одну из задач теории графов:

4. Размещение центров.

**package by.nik.lab6**;  
  
**public class** Main {  
  
 **public static void** main(**String**[] args) {  
  
 **int**[][] adjacentMatrix **=** {{***0***, ***1***, ***0***, ***1***}, *//матрица смежности* {***1***, ***0***, ***1***, ***0***},  
 {***0***, ***1***, ***0***, ***1***},  
 {***1***, ***0***, ***1***, ***0***}};  
 **int**[][] incidencesMatrix **=** {{***1***, ***0***, ***0***, ***1***}, *// матрица инцидентности* {***1***, ***1***, ***0***, ***0***},  
 {***0***, ***1***, ***1***, ***0***},  
 {***0***, ***0***, ***1***, ***1***}};  
 **int**[][] adjacentList **=** {{***0***, ***1***}, {***1***, ***2***}, {***2***, ***3***}, {***3***, ***0***}}; *// списки смежности* **int**[][] ribList **=** {{***0***, ***1***, ***1***},  
 {***0***, ***3***, ***1***},  
 {***1***, ***2***, ***1***},  
 {***1***, ***0***, ***1***},  
 {***2***, ***1***, ***1***},  
 {***2***, ***3***, ***1***},  
 {***3***, ***2***, ***1***},  
 {***3***, ***0***, ***1***}}; *// списки дуг* Graph graph **= new** Graph(adjacentMatrix, incidencesMatrix, adjacentList, ribList);  
 Graph graph2Copy **= new** Graph(graph);  
  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.printAdjacentMatrix();  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.printIncidencesMatrix();  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.printAdjacentList();  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.printRibList();  
 System.out.println(**"\*\*\*\*\*\*"**);  
 System.out.println(**"После добавления"**);  
 graph2Copy.addNode();  
 graph2Copy.addRib(***0***, ***4***);  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.printAdjacentMatrix();  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.printIncidencesMatrix();  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.printAdjacentList();  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.printRibList();  
  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.search();  
 System.out.println(**"\*\*\*\*\*\*"**);  
 graph2Copy.center();  
 }  
}

**package by.nik.lab6**;  
  
**import java.util.\***;  
  
**public class** Graph {  
 **private int**[][] adjacentMatrix; *//матрица смежности* **private int**[][] incidencesMatrix; *// матрица инцидентности* **private int**[][] adjacentList; *// списки смежности* **private int**[][] ribList; *// списки дуг* **public** Graph() {  
 }  
  
 **public** Graph(**int**[][] adjacentMatrix, **int**[][] incidencesMatrix, **int**[][] adjacentList, **int**[][] ribList) {  
 this.adjacentMatrix **=** adjacentMatrix;  
 this.incidencesMatrix **=** incidencesMatrix;  
 this.adjacentList **=** adjacentList;  
 this.ribList **=** ribList;  
 }  
  
 **public** Graph(Graph graph) {  
 this(graph.getAdjacentMatrix(), graph.getIncidencesMatrix(), graph.getAdjacentList(), graph.getRibList());  
 }  
  
 **public void** addNode() {  
 *// смежная* **int**[][] a **= new int**[adjacentMatrix.length **+ *1***][adjacentMatrix.length **+ *1***];  
 **for** (**int** i **= *0***; i **<** adjacentMatrix.length; i**++**)  
 System.arraycopy(adjacentMatrix[i], ***0***, a[i], ***0***, adjacentMatrix.length);  
 this.adjacentMatrix **=** a;  
 *// инцидентности* a **= new int**[incidencesMatrix.length **+ *1***][incidencesMatrix.length];  
 **for** (**int** i **= *0***; i **<** incidencesMatrix.length; i**++**)  
 System.arraycopy(incidencesMatrix[i], ***0***, a[i], ***0***, incidencesMatrix.length);  
 this.incidencesMatrix **=** a;  
 }  
  
 **public void** addRib(**int** firstNode, **int** secondNode) {  
 **if** (firstNode **>=** adjacentMatrix.length **||** secondNode **>=** adjacentMatrix.length) {  
 System.out.println(**"Такого узла не существует:" +** firstNode **+ " " +** secondNode);  
 }  
 *// смежная* adjacentMatrix[firstNode][secondNode] **= *1***;  
 adjacentMatrix[secondNode][firstNode] **= *1***;  
 *// инцидентности* **int**[][] a **= new int**[incidencesMatrix.length][incidencesMatrix.length];  
 **for** (**int** i **= *0***; i **<** incidencesMatrix.length; i**++**)  
 **for** (**int** j **= *0***; j **<** incidencesMatrix[i].length; j**++**)  
 a[i][j] **=** incidencesMatrix[i][j];  
 a[firstNode][incidencesMatrix.length **- *1***] **= *1***;  
 a[secondNode][incidencesMatrix.length **- *1***] **= *1***;  
 this.incidencesMatrix **=** a;  
 *// списки смежности* a **= new int**[adjacentList.length **+ *1***][***2***];  
 **for** (**int** i **= *0***; i **<** adjacentList.length; i**++**)  
 **for** (**int** j **= *0***; j **<** adjacentList[i].length; j**++**)  
 a[i][j] **=** adjacentList[i][j];  
 a[adjacentList.length][***0***] **=** firstNode;  
 a[adjacentList.length][***1***] **=** secondNode;  
 this.adjacentList **=** a;  
 *// списки дуг* a **= new int**[ribList.length **+ *2***][***3***];  
 **for** (**int** i **= *0***; i **<** ribList.length; i**++**)  
 **for** (**int** j **= *0***; j **<** ribList[i].length; j**++**)  
 a[i][j] **=** ribList[i][j];  
 a[ribList.length][***0***] **=** firstNode;  
 a[ribList.length][***1***] **=** secondNode;  
 a[ribList.length][***2***] **= *1***;  
 a[ribList.length **+ *1***][***0***] **=** secondNode;  
 a[ribList.length **+ *1***][***1***] **=** firstNode;  
 a[ribList.length **+ *1***][***2***] **= *1***;  
 this.ribList **=** a;  
 }  
  
 **public void** search() {  
 Queue<**Integer**> queue **= new** ArrayDeque<**Integer**>();  
 **int**[] nodes **= new int**[adjacentMatrix.length]; *// вершины графа* **for** (**int** i **= *0***; i **<** adjacentMatrix.length; i**++**)  
 nodes[i] **= *0***; *// исходно все вершины равны 0* queue.add(***0***); *// помещаем в очередь первую вершину* **while** (queue.size() **!= *0***) { *// пока очередь не пуста* **int** node **=** queue.poll(); *// извлекаем вершину* nodes[node] **= *2***; *// отмечаем ее как посещенную* **for** (**int** j **= *0***; j **<** adjacentMatrix.length; j**++**) { *// проверяем для нее все смежные вершины* **if** (adjacentMatrix[node][j] **== *1* &&** nodes[j] **== *0***) { *// если вершина смежная и не обнаружена* queue.add(j); *// добавляем ее в очередь* nodes[j] **= *1***; *// отмечаем вершину как обнаруженную* }  
 }  
 System.out.println(node);  
 ; *// выводим номер вершины* }  
 }  
  
 **public void** center() {  
 **int** N **=** adjacentMatrix.length; *// Количество вершин в графе* **int** INF **=** Integer.MAX\_VALUE; *// мксимальное* **int**[][] d **= new int**[N][N]; *// Дистанции в графе* **int**[] e **= new int**[N]; *// Эксцентриситет вершин* Set<**Integer**> c **= new** HashSet<>(); *// Центр графа* **int** rad **=** INF; *// Радиус графа  
  
 // Алгоритм Флойда-Уоршелла* **for** (**int** k **= *0***; k **<** N; k**++**) {  
 **for** (**int** j **= *0***; j **<** N; j**++**) {  
 **for** (**int** i **= *0***; i **<** N; i**++**) {  
 d[i][j] **=** Math.min(d[i][j], d[i][k] **+** d[k][j]);  
 }  
 }  
 }  
  
 *// Нахождение эксцентриситета* **for** (**int** i **= *0***; i **<** N; i**++**) {  
 **for** (**int** j **= *0***; j **<** N; j**++**) {  
 e[i] **=** Math.max(e[i], d[i][j]);  
 }  
 }  
  
 *// Нахождение диаметра и радиуса* **for** (**int** i **= *0***; i **<** N; i**++**) {  
 rad **=** Math.min(rad, e[i]);  
 }  
  
 **for** (**int** i **= *0***; i **<** N; i**++**) {  
 **if** (e[i] **==** rad) {  
 c.add(i);  
 System.out.println(i);  
 }  
 }  
 }  
  
 **public int**[][] getAdjacentMatrix() {  
 **return** adjacentMatrix;  
 }  
  
 **public void** printAdjacentMatrix() {  
 **for** (**int** i **= *0***; i **<** adjacentMatrix.length; i**++**) {  
 **for** (**int** j **= *0***; j **<** adjacentMatrix[i].length; j**++**) {  
 System.out.print(adjacentMatrix[i][j] **+ "*\t*"**);  
 }  
 System.out.println();  
 }  
 }  
  
 **public void** setAdjacentMatrix(**int**[][] adjacentMatrix) {  
 this.adjacentMatrix **=** adjacentMatrix;  
 }  
  
 **public int**[][] getIncidencesMatrix() {  
 **return** incidencesMatrix;  
 }  
  
 **public void** printIncidencesMatrix() {  
 **for** (**int** i **= *0***; i **<** incidencesMatrix.length; i**++**) {  
 **for** (**int** j **= *0***; j **<** incidencesMatrix[i].length; j**++**) {  
 System.out.print(incidencesMatrix[i][j] **+ "*\t*"**);  
 }  
 System.out.println();  
 }  
 }  
  
 **public void** setIncidencesMatrix(**int**[][] incidencesMatrix) {  
 this.incidencesMatrix **=** incidencesMatrix;  
 }  
  
 **public int**[][] getAdjacentList() {  
 **return** adjacentList;  
 }  
  
 **public void** printAdjacentList() {  
 **for** (**int** i **= *0***; i **<** adjacentList.length; i**++**) {  
 **for** (**int** j **= *0***; j **<** adjacentList[i].length; j**++**) {  
 System.out.print(adjacentList[i][j] **+ "*\t*"**);  
 }  
 System.out.println();  
 }  
 }  
  
 **public void** setAdjacentList(**int**[][] adjacentList) {  
 this.adjacentList **=** adjacentList;  
 }  
  
 **public int**[][] getRibList() {  
 **return** ribList;  
 }  
  
 **public void** printRibList() {  
 **for** (**int** i **= *0***; i **<** ribList.length; i**++**) {  
 **for** (**int** j **= *0***; j **<** ribList[i].length; j**++**) {  
 System.out.print(ribList[i][j] **+ "*\t*"**);  
 }  
 System.out.println();  
 }  
 }  
  
 **public void** setRibList(**int**[][] ribList) {  
 this.ribList **=** ribList;  
 }  
  
 @**Override  
 public** String toString() {  
 **return "CityGraph{" +  
 "adjacentMatrix=" +** Arrays.toString(adjacentMatrix) **+  
 ", incidencesMatrix=" +** Arrays.toString(incidencesMatrix) **+  
 ", adjacentList=" +** Arrays.toString(adjacentList) **+  
 ", ribList=" +** Arrays.toString(ribList) **+  
 '}'**;  
 }  
  
 @**Override  
 public boolean** equals(Object o) {  
 **if** (this **==** o) **return *true***;  
 **if** (o **== *null* ||** getClass() **!=** o.getClass()) **return *false***;  
 Graph cityGraph **=** (Graph) o;  
 **return** Arrays.equals(adjacentMatrix, cityGraph.adjacentMatrix) **&&** Arrays.equals(incidencesMatrix, cityGraph.incidencesMatrix) **&&** Arrays.equals(adjacentList, cityGraph.adjacentList) **&&** Arrays.equals(ribList, cityGraph.ribList);  
 }  
  
 @**Override  
 public int** hashCode() {  
 **int** result **=** Arrays.hashCode(adjacentMatrix);  
 result **= *31* \*** result **+** Arrays.hashCode(incidencesMatrix);  
 result **= *31* \*** result **+** Arrays.hashCode(adjacentList);  
 result **= *31* \*** result **+** Arrays.hashCode(ribList);  
 **return** result;  
 }  
}