**DSA cheatsheet**

**Important Concepts:**

|  |  |
| --- | --- |
| String operations:  str1==str2 //compares address;  String newStr = str1.equals(str2); //compares the values  String newStr = str1.equalsIgnoreCase() //compares the values ignoring the case  newStr = str1.length() //calculates length  newStr = str1.charAt(i) //extract i'th character  newStr = str1.toUpperCase() //returns string in ALL CAPS  newStr = str1.toLowerCase() //returns string in ALL LOWERvCASE  newStr = str1.replace(oldVal, newVal) //search and replace  newStr = str1.trim() //trims surrounding whitespace  newStr = str1.contains("value"); //check for the values  newStr = str1.toCharArray(); // convert String to character type array  newStr = str1.IsEmpty(); //Check for empty String  newStr = str1.endsWith(); //Checks if string ends with the given suffix |  |
| Gap traversal:  public static void main(String[] args) {  int[][] arr = {  {5, 6, 7, 8},  {0, 1, 2, 9},  {5, 2, 5, 8},  {0, 3, 4, 0},  // {2, 3, 4, 5}  };  for (int g = 0; g < arr.length; g++) {  for (int i = 0, j = g; j < arr[0].length; i++, j++) {  if (g == 0) {  System.out.print(arr[i][j] + " ");  }  }  }  } | Prefix suffix:  public static void main(String[] args) {  String str="aabb";  for (int i = 0; i < str.length(); i++) {  String pre=str.substring(0,i+1);  String suff=str.substring(i+1);  System.out.println(pre+" "+suff);  }  } |
| String swap:  static String swap(String str, int i, int j) {  char ith = str.charAt(i);  char jth = str.charAt(j);  String pre = str.substring(0, i);  String mid = str.substring(i + 1, j);  String post = str.substring(j + 1);  return pre + jth + mid + ith + post;  } | Subsequence:  public static void main(String[] args) {  String s = "abc";  sol(s, "");  }  private static void sol(String q, String a) {  if (q.length() == 0) {  System.out.println("" + a);  return;  }  char ch = q.charAt(0);  String rest = q.substring(1);  sol(rest, a);  sol(rest, ch + a);  } |
| Comparable:  class Student implements Comparable<Student>{  int id;  int age;  public Student(int id, int age) {  this.id = id;  this.age = age;  }  @Override  public int compareTo(Student s) {  if(age>s.age){  return 1;  }else{  return -1;  }  }  }  public class WhyComparable {  public static void main(String args[]){  Student s1=new Student(1,14);  Student s2=new Student(1,13);  System.out.println(""+s1.compareTo(s2));  }  } | PriorityQueue:  public static void main(String[] args) {  PriorityQueue<Integer> p = new PriorityQueue<>();  p.add(2);  p.add(3);  p.add(1);  while (p.size() > 0) {  System.out.println("" + p.remove());  }  } |

**Basic Math:**

|  |  |
| --- | --- |
| Count digits:  private static int countDigits(int n) {  int cnt = 0;  while (n != 0) {  n = n / 10;  cnt++;  }  return cnt;  } | Count digits recursive:  public static int countDigitsR(int n) {  if (n == 0) {  return 0;  } else {  return 1 + countDigitsR(n / 10);  }  } |
| Is palindrome:  static boolean isPalindrome(int n) {  int temp = n;  int rev = 0;  while (temp != 0) {  int ld = temp % 10;  rev = rev \* 10 + ld;  temp = temp / 10;  }  return (rev == n);  } | Factorial recursive:  static int factR(int n) {  if (n == 0) {  return 1;  }  return n \* factR(n - 1);  } |
| Count trailing zeroes:  static int countTrailingZeroes(int n) {  int res = 0;  for (int i = 5; i <= n; i = i \* 5) {  res = res + n / i;  }  return res;  } | Check prime:  static boolean isPrime(int n) {  if (n == 1) {  return false;  }  for (int i = 2; i \* i <= n; i++) {  if (n % i == 0) {  return false;  }  }  return true;  } |
| Calculate power:  static int power(int x, int n) {  if (n == 0) {  return 1;  }  int temp = power(x, n / 2);  temp = temp \* temp;  if (n % 2 == 0) {  return temp;  } else {  return temp \* x;  }  } | Prime seive:  public static void main(String[] args) {  boolean myseive[] = seive(20);  for (int i = 0; i < myseive.length; i++) {  System.out.println(i + " " + myseive[i]);  }  }  static boolean[] seive(int n) {  boolean[] arr = new boolean[n + 1];  Arrays.fill(arr, true);  arr[0] = false;  arr[1] = false;  for (int i = 2; i < Math.sqrt(n); i++) {  for (int j = 2 \* i; j <= n; j = j + i) {  arr[j] = false;  }  }  return arr;  } |
| Trailing zeroes:  public static void main(String[] args) {  int res=1000;  int n=7;  for(int i=5;i<=n;i=i\*5){  res=res+n/i;  }  System.out.println("zeroes: "+res);  } |  |

**Binomial:-**

|  |  |
| --- | --- |
| nCr:  static int nCr(int n, int r) {  int[] dp = new int[r + 1];  dp[0] = 1;  for (int i = 1; i <= n; i++) {  for (int j = Math.min(i, r); j > 0; j--) {  dp[j] = dp[j] + dp[j - 1];  }  }  return dp[r];  } | Combination:  static int nCr(int n, int r) {  int res = 1;  for (int i = n; i >= (n - r + 1); i--) {  res = res \* i;  }  for (int i = 2; i <= r; i++) {  res = res / i;  }  return res;  } |
| Permutation of String:  static void permute(String str, int i) {  if (i == str.length() - 1) {  System.out.println(str);  } else {  for (int j = i; j <= str.length() - 1; j++) {  str = swap(str, i, j);  permute(str, i + 1);  str = swap(str, i, j);  }  }  }  static String swap(String a, int i, int j) {  char temp;  char[] charArray = a.toCharArray();  temp = charArray[i];  charArray[i] = charArray[j];  charArray[j] = temp;  return String.valueOf(charArray);  } |  |

**Catalan:**

|  |  |
| --- | --- |
| static int catalanDP(int n) {  // Table to store results of subproblems  int dp[] = new int[n + 2];  // Initialize first two values in table  dp[0] = 1;  dp[1] = 1;  // Fill entries in dp[]  // using recursive formula  for (int i = 2; i <= n; i++) {  dp[i] = 0;  for (int j = 0; j < i; j++) {  dp[i] += dp[j] \* dp[i - j - 1];  }  }  // Return last entry  return dp[n];  } |  |

**Divisors:**

|  |  |
| --- | --- |
| All divisors:  static void printDivisors(int n) {  for (int i = 1; i \* i <= n; i++) {  if (n % i == 0) {  System.out.print(i + " ");  if (i != n / i) {  System.out.print((n / i) + " ");  }  }  }  } | Count divisors:  static int countDivisors(int n) {  int res = 1;  for (int i = 2; i \* i <= n; i++) {  int cnt = 0;  while (n % i == 0) {  n = n / i;  cnt++;  }  res = res \* (cnt + 1);  }  if (n >= 2) {  res = res \* 2;  }  return res;  } } |
|  |  |

**Fibonacci:**

|  |  |
| --- | --- |
| Fib:  //Fm+n=FmFn+1+Fm-1Fn;  static int fib(int n) {  if (n == 0 || n == 1) {  return 1;  } else {  return fib(n - 1) + fib(n - 2);  }  } | //Cassini  //Fn+1\*Fn-1-Fn^2=(-1)^n;  double nplusone = fib(5);  double nminusone = fib(3);  double nsqr = Math.pow(fib(4), 2);  double res=Math.pow(-1,4);  double result = nplusone \* nminusone - nsqr;  System.out.println("" + result);  System.out.println("" + res); |
| DudenyCow:  static int count(int n) {  if (n == 0 || n == 1) {  return 1;  } else {  return count(n - 1) + count(n - 2);  }  } | FibomodM:  static int fibMod(int n, int m)  {  ArrayList<Integer> fib=new ArrayList<>();  fib.add(0);  fib.add(1);  int pis=1;  for(int i=2;i<m\*m;i++)  {  fib.add(fib.get(i-1)+fib.get(i-2));  if(fib.get(i)%m==1 && fib.get(i-1)%m==0){  pis=i-1;  break;  }  }  return fib.get(n%pis)%m;    } |
| GCD(Fm,Fn)=Fgcd(m,n):-  static int gcd(int a, int b)  {  // Everything divides 0  if (a == 0)  return b;  if (b == 0)  return a;    if (a == b)  return a;  if (a > b)  return gcd(a-b, b);  return gcd(a, b-a);  } | Binary strings with no two consecutive ones:-  public class BinaryStringswithNo2consecutive1 {  static int fib(int n) {  if (n == 0 || n == 1) {  return 1;  } else {  return fib(n - 1) + fib(n - 2);  }  }  public static void main(String[] args) {  int n=3;  int count=fib(n+2);  System.out.println(""+count);  }  } |

**GCD & LCM:**

|  |  |
| --- | --- |
| Euclid:  static int gcd(int a, int b)  {  if (b == 0)  return a;  return gcd(b, a % b);  } | Extended Euclid:  public static int gcdExtended(int a, int b, int x, int y)  {  // Base Case  if (a == 0)  {  x = 0;  y = 1;  return b;  }  int x1=1, y1=1;  // To store results of recursive call  int gcd = gcdExtended(b%a, a, x1, y1);  // Update x and y using results of recursive  // call  x = y1 - (b/a) \* x1;  y = x1;  return gcd;  } |
| // Recursive function to return gcd of a and b:  static int gcd(int a, int b)  {  // Everything divides 0  if (a == 0)  return b;  if (b == 0)  return a;    // base case  if (a == b)  return a;    // a is greater  if (a > b)  return gcd(a-b, b);  return gcd(a, b-a);  } | LCM:  static int lcm(int a, int b)  {  return (a / gcd(a, b)) \* b;  } |

**Geometric:**

|  |  |
| --- | --- |
| Orientation:  public static int orientation(Point p, Point q, Point r) {  int val = (q.y - p.y) \* (r.x - q.x)  - (q.x - p.x) \* (r.y - q.y);  if (val == 0) {  return 0; // collinear  }  return (val > 0) ? 1 : 2; // clock or counterclock wise  } |  |

**Number Theory:**

|  |  |
| --- | --- |
| // Function to return GCD of a and b  static int gcd(int a, int b)  {  if (a == 0)  return b;  return gcd(b % a, a);  }  // A simple method to evaluate  // Euler Totient Function  static int phi(int n)  {  int result = 1;  for (int i = 2; i < n; i++)  if (gcd(i, n) == 1)  result++;  return result;  } |  |

**Prime:**

|  |  |
| --- | --- |
| Check prime:  static boolean isPrime(int n) {  if (n == 1) {  return false;  }  if (n == 2 || n == 3) {  return true;  }  if (n % 2 == 0 || n % 3 == 0) {  return false;  }  for (int i = 5; i \* i <= n; i = i + 6) {  if (n % i == 0 || n % (i + 2) == 0) {  return false;  }  }  return true;  } |  |

**Prime Factorization:**

|  |  |
| --- | --- |
| Prime factors:  static void printPrimeFactors(int n) {  if (n <= 1) {  return;  }  while (n % 2 == 0) {  System.out.print(2 + " ");  n = n / 2;  }  while (n % 3 == 0) {  System.out.print(3 + " ");  n = n / 3;  }  for (int i = 5; i \* i <= n; i = i + 6) {  while (n % i == 0) {  System.out.print(i + " ");  n = n / i;  }  while (n % (i + 2) == 0) {  System.out.print((i + 2) + " ");  n = n / (i + 2);  }  }  if (n > 3) {  System.out.print(n + " ");  }  System.out.println();  } | Smallest prime factor:  public static void smallestPrimeFactor(int n) {  // Create a vector to store least primes.  // Initialize all entries as 0.  int[] least\_prime = new int[n + 1];  // We need to print 1 for 1.  System.out.print(1 + " ");  for (int i = 2; i <= n; i++) {  // least\_prime[i] == 0  // means it i is prime  if (least\_prime[i] == 0) {  System.out.print(i + " ");  // mark it as a divisor for all its  // multiples if not already marked  for (int j = i \* i; j <= n; j += i) {  if (least\_prime[j] == 0) {  least\_prime[j] = i;  }  }  } else {  System.out.print(least\_prime[i] + " ");  }  }  } |

**Bit magic:**

|  |  |
| --- | --- |
| Operations:  System.out.println(a&b);  System.out.println(a|b);  System.out.println(a^b);  // left shift  System.out.println(a << 1);  System.out.println(a << 2);    // right shift  System.out.println(a >> 1);  System.out.println(a >> 2);  // negation Operator  System.out.println(~a); | Binary to gray:  public static int greyConverter(int n) {  return n ^ (n >> 1);  }  Gray to binary:  public static int binaryConverter(int n) {  int res = 0;  while (n > 0) {  res ^= n;  n >>= 1;  }  return res;  } |
| Unique number:  private static int unique(int[] arr) {  int n = arr.length;  int cumxor = 0;  for (int i = 0; i < n; i++) {  cumxor = cumxor ^ arr[i];  }  return cumxor;  } | Clear ith bit:  private static int clearithbit(int n, int i) {  int mask = ~(1 << i);  n = n & mask;  return n;  } |
| Get ith bit:  private static int getithbit(int n, int i) {  int mask = 1 << i;  int res = n & mask;  if (res == 0) {  return 0;  } else {  return 1;  }  } | Set ith bit:  private static int setithbit(int n, int i) {  int mask = 1 << i;  n = n | mask;  return n;  } |
| Check Kth bit:  public static void isKthBitSet(int n,  int k) {  if ((n & (1 << (k - 1))) == 1) {  System.out.print("SET");  } else {  System.out.print("NOT SET");  }  } | Count set bits:  static int countSetBits(int n) {  int count = 0;  while (n > 0) {  n &= (n - 1);  count++;  }  return count;  } |
| Power of two:  static boolean isPow2(int n) {  if (n == 0) {  return true;  }  return ((n & (n - 1)) == 0);  } | Power:  static int power(int x, int y) {  // Initialize result  int res = 1;  while (y > 0) {  // If y is odd,  // multiply  // x with result  if ((y & 1) == 1) {  res = res \* x;  }  // y must be even now  y = y >> 1; // y = y/2  x = x \* x; // Change x to x^2  }  return res;  } |
| Even odd:  static void evenodd(int a) {  System.out.println((a & 1) == 0 ? "EVEN" : "ODD");  } | Mul by 2:  static void mulby2(int a) {  System.out.println(a << 1);  } |
| Div by 2:  static void divby2(int a) {  System.out.println(a >> 1);  } | Swap:  static void swap(int a, int b) {  a ^= b;  b ^= a;  a ^= b;  System.out.println(a + " "+b);  } |

**Graph algo:**

|  |  |
| --- | --- |
| DFS:  public class DFS {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  static void DFSRec(ArrayList<ArrayList<Integer>> adj, int s, boolean[] visited) {  visited[s]=true;  System.out.print(" "+s);  for(int nbr:adj.get(s)){  if(visited[nbr]==false){  visited[nbr]=true;  DFSRec(adj,nbr,visited);  }  }  }  static void DFS(ArrayList<ArrayList<Integer>> adj, int V, int s) {  boolean[] visited = new boolean[V];  DFSRec(adj, s, visited);  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 1, 4);  addEdge(adj, 3, 4);  System.out.println("Following is Depth First Traversal: ");  DFS(adj, V, 0);  }  } | BFS:  public class BFS {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  static void BFS(ArrayList<ArrayList<Integer>> adj, int V, int s) {  boolean[] visited = new boolean[V];  visited[s] = true;  //System.out.println(""+s);  Queue<Integer> q = new LinkedList<>();  q.add(s);  while (q.size() > 0) {  int f=q.poll();  System.out.println(f);  for (int nbr : adj.get(f)) {  if(visited[nbr]==false){  visited[nbr]=true;  q.add(nbr);  }  }  }  }  public static void main(String[] args) {  int V = 5;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 1, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 3, 4);  addEdge(adj, 2, 4);  System.out.println("Following is Breadth First Traversal: ");  BFS(adj, V, 0);  }  } |
| Shortest path(Dijkastra):  public class ShortestPathinWeights {  static class Edge {  int src;  int nbr;  int wt;  Edge(int src, int nbr, int wt) {  this.src = src;  this.nbr = nbr;  this.wt = wt;  }  }  public static class Pair implements Comparable<Pair> {  int v;  String psf;  int wsf;  Pair(int v, String psf, int wsf) {  this.v = v;  this.psf = psf;  this.wsf = wsf;  }  public int compareTo(Pair o) {  return this.wsf - o.wsf;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2, int wt) {  graph[v1].add(new Edge(v1, v2, wt));  graph[v2].add(new Edge(v2, v1, wt));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  int edges = 9;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>();  }  addEdge(graph, 0, 1, 10);  addEdge(graph, 1, 2, 10);  addEdge(graph, 2, 3, 10);  addEdge(graph, 0, 3, 40);  addEdge(graph, 3, 4, 2);  addEdge(graph, 4, 5, 3);  addEdge(graph, 5, 6, 3);  addEdge(graph, 4, 6, 8);  addEdge(graph, 2, 5, 5);  int src = 0;  boolean[] visited=new boolean[vtces];  PriorityQueue<Pair> pq=new PriorityQueue<>();  pq.add(new Pair(src,src+"",0));    while(pq.size()>0){  Pair rem=pq.remove();  if(visited[rem.v]==true){  continue;  }  visited[rem.v]=true;  System.out.println(rem.v+" via "+rem.psf+" @ "+rem.wsf);  for(Edge e:graph[rem.v]){  if(visited[e.nbr]==false){  pq.add(new Pair(e.nbr,rem.psf+e.nbr,rem.wsf+e.wt));  }  }  }  }  } | BFS Path:  public class BFSPath {  static class Edge {  int src;  int nbr;  Edge(int src, int nbr) {  this.src = src;  this.nbr = nbr;  }  }  static class Pair {  int v;  String psf;  Pair(int v, String psf) {  this.v = v;  this.psf = psf;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2) {  graph[v1].add(new Edge(v1, v2));  graph[v2].add(new Edge(v2, v1));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>();  }  int edges = 8;  addEdge(graph, 0, 1);  addEdge(graph, 1, 2);  addEdge(graph, 2, 3);  addEdge(graph, 0, 3);  addEdge(graph, 3, 4);  addEdge(graph, 4, 5);  addEdge(graph, 5, 6);  addEdge(graph, 4, 6);  int src = 2;  ArrayDeque<Pair> q = new ArrayDeque<>();  boolean[] visited = new boolean[vtces];  //visited[src]=true;  q.add(new Pair(src, src + ""));  while (!q.isEmpty()) {  Pair rem = q.removeFirst();  if (visited[rem.v] == true) {  continue;  }  visited[rem.v] = true;  System.out.println(rem.v + "@" + rem.psf);  for (Edge e : graph[rem.v]) {  if (visited[e.nbr] == false) {  q.add(new Pair(e.nbr, rem.psf + e.nbr));  }  }  }  }  } |
| Get connected components:  public class GetConnectedComponents {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 1, 4);  addEdge(adj, 3, 4);  System.out.println("Following is Depth First Traversal: ");  //DFS(adj, V, 0);  ArrayList<ArrayList<Integer>> comps = new ArrayList<>();  boolean[] visited = new boolean[V];  for (int i = 0; i < V; i++) {  if (visited[i] == false) {  ArrayList<Integer> comp = new ArrayList<>();  drawTree(adj, i, comp, visited);  comps.add(comp);  }  }  System.out.println(comps);  }  static void drawTree(ArrayList<ArrayList<Integer>> adj, int src, ArrayList<Integer> comp, boolean[] visited) {  visited[src] = true;  comp.add(src);  for (int nbr : adj.get(src)) {  if (visited[nbr] == false) {  drawTree(adj, nbr, comp, visited);  }  }  }  } | Is connected:  public class IsConnected {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 1, 4);  addEdge(adj, 3, 4);    ArrayList<ArrayList<Integer>> comps = new ArrayList<>();  boolean[] visited = new boolean[V];  for (int i = 0; i < V; i++) {  if (visited[i] == false) {  ArrayList<Integer> comp = new ArrayList<>();  drawTree(adj, i, comp, visited);  comps.add(comp);  }  }  System.out.println(comps.size()==1);  }    static void drawTree(ArrayList<ArrayList<Integer>> adj, int src, ArrayList<Integer> comp, boolean[] visited) {  visited[src] = true;  comp.add(src);  for (int nbr : adj.get(src)) {  if (visited[nbr] == false) {  drawTree(adj, nbr, comp, visited);  }  }  }  } |
| Is bipartite:  public class IsBipartite {  static class Edge {  int src;  int nbr;  int wt;  Edge(int src, int nbr, int wt) {  this.src = src;  this.nbr = nbr;  this.wt = wt;  }  }  public static class Pair {  int v, level;  String psf;  Pair(int v, String psf, int level) {  this.v = v;  this.psf = psf;  this.level = level;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2, int wt) {  graph[v1].add(new Edge(v1, v2, wt));  graph[v2].add(new Edge(v2, v1, wt));  }  public static void main(String[] args) {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>(0);  }  int edges = 9;  addEdge(graph, 0, 1, 10);  addEdge(graph, 1, 2, 10);  addEdge(graph, 2, 3, 10);  addEdge(graph, 0, 3, 10);  addEdge(graph, 3, 4, 10);  addEdge(graph, 4, 5, 10);  addEdge(graph, 5, 6, 10);  addEdge(graph, 4, 6, 10);  int[] visited = new int[vtces];  Arrays.fill(visited, -1);  for (int i = 0; i < vtces; i++) {  if (visited[i] == -1) {  boolean isBi = check(graph, i, visited);  if (isBi == false) {  System.out.println(false);  return;  }  }  }  System.out.println(true);  }  static boolean check(ArrayList<Edge>[] graph, int src, int[] visited) {  ArrayDeque<Pair> q = new ArrayDeque<>();  q.add(new Pair(src, src + "", 0));  while (q.size() > 0) {  Pair rem = q.removeFirst();  if (visited[rem.v] != -1) {  if (rem.level != visited[rem.v]) {  return false;  }  } else {  visited[rem.v] = rem.level;  }  for (Edge e : graph[rem.v]) {  if (visited[e.nbr] == -1) {  q.add(new Pair(e.nbr, rem.psf + e.nbr, rem.level + 1));  }  }  }  return true;  }  } | Hamiltonian:  public class Hamiltonian {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 1, 2);  addEdge(adj, 2, 3);  addEdge(adj, 0, 3);  addEdge(adj, 3, 4);  addEdge(adj, 4, 5);  addEdge(adj, 5, 6);  addEdge(adj, 4, 6);  addEdge(adj, 2, 5);  int src = 0;  HashSet<Integer> visited = new HashSet<>();  hamil(adj, src, visited, src + "", src);  }  public static void hamil(ArrayList<ArrayList<Integer>> adj, int src, HashSet<Integer> visited,  String psf, int osrc) {  if (visited.size() == adj.size() - 1) {  System.out.print(psf);  boolean closingEdge = false;  for (int nbr : adj.get(src)) {  if (nbr == osrc) {  closingEdge = true;  break;  }  }  if (closingEdge == true) {  System.out.println("\*");  } else {  System.out.println(".");  }  return;  }  visited.add(src);  for (int nbr : adj.get(src)) {  if (visited.contains(nbr) == false) {  hamil(adj, nbr, visited, psf + nbr, osrc);  }  }  visited.remove(src);  }  } |
| Print All paths:  public class PrintAllPaths {  static void addEdge(ArrayList<ArrayList<Integer>> adj, int u, int v) {  adj.get(u).add(v);  adj.get(v).add(u);  }  static void dfs(ArrayList<ArrayList<Integer>> adj, int V, int src, int dest) {  boolean[] visited = new boolean[V];  printAllPaths(adj, src, dest, visited, "");  }  static void printAllPaths(ArrayList<ArrayList<Integer>> adj, int src, int dest, boolean[] visited, String psf) {  if (src == dest) {  System.out.println("" + psf);  return;  }  visited[src] = true;  for (int nbr : adj.get(src)) {  if (visited[nbr] == false) {  printAllPaths(adj, nbr, dest, visited, psf + " " + nbr);  }  }  }  public static void main(String[] args) {  int V = 7;  ArrayList<ArrayList<Integer>> adj = new ArrayList<ArrayList<Integer>>(V);  for (int i = 0; i < V; i++) {  adj.add(new ArrayList<Integer>());  }  addEdge(adj, 0, 1);  addEdge(adj, 0, 2);  addEdge(adj, 2, 3);  addEdge(adj, 1, 3);  addEdge(adj, 1, 4);  addEdge(adj, 3, 4);  addEdge(adj, 2, 4);  System.out.println("Following is Depth First Traversal: ");  dfs(adj, V, 0, 4);  }  } | Prim(MST):  public class MinimumWireConnectPCs {  static class Edge {  int src;  int nbr;  int wt;  Edge(int src, int nbr, int wt) {  this.src = src;  this.nbr = nbr;  this.wt = wt;  }  }  static class Pair implements Comparable<Pair> {  int v;  int av;  int wt;  Pair(int v, int av, int wt) {  this.v = v;  this.av = av;  this.wt = wt;  }  public int compareTo(Pair o) {  return this.wt - o.wt;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2, int wt) {  graph[v1].add(new Edge(v1, v2, wt));  graph[v2].add(new Edge(v2, v1, wt));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>();  }  int edges = 8;  addEdge(graph, 0, 1, 10);  addEdge(graph, 1, 2, 10);  addEdge(graph, 2, 3, 10);  addEdge(graph, 0, 3, 40);  addEdge(graph, 3, 4, 2);  addEdge(graph, 4, 5, 3);  addEdge(graph, 5, 6, 3);  addEdge(graph, 4, 6, 8);  PriorityQueue<Pair> pq = new PriorityQueue<>();  pq.add(new Pair(0, -1, 0));  boolean[] visited = new boolean[vtces];  while (pq.size() > 0) {  Pair rem = pq.remove();  if (visited[rem.v] == true) {  continue;  }  visited[rem.v] = true;  if (rem.av != -1) {  System.out.println("[" + rem.v + "-" + rem.av + "@" + rem.wt + "]");  }  for (Edge e : graph[rem.v]) {  if (visited[e.nbr] == false) {  pq.add(new Pair(e.nbr, rem.v, e.wt));  }  }  }  }  } |
| Topological sort:  public class OrderOfCompilation {  static class Edge {  int src;  int nbr;  Edge(int src, int nbr) {  this.src = src;  this.nbr = nbr;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2) {  graph[v1].add(new Edge(v1, v2));  graph[v2].add(new Edge(v2, v1));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>();  }  int edges = 7;  addEdge(graph, 0, 1);  addEdge(graph, 1, 2);  addEdge(graph, 2, 3);  addEdge(graph, 0, 3);  addEdge(graph, 4, 5);  addEdge(graph, 5, 6);  addEdge(graph, 4, 6);  boolean[] visited = new boolean[vtces];  Stack<Integer> st = new Stack<>();  for (int i = 0; i < vtces; i++) {  if (visited[i] == false) {  topologicalSort(graph, i, visited, st);  }  }  while (st.size() > 0) {  System.out.println(st.pop());  }  }  public static void topologicalSort(ArrayList<Edge>[] graph, int src, boolean[] visited, Stack<Integer> st) {  visited[src] = true;  for (Edge e : graph[src]) {  if (visited[e.nbr] == false) {  topologicalSort(graph, e.nbr, visited, st);  }  }  st.push(src);  }  } | Spread Infection: (Time dependent)  public class SpreadofInfection {  static class Edge {  int src;  int nbr;  int wt;  Edge(int src, int nbr, int wt) {  this.src = src;  this.nbr = nbr;  this.wt = wt;  }  }  static class Pair {  int v;  int time;  Pair(int v, int time) {  this.v = v;  this.time = time;  }  }  static void addEdge(ArrayList<Edge>[] graph, int v1, int v2, int wt) {  graph[v1].add(new Edge(v1, v2, wt));  graph[v2].add(new Edge(v2, v1, wt));  }  public static void main(String[] args) throws Exception {  int vtces = 7;  ArrayList<Edge>[] graph = new ArrayList[vtces];  for (int i = 0; i < vtces; i++) {  graph[i] = new ArrayList<>(0);  }  int edges = 8;  addEdge(graph, 0, 1, 10);  addEdge(graph, 1, 2, 10);  addEdge(graph, 2, 3, 10);  addEdge(graph, 0, 3, 10);  addEdge(graph, 3, 4, 10);  addEdge(graph, 4, 5, 10);  addEdge(graph, 5, 6, 10);  addEdge(graph, 4, 6, 10);  int src = 6;  int t = 3;  ArrayDeque<Pair> q = new ArrayDeque<>();  q.add(new Pair(src, 1));  int[] visited = new int[vtces];  int count = 0;  while (q.size() > 0) {  Pair rem = q.removeFirst();  if (visited[rem.v] > 0) {  continue;  }  visited[rem.v] = rem.time;  if (rem.time > t) {  break;  }  count++;  for (Edge e : graph[rem.v]) {  if (visited[e.nbr] == 0) {  q.add(new Pair(e.nbr, rem.time + 1));  }  }  }  System.out.println("" + count);  }  } |
| TopologicalDFS:  public class TopologicalDFS {  static class Graph {  int V;  ArrayList<Integer>[] list;  public Graph(int v) {  V = v;  list = new ArrayList[v];  for (int i = 0; i < v; i++) {  list[i] = new ArrayList<>();  }  }  void addEdge(int i, int j) {  list[i].add(j);  }  void dfs(int node, boolean[] visited, ArrayList<Integer> ordering) {  visited[node] = true;  for (int nbr : list[node]) {  if (!visited[nbr]) {  dfs(nbr, visited, ordering);  }  }  //at this point  ordering.add(node);  }  //Complete this method  void dfs\_topological\_sort() {  boolean[] visited = new boolean[V];  ArrayList<Integer> ordering = new ArrayList<>();  //we call dfs from every node if it not visited  for (int i = 0; i < V; i++) {  if (!visited[i]) {  dfs(i, visited, ordering);  }  }  Collections.reverse(ordering);  //finaly print the ordeirng  for (int node : ordering) {  System.out.print(node + " ");  }  System.out.println();  }  }  public static void main(String[] args) {  Graph g = new Graph(6);  g.addEdge(0, 2);  g.addEdge(2, 3);  g.addEdge(3, 5);  g.addEdge(4, 5);  g.addEdge(1, 4);  g.addEdge(1, 2);  g.dfs\_topological\_sort();  }  } | Topological BFS:  public class TopologicalBFS {  static class Graph{  int V;  ArrayList<Integer>[] list;  public Graph(int v){  V = v;  list = new ArrayList[v];  for(int i = 0; i < v; i++){  list[i] = new ArrayList<>();  }  }  void addEdge(int i, int j){  list[i].add(j);  }  //Complete this method  void topological\_sort() {  int[] indegree = new int[V];  //Iterate over all the edges to find the right indegree  for(int i=0;i<V;i++){  for(int nbr : list[i]){  indegree[nbr]++;  }  }  //bfs  Queue<Integer> q = new LinkedList<>();  //init the q with nodes having 0 indegree  for(int i=0;i<V;i++){  if(indegree[i]==0){  q.add(i);  }  }  //start popping  while(!q.isEmpty()){  int node = q.poll();  System.out.print(node + " ");  //iterate over the nbrs of this node and reducec their indegree by 1  for(int nbr : list[node]){  indegree[nbr]--;  if(indegree[nbr]==0){  q.add(nbr);  }  }  }  }  }  public static void main(String[] args){  Graph g = new Graph(6);  g.addEdge(0, 2);  g.addEdge(2, 3);  g.addEdge(3, 5);  g.addEdge(4, 5);  g.addEdge(1, 4);  g.addEdge(1, 2);  g.topological\_sort();  }  } |
| Dfs Trees And BackEdges:  public class DfsTreesAndBackEdges {  static final int N = (int) (1e5 + 1);  static ArrayList<Integer>[] gr = new ArrayList[N];  static boolean[] vis = new boolean[N];  static boolean cycle = false;  static {  for (int i = 0; i < gr.length; i++) {  gr[i] = new ArrayList<>();  }  }  static void dfs(int cur, int par) {  System.out.println(cur + " ");  vis[cur] = true;  for (int x : gr[cur]) {  if (vis[x] == false) {  dfs(x, cur);  } else if (x != par) {  System.out.println(cur + "-" + x);  cycle = true;  }  }  }  static void dfs1(int cur) {  System.out.print("-" + cur);  vis[cur] = true;  for (int nbr : gr[cur]) {  if (vis[nbr] == false) {  dfs1(nbr);  }  }  }  public static void main(String[] args) {  Scanner scn = new Scanner(System.in);  int n = 7, m = 8;  // for (int i = 0; i < m; i++) {  // int x = scn.nextInt(), y = scn.nextInt();  // gr[x].add(y);  // gr[y].add(x);  //  // }  gr[1].add(2);  gr[2].add(1);  gr[1].add(3);  gr[3].add(1);  gr[2].add(3);  gr[3].add(2);  gr[2].add(4);  gr[4].add(2);  gr[4].add(5);  gr[5].add(4);  gr[5].add(6);  gr[6].add(5);  gr[6].add(7);  gr[7].add(6);  gr[7].add(4);  gr[4].add(7);  for (int i = 1; i <= n; i++) {  if (!vis[i]) {  dfs(i, 0);  //dfs1(i);  }  }  if (cycle) {  System.out.print("Yes cycle found");  } else {  System.out.print("Not found");  }  }  } | Dfs Trees And BackEdges directed:  public class DfsTreesAndBackEdgesDirected {  static final int N = (int) (1e5 + 1);  static ArrayList<Integer>[] gr = new ArrayList[N];  static int[] vis = new int[N];  static boolean cycle = false;  static {  for (int i = 0; i < N; i++) {  gr[i] = new ArrayList<>();  }  }    static void dfs(int cur, int par) {  vis[cur] = 1;  for (int x : gr[cur]) {  if (vis[x] == 0) {  dfs(x, cur);  }  else if (x != par && vis[x] == 1) {  // backedge  System.out.println(cur+"-"+x);  cycle = true;  }  }  // visited and not in call stack  vis[cur] = 2;  return;  }  public static void main(String[] args) {  Scanner scn = new Scanner(System.in);  int n = 7, m = 8;  // for (int i = 0; i < m; i++) {  // int x = scn.nextInt(), y = scn.nextInt();  // gr[x].add(y);  // gr[y].add(x);  //  // }  gr[1].add(3);  gr[1].add(2);  gr[2].add(3);  gr[2].add(4);  gr[4].add(5);  gr[5].add(6);  gr[6].add(7);  gr[7].add(4);  for (int i = 1; i <= n; i++) {  if (vis[i] == 0) {  dfs(i, 0);  }  }  if (cycle) {  System.out.print("Yes cycle found");  } else {  System.out.print("Not found");  }  }  } |
| Articulation Point and Bridges:  public class ArticulationPointAndBridges {  static class Pair {  int first, second;  Pair(int a, int b) {  first = a;  second = b;  }  }  static final int N = (int) (1e5 + 1);  static ArrayList<Integer>[] gr = new ArrayList[N];  static Set<Integer> arti\_points = new HashSet<>();  static {  for (int i = 0; i < N; i++) {  gr[i] = new ArrayList<>();  }  }  static int tme = 1;  static boolean[] vis = new boolean[N];  static int[] disc = new int[N];  static int[] low = new int[N];  static ArrayList<Pair> bridges = new ArrayList<>();  static void dfs(int cur, int par) {  vis[cur] = true;  disc[cur] = low[cur] = tme++;  int child = 0;  for (int x : gr[cur]) {  if (!vis[x]) {  dfs(x, cur);  child++;  // we know low and disc of x  low[cur] = Math.min(low[cur], low[x]);  // bridges  if (low[x] > disc[cur]) {  bridges.add(new Pair(cur, x));  }  // articulation points  if (par != 0 && low[x] >= disc[cur]) {  arti\_points.add(cur);  }  } else if (x != par) {  // backedge  low[cur] = Math.min(low[cur], disc[x]);  }  }  // root is an arti or not  if (par == 0 && child > 1) {  arti\_points.add(cur);  }  }  static void addEdge(int x, int y) {  gr[x].add(y);  gr[y].add(x);  }  public static void main(String[] args) {  Scanner scn = new Scanner(System.in);  int n = 5, m = 5;  addEdge(1, 2);  addEdge(2, 3);  addEdge(2, 4);  addEdge(3, 4);  addEdge(4, 5);  dfs(1, 0);  for (int x : arti\_points) {  System.out.println(x);  }  for (Pair x : bridges) {  System.out.println(x.first + " " + x.second);  }  }  } | Cycle In Graph:  public class CycleInGraph {  static final int N = (int) (1e5 + 1);  static ArrayList<Integer>[] gr = new ArrayList[N];  static int[] vis = new int[N];  static int[] Par = new int[N];  static {  for (int i = 0; i < N; i++) {  gr[i] = new ArrayList<>();  }  }  static boolean cycle = false;  static void dfs(int cur, int par) {  // visited and in call stack  vis[cur] = 1;  Par[cur] = par;  for (int x : gr[cur]) {  if (vis[x] == 0) {  dfs(x, cur);  } else if (x != par && vis[x] == 1) {  // backedge  cycle = true;  int u = cur, v = x;  while (u != v) {  System.out.print(u + " ");  u = Par[u];  }  System.out.print(u + " ");  System.exit(0);  }  }  // visited and not in call stack  vis[cur] = 2;  return;  }  public static void main(String[] args) {  Scanner scn = new Scanner(System.in);  int n = 7, m = 8;  // int n = scn.nextInt(), m = scn.nextInt();  //  // for (int i = 0; i < m; i++) {  // int x = scn.nextInt(), y = scn.nextInt();  // gr[x].add(y);  // gr[y].add(x);  //  // }  gr[1].add(2);  gr[2].add(1);  gr[1].add(3);  gr[3].add(1);  gr[2].add(3);  gr[3].add(2);  gr[2].add(4);  gr[4].add(2);  gr[4].add(5);  gr[5].add(4);  gr[5].add(6);  gr[6].add(5);  gr[6].add(7);  gr[7].add(6);  gr[7].add(4);  gr[4].add(7);  for (int i = 1; i <= n; i++) {  if (vis[i] == 0) {  dfs(i, 0);  }  }  if (cycle) {  System.out.print("Yes cycle found");  } else {  System.out.print("Not found");  }  }  } |
| DSU:  public class DSU\_1 {  static class Pair {  public int first, second;  public Pair(int a, int b) {  first = a;  second = b;  }  }  static class Graph {  int V;  List<Pair> l;  public Graph(int V) {  this.V = V;  this.l = new ArrayList<>();  }  void addEdge(int u, int v) {  l.add(new Pair(u, v));  }  // Find  int findSet(int i, int parent[]) {  if (parent[i] == -1) {  return i;  }  return findSet(parent[i], parent);  }  //Union  void union\_set(int x, int y, int parent[]) {  int s1 = findSet(x, parent);  int s2 = findSet(y, parent);  if (s1 != s2) {  parent[s1] = s2;  }  }  boolean contains\_cycle() {  //DSU Logic to check if graph contains cycle or not  int parent[] = new int[V];  for (int i = 0; i < V; i++) {  parent[i] = -1;  }  //iterate over the edge list  for (Pair edge : l) {  int i = edge.first;  int j = edge.second;  int s1 = findSet(i, parent);  int s2 = findSet(j, parent);  if (s1 != s2) {  union\_set(s1, s2, parent);  } else {  System.out.println("Same parents" + s1 + " and " + s2);  return true;  }  }  return false;  }  }  public static void main(String[] args) {  Graph g = new Graph(4);  g.addEdge(0, 1);  g.addEdge(1, 2);  g.addEdge(2, 3);  g.addEdge(3, 0);  System.out.print(g.contains\_cycle());  }  } | Eular tour:  public class EulerTour {  static final int N = (int) (1e5 + 1);  static ArrayList<Integer>[] gr = new ArrayList[N];  static {  for (int i = 0; i < N; i++) {  gr[i] = new ArrayList<>();  }  }  static void dfs1(int cur, int par) {  // time in  System.out.print(cur + " ");  for (int x : gr[cur]) {  if (x != par) {  // x is child node  dfs1(x, cur);  }  }  // time out  System.out.print(cur + " ");  }  static void dfs2(int cur, int par) {  System.out.print(cur + " ");  for (int x : gr[cur]) {  if (x != par) {  // x is child node  dfs2(x, cur);  System.out.print(cur + " ");  }  }  }  public static void main(String[] args) {  Scanner scn = new Scanner(System.in);  int n = scn.nextInt();  for (int i = 0; i < n - 1; i++) {  int x = scn.nextInt(), y = scn.nextInt();  gr[x].add(y);  gr[y].add(x);  }  // dfs1(1, 0);  dfs2(1, 0);  }  } |
| Flood fill largest island:  static int[][] arr = {  {0, 0, 1, 1, 1, 1, 1, 1},  {0, 0, 1, 1, 1, 1, 1, 1},  {0, 0, 0, 0, 0, 0, 0, 0},  {1, 1, 0, 0, 0, 0, 1, 0},  {1, 0, 1, 1, 0, 1, 1, 0},  {1, 0, 1, 0, 0, 1, 1, 0},  {1, 0, 0, 1, 1, 1, 1, 0},  {1, 1, 0, 1, 1, 1, 1, 0}  };  static int[] dx = {-1, 1, 0, 0};  static int[] dy = {0, 0, 1, -1};  public static StringBuilder psf = new StringBuilder();  public static void main(String[] args) {  HashSet<String> set = new HashSet<>();  int c = 0;  for (int i = 0; i < arr.length; i++) {  for (int j = 0; j < arr[0].length; j++) {  if (arr[i][j] == 1) {  c++;  psf = new StringBuilder();  psf.append("X");  dfs(i, j);  set.add(psf.toString());  }  }  }  int largest=0;  for (String s : set) {  if(s.length()>largest){  largest=s.length();  }  System.out.println(" " + s);  }  System.out.println(" " + largest);  }  private static void dfs(int i, int j) {  arr[i][j] = 0;  for (int k = 0; k < 4; k++) {  int ii = i + dx[k];  int jj = j + dy[k];  if (ii >= 0 && ii < arr.length && jj >= 0 && jj < arr[0].length && arr[ii][jj] == 1) {  psf.append("C");  dfs(ii, jj);  }  }  } | Flood fill no of color:  static int[][] arr = {  {0, 0, 1, 1, 1, 1, 1, 1},  {0, 0, 1, 1, 1, 1, 1, 1},  {0, 0, 0, 0, 0, 0, 0, 0},  {1, 1, 0, 0, 0, 0, 1, 0},  {1, 0, 1, 1, 0, 1, 1, 0},  {1, 0, 1, 0, 0, 1, 1, 0},  {1, 0, 0, 1, 1, 1, 1, 0},  {1, 1, 0, 1, 1, 1, 1, 0}  };  static int[] dx = {-1, 1, 0, 0};  static int[] dy = {0, 0, 1, -1};  public static void main(String[] args) {  int color = 3;  for (int i = 0; i < arr.length; i++) {  for (int j = 0; j < arr[0].length; j++) {  if (arr[i][j] == 1) {  //color++;  dfs(i, j, color);  color++;  }  }  }  for (int i = 0; i < arr.length; i++) {  for (int j = 0; j < arr[0].length; j++) {  System.out.print(arr[i][j] + " ");  }  System.out.println("");  }  }  private static void dfs(int i, int j, int color) {  arr[i][j] = color;  for (int k = 0; k < 4; k++) {  int ii = i + dx[k];  int jj = j + dy[k];  if (ii >= 0 && ii < arr.length && jj >= 0 && jj < arr[0].length && arr[ii][jj] == 1) {  dfs(ii, jj, color);  }  }  } |
| Flood fill No of islands:  static int[][] arr = {  {0, 0, 1, 1, 1, 1, 1, 1},  {0, 0, 1, 1, 1, 1, 1, 1},  {0, 0, 0, 0, 0, 0, 0, 0},  {1, 1, 0, 0, 0, 0, 1, 0},  {1, 0, 1, 1, 0, 1, 1, 0},  {1, 0, 1, 0, 0, 1, 1, 0},  {1, 0, 0, 1, 1, 1, 1, 0},  {1, 1, 0, 1, 1, 1, 1, 0}  };  static int[] dx = {-1, 1, 0, 0};  static int[] dy = {0, 0, 1, -1};  public static void main(String[] args) {  int c = 0;  for (int i = 0; i < arr.length; i++) {  for (int j = 0; j < arr[0].length; j++) {  if (arr[i][j] == 1) {  c++;  dfs(i, j);  }  }  }  System.out.println(c);  }  private static void dfs(int i, int j) {  arr[i][j] = 0;  for (int k = 0; k < 4; k++) {  int ii = i + dx[k];  int jj = j + dy[k];  if (ii >= 0 && ii < arr.length && jj >= 0 && jj < arr[0].length && arr[ii][jj] == 1) {  dfs(ii, jj);  }  }  } | Flood fill No of distinct islands:  static int[][] arr = {  {0, 0, 1, 1},  {0, 0, 1, 1},  {1, 0, 0, 0},  {1, 1, 0, 0}};  // static int[][] arr = {  // {0, 0, 1, 1, 1, 1, 1, 1},  // {0, 0, 1, 1, 1, 1, 1, 1},  // {0, 0, 0, 0, 0, 0, 0, 0},  // {1, 1, 0, 0, 0, 0, 1, 0},  // {1, 0, 1, 1, 0, 1, 1, 0},  // {1, 0, 1, 0, 0, 1, 1, 0},  // {1, 0, 0, 1, 1, 1, 1, 0},  // {1, 1, 0, 1, 1, 1, 1, 0}  // };  static int[] dx = {0, 1, 0, -1};  static int[] dy = {1, 0, -1, 0};  public static StringBuilder psf = new StringBuilder();  public static void main(String[] args) {  HashSet<String> set = new HashSet<>();  int c = 0;  for (int i = 0; i < arr.length; i++) {  for (int j = 0; j < arr[0].length; j++) {  if (arr[i][j] == 1) {  c++;  psf = new StringBuilder();  psf.append("x");  dfs(i, j);  set.add(psf.toString());  }  }  }  int largest=0;  for (String s : set) {  if(s.length()>largest){  largest=s.length();  }  System.out.println(" " + s);  }  System.out.println(" " + largest);  }  private static void dfs(int row, int col) {  arr[row][col] = 0;      if (row - 1 >= 0 && arr[row - 1][col] == 1) {  psf.append("u");  dfs( row - 1, col);  }  if (col + 1 < arr[0].length && arr[row][col + 1] == 1) {  psf.append("r");  dfs( row, col + 1);  }  if (row + 1 < arr.length && arr[row + 1][col] == 1) {  psf.append("d");  dfs(row + 1, col);  }  if (col - 1 >= 0 && arr[row][col - 1] == 1) {  psf.append("l");  dfs(row, col - 1);  }    psf.append("z");    } |
| Floyd warshall:  public class FloydWarshal {  final static int INF = 10000;  static int[][] floyd\_warshall(int[][] graph){  int[][] dist = graph;  int V = graph.length;  //Phases, in kth phase we included vertices (1,2,...k) as intermediate vertices  for(int k=0;k<V;k++){  //Iterate over entire 2D Matrix  for(int i=0;i<V;i++){  for(int j=0;j<V;j++){  //if vertex k is included, and can we minimise the dist ?  if(dist[i][j] > dist[i][k] + dist[k][j]){  dist[i][j] = dist[i][k] + dist[k][j];  }  }  }  }  return dist;  }  public static void main(String[] args){  int[][] graph = {  {0,INF,-2,INF},  {4,0,3,INF},  {INF,INF,0,2},  {INF,-1,INF,0}  };  int[][] result = floyd\_warshall(graph);  for(int i=0;i<result.length;i++){  for(int j=0;j<result.length;j++){  System.out.print(result[i][j]+" ");  }  System.out.println();  }  }  } | Bellman ford:  public class BellmanFord {  static int[] bellman\_ford(int V,int src,int[][] edges){  //create a vector  int[] dist = new int[V+1];  Arrays.fill(dist, Integer.MAX\_VALUE);  dist[src] = 0;  //relax all edges v-1 times  for(int i=0;i<V-1;i++){  for(int[] edge : edges){  int u = edge[0];  int v = edge[1];  int wt = edge[2];  if(dist[u]!=Integer.MAX\_VALUE && dist[u] + wt < dist[v]){  dist[v] = dist[u] + wt;  }  }  }  // negative wt cycle  for(int[] edge : edges){  int u = edge[0];  int v = edge[1];  int wt = edge[2];  if(dist[u]!=Integer.MAX\_VALUE && dist[u] + wt < dist[v]){  System.out.println("negative Wt cycle found");  System.exit(0);  }  }  return dist;  }  public static void main(String[] args){  Scanner scn = new Scanner(System.in);  int n = scn.nextInt(),m = scn.nextInt();  //edge list  int[][] edges = new int[m][3]; // (a,b,3) (c,d,5) ....  for(int i=0;i<m;i++){  int u = scn.nextInt(),v = scn.nextInt(),wt = scn.nextInt();  edges[i] = new int[]{u, v, wt};  }  //bellman algorithm  int[] distances = bellman\_ford(n,1,edges);  for(int i=1;i<=n;i++){  System.out.println("Node "+i+" is at dist "+distances[i]);  }  }  } |
| Ancestor printing:  public class AncestorPrinting {  static final int N = (int) (1e5 + 1);  static ArrayList<Integer>[] gr = new ArrayList[N];  static {  for (int i = 0; i < N; i++) {  gr[i] = new ArrayList<>();  }  }  static int[] Par = new int[N];  static void dfs(int cur, int par) {  Par[cur] = par;  for (int x : gr[cur]) {  if (x != par) {  // x is child node  dfs(x, cur);  }  }  }  public static void main(String[] args) {  Scanner scn = new Scanner(System.in);  int n = scn.nextInt();  for (int i = 0; i < n - 1; i++) {  int x = scn.nextInt(), y = scn.nextInt();  gr[x].add(y);  gr[y].add(x);  }  dfs(1, 0);  int x = 6;  // print all ancestors of 5  while (x > 0) {  System.out.println(x);  x = Par[x];  }  }  } | Strongly connected component:  public class KosarajuAlgorithm {  static final int N = (int) (1e5 + 1);  static ArrayList<Integer>[] gr = new ArrayList[N], grr = new ArrayList[N], components = new ArrayList[N];  static {  for (int i = 0; i < N; i++) {  gr[i] = new ArrayList<>();  grr[i] = new ArrayList<>();  components[i] = new ArrayList<>();  }  }  static boolean[] vis = new boolean[N];  static int[] col = new int[N];  static ArrayList<Integer> order = new ArrayList<>();  // topological sorting  static void dfs1(int cur) {  vis[cur] = true;  for (int x : gr[cur]) {  if (!vis[x]) {  dfs1(x);  }  }  order.add(cur);  }  static void dfs2(int cur, int comp) {  vis[cur] = true;  col[cur] = comp;  components[comp].add(cur);  for (int x : grr[cur]) {  if (!vis[x]) {  dfs2(x, comp);  }  }  }  static void addedge(int x, int y) {  gr[x].add(y);  grr[y].add(x);  }  public static void main(String[] args) {  Scanner scn = new Scanner(System.in);  int n = 4, m = 5;  // for (int i = 0; i < m; i++) {  // int x = scn.nextInt(), y = scn.nextInt();  // gr[x].add(y);  // grr[y].add(x);  // }  addedge(1,2);  addedge(2,3);  addedge(3,1);  addedge(3,4);  addedge(4,3);  for (int i = 1; i <= n; i++) {  if (!vis[i]) {  dfs1(i);  }  }  Collections.reverse(order);  Arrays.fill(vis, false);  int comp = 1;  for (int x : order) {  if (!vis[x]) {  dfs2(x, comp++);  }  }  for (int i = 1; i <= n; i++) {  System.out.println(i + " " + col[i]);  }  System.out.println("Total strongly components are -> " + (comp - 1));  // complexity O(n+m)  }  } |
| Travelling salesman:  public class TravellingSalesMan {  static int tsp(int[][] dist, int setOfCities, int city, int n, int[][] dp) {  //base case  if (setOfCities == (1 << n) - 1) {  //return the cost from the city to the original  return dist[city][0];  }  if (dp[setOfCities][city] != -1) {  return dp[setOfCities][city];  }  //otherwise try all possible options  int ans = Integer.MAX\_VALUE;  for (int choice = 0; choice < n; choice++) {  //need to check if city is visited or not  if ((setOfCities & (1 << choice)) == 0) {  int subProb = dist[city][choice] + tsp(dist, setOfCities | (1 << choice), choice, n, dp);  ans = Math.min(ans, subProb);  }  }  dp[setOfCities][city] = ans;  return ans;  }  public static void main(String[] args) {  int[][] dist = {  {0, 20, 42, 25},  {20, 0, 30, 34},  {42, 30, 0, 10},  {25, 34, 10, 0}  };  int n = 4;  int[][] dp = new int[1 << n][n];  for (int[] v : dp) {  Arrays.fill(v, -1);  }  System.out.println(tsp(dist, 1, 0, n, dp));  }  } | Kruskal:  public class Kruskal {  static class DSU{  int[] parent, rank;  public DSU(int n){  parent = new int[n];  rank = new int[n];  //parent -1, rank = 1  for(int i=0;i<n;i++){  parent[i] = -1;  rank[i] = 1;  }  }  int find(int i){  //base case  if(parent[i]==-1){  return i;  }  //otherwise  return parent[i] = find(parent[i]);  }  void unite(int x,int y){  int s1 = find(x);  int s2 = find(y);  if(s1!=s2){  //union by rank  if(rank[s1]<rank[s2]){  parent[s1] = s2;  rank[s2] += rank[s1];  }  else{  parent[s2] = s1;  rank[s1] += rank[s2];  }  }  }  }  static class Graph{  ArrayList<ArrayList<Integer>> edgeList;  int V;  public Graph(int V) {  this.V = V;  edgeList = new ArrayList<>();  }  void addEdge(int x,int y,int w){  ArrayList<Integer> list = new ArrayList<>();  list.add(w);  list.add(x);  list.add(y);  edgeList.add(list);  }  int kruskal\_mst(){  //Main Logic = Easy!!!  //1. Sort all the edges based upon weight  edgeList.sort(new Comparator<ArrayList<Integer>>() {  @Override  public int compare(ArrayList<Integer> o1, ArrayList<Integer> o2) {  return o1.get(0) - o2.get(0);  }  });  //Init a DSU  DSU s = new DSU(V);  int ans = 0;  for(List<Integer> edge : edgeList){  int w = edge.get(0);  int x = edge.get(1);  int y = edge.get(2);  //take that edge in MST if it doesnt form a cycle  if(s.find(x)!=s.find(y)){  s.unite(x,y);  ans += w;  }  }  return ans;  }  }  public static void main(String[] args){  Scanner scn = new Scanner(System.in);  int n = scn.nextInt(),m = scn.nextInt();  Graph g = new Graph(n);  for(int i=0;i<m;i++){  int x = scn.nextInt(),y= scn.nextInt(),w= scn.nextInt();  g.addEdge(x-1,y-1,w);  }  System.out.println(g.kruskal\_mst());  }  } |

**Trees:**

|  |  |
| --- | --- |
| Postorder:  private void printPostorder(Node1 node) {  if (node == null) {  return;  }  printPostorder(node.left);  printPostorder(node.right);  System.out.print(node.key + " ");  } |  |
| Inorder:  private void printPostorder(Node1 node) {  if (node == null) {  return;  }  printPostorder(node.left);  System.out.print(node.key + " ");  printPostorder(node.right);  } | Iterative inorder:  static void inorderIter(Node1 root) {  if (root == null) {  return;  }  Stack<Node1> s = new Stack<Node1>();  Node1 curr = root;  while (curr != null & !s.isEmpty()) {  while (curr != null) {  s.push(curr);  curr = curr.left;  }  curr = s.pop();  System.out.print(" " + curr.key);  curr = curr.right;  }  } |
| Preorder:  private void printPostorder(Node1 node) {  if (node == null) {  return;  }  System.out.print(node.key + " ");  printPostorder(node.left);  printPostorder(node.right);  } | Size:  public int size(Node1 root) {  if (root == null) {  return 0;  } else {  return 1 + size(root.left) + size(root.right);  }  } |
| Max:  public int max(Node1 root) {  if (root == null) {  return -1;  } else {  return Math.max(root.key, Math.max(max(root.left), max(root.right)));  }  } | Height:  public int height(Node1 root) {  if (root == null) {  return 0;  } else {  return (Math.max(height(root.left), height(root.right))) + 1;  }  } |
| Print at distance K:  public void printKdist(Node1 root, int k) {  if (root == null) {  return;  }  if (k == 0) {  System.out.print(root.key + " ");  } else {  printKdist(root.left, k - 1);  printKdist(root.right, k - 1);  }  } | Diameter:  int diameter(Node root) {  if (root == null) {  return 0;  }  int d1 = 1 + height(root.left) + height(root.right);  int d2 = diameter(root.left);  int d3 = diameter(root.right);  return Math.max(d1, Math.max(d2, d3));  } |
| Find a node:  static Node1 find(Node1 root, int x) {  if (root == null) {  return null;  }  if (root.key == x) {  return root;  }  Node1 left = find(root.left, x);  if (left != null) {  return left;  }  Node1 right = find(root.right, x);  if (right != null) {  return right;  }  return null;  } | Pair sum:  static void pairSum(Node1 curr, int sum, Node1 root) {  if (curr == null) {  return;  }  int s = sum - curr.key;  Node1 f = null;  if (curr.key < s) {  f = find(root, s);  }  if (f != null) {  System.out.println(curr.key + " " + f.key);  }  pairSum(curr.left, sum, root);  pairSum(curr.right, sum, root);  } |
| Maximum width:  int maxWidth(Node1 root) {  Queue<Node1> q = new LinkedList<Node1>();  q.add(root);  int res = 0;  while (!q.isEmpty()) {  int size = q.size();  res = Math.max(res, size);  for (int i = 0; i < size; i++) {  Node1 curr = q.poll();  if (curr.left != null) {  q.add(curr.left);  }  if (curr.right != null) {  q.add(curr.right);  }  }  }  return res;  } | Is balanced:  int isBalanced(Node1 root) {  if (root == null) {  return 0;  }  int lh = isBalanced(root.left);  if (lh == -1) {  return -1;  }  int rh = isBalanced(root.right);  if (rh == -1) {  return -1;  }  if (Math.abs(lh - rh) > 1) {  return -1;  } else {  return (Math.max(lh, rh) + 1);  }  } |
| Left view:  public void leftView(Node1 root) {  Queue<Node1> q = new LinkedList<Node1>();  q.add(root);  while (!q.isEmpty()) {  int cnt = q.size();  for (int i = 0; i < cnt; i++) {  Node1 curr = q.poll();  if (i == 0) {  System.out.print(curr.key + " ");  }  if (curr.left != null) {  q.add(curr.left);  }  if (curr.right != null) {  q.add(curr.right);  }  }  }  } | Level order line by line:  public void levelOrderLinebyLine(Node1 root) {  Queue<Node1> q = new LinkedList<Node1>();  q.add(root);  q.add(null);  while (q.size() > 1) {  Node1 curr = q.poll();  if (curr == null) {  System.out.println();  q.add(null);  continue;  }  System.out.print(curr.key + " ");  if (curr.left != null) {  q.add(curr.left);  }  if (curr.right != null) {  q.add(curr.right);  }  }  } |
| Print spiral:  public void printSpiral(Node1 root) {  if (root == null) {  return;  }  Queue<Node1> q = new LinkedList<Node1>();  Stack<Integer> s = new Stack<Integer>();  boolean reverse = false;  q.add(root);  while (q.isEmpty() == false) {  int cnt = q.size();  for (int i = 0; i < cnt; i++) {  Node1 curr = q.poll();  if (reverse) {  s.push(curr.key);  } else {  System.out.print(curr.key + " ");  }  if (curr.left != null) {  q.add(curr.left);  }  if (curr.right != null) {  q.add(curr.right);  }  if (reverse) {  while (s.isEmpty() == false) {  System.out.print(s.pop() + " ");  }  }  reverse = !reverse;  //System.out.println();  }  }  } | Is children sum:  boolean isChildrenSum(Node1 root) {  if (root == null) {  return true;  }  if (root.left == null && root.right == null) {  return true;  }  int sum = 0;  if (root.left != null) {  sum = sum + root.left.key;  }  if (root.right != null) {  sum = sum + root.right.key;  }  return (root.key == sum && isChildrenSum(root.left) && isChildrenSum(root.right));  } |
| Binary tree to Linked List:  static Node BinTree2LL(Node root) {  if (root == null) {  return root;  }  Node head = BinTree2LL(root.left);  if (prev == null) {  head = root;  } else {  root.left = prev;  prev.right = root;  }  prev = root;  BinTree2LL(root.right);  return head;  } | LCA:  static Node LCA(Node root, int n1, int n2) {  if (root == null) {  return null;  }  if (root.key == n1 || root.key == n2) {  return root;  }  Node lca1 = LCA(root.left, n1, n2);  Node lca2 = LCA(root.right, n1, n2);  if (lca1 != null && lca2 != null) {  return root;  }  if (lca1 != null) {  return lca1;  } else {  return lca2;  }  } |
| Left clone:  private static Node leftclone(Node root) {  if (root == null) {  return null;  }  Node lcr = leftclone(root.left);  Node rcr = leftclone(root.right);  Node newNode = new Node(root.key, lcr, null);  root.left = newNode;  root.right = rcr;  return root;  } | Right clone:  private static Node rightclone(Node root) {  if (root == null) {  return null;  }  Node lcr = leftclone(root.left);  Node rcr = leftclone(root.right);  Node newNode = new Node(root.key, null, rcr);  root.left = lcr;  root.right = newNode;  return root;  } |
| Level Order line by line LeftView:  static void levelOrderlinebylineLeftView(Node root,int k) {  Queue<Node> q = new ArrayDeque<>();  q.add(root);  while (!q.isEmpty()) {  int size = q.size();  for (int i = 0; i < size; i++) {  Node f = q.poll();  if(i==0){  System.out.print(f.key+" ");  }    if(f.left!=null){  q.add(f.left);  }  if(f.right!=null){  q.add(f.right);  }  }  System.out.println("");  }  } | Level Order line by line RightView:  static void levelOrderlinebylineRightView(Node root,int k) {  Queue<Node> q = new ArrayDeque<>();  q.add(root);  while (!q.isEmpty()) {  int size = q.size();  for (int i = 0; i < size; i++) {  Node f = q.poll();  if(i==size-1){  System.out.print(f.key+" ");  }    if(f.left!=null){  q.add(f.left);  }  if(f.right!=null){  q.add(f.right);  }  }  System.out.println("");  }  } |
| Node to root path:  static boolean path(Node root, int data, ArrayList<Integer> path) {  if (root == null) {  return false;  }  if (root.key == data) {  path.add(root.key);  return true;  }  boolean leftf = path(root.left, data, path);  if (leftf == true) {  path.add(root.key);  return true;  }  boolean rightf = path(root.right, data, path);  if (rightf == true) {  path.add(root.key);  return true;  }  return false;  } | Remove leaf nodes:  static Node sol(Node node) {  if (node == null) {  return null;  }  if (node.left == null && node.right == null) {  return null;  }  node.left = sol(node.left);  node.right = sol(node.right);  return node;  } |
| Single child node:  static void sol(Node root, Node parent) {  if (root == null) {  return;  }  if (parent!=null && parent.left == root && parent.right == null) {  System.out.println("" + root.key);  } else if (parent!=null && parent.right == root && parent.left == null) {  System.out.println("" + root.key);  }  sol(root.left, root);  sol(root.right, root);  } |  |

BST:

|  |  |
| --- | --- |
| Insert node recursive:  Node insertRec(Node root, int key) {  if (root == null) {  root = new Node(key);  return root;  }  if (key < root.key) {  root.left = insertRec(root.left, key);  root.lcount++;  }  if (key > root.key) {  root.right = insertRec(root.right, key);  }  return root;  } | Search Recursive:  static boolean searchRec(Node root, int x) {  if (root == null) {  return false;  }  if (x == root.key) {  return true;  }  if (x < root.key) {  return searchRec(root.left, x);  }  if (x > root.key) {  return searchRec(root.right, x);  }  return false;  } |
| Search Iterative:  static boolean searchI(Node root, int x) {  Node curr = root;  while (curr != null) {  if (x == curr.key) {  return true;  }  if (x < curr.key) {  curr = curr.left;  }  if (x > curr.key) {  curr = curr.right;  }  }  return false;  } | Floor:  static Node floor(Node root, int x) {  Node res = null;  while (root != null) {  if (x == root.key) {  return root;  } else if (x < root.key) {  root = root.left;  } else {  res = root;  root = root.right;  }  }  return res;  } |
| Ceiling:  static Node ceiling(Node root, int x) {  Node res = null;  while (root != null) {  if (x == root.key) {  return root;  } else if (x < root.key) {  res = root;  root = root.left;  } else {  root = root.right;  }  }  return res;  } | Is BST:  static boolean isBST(Node root, int min, int max) {  if (root == null) {  return true;  }  return (root.key > Integer.MIN\_VALUE && root.key < Integer.MAX\_VALUE  && isBST(root.left, Integer.MIN\_VALUE, root.key)  && isBST(root.right, root.key, Integer.MAX\_VALUE));  } |
| Is pair sum:  public static boolean isPairSum(Node root, int sum, HashSet<Integer> s)  {  if(root==null)return false;  if(isPairSum(root.left,sum,s)==true){  return true;  }  if(s.contains(sum-root.key)){  return true;  }else{  s.add(root.key);  }  return isPairSum(root.right,sum,s);  } | Kth smallest:  public static Node kthSmallest(Node root, int k) {  if (root == null) {  return null;  }  int count = root.lcount + 1;  if (count == k) {  return root;  }  if (count > k) {  return kthSmallest(root.left, k);  }  return kthSmallest(root.right, k - count);  } |
| Get successor:  static Node getSucc(Node root) {  Node curr = root.right;  while (curr != null && curr.left != null) {  curr = curr.left;  }  return curr;  } | Delete:  static Node delete(Node root, int x) {  if (root == null) {  return null;  }  if (root.key > x) {  root.left = delete(root.left, x);  } else if (root.key < x) {  root.right = delete(root.right, x);  } else {  if (root.left == null) {  return root.right;  } else if (root.right == null) {  return root.left;  } else {  Node succ = getSucc(root);  root.key = succ.key;  root.right = delete(root.right, succ.key);  }  }  return root;  } |

**Linked List:**

|  |  |
| --- | --- |
| Print list:  void printList(Node head) {  Node temp = head;  while (temp != null) {  System.out.print(temp.data + " ");  temp = temp.next;  }  System.out.println();  } | Push:  void push(int new\_data) {  Node new\_node = new Node(new\_data);  new\_node.next = head;  head = new\_node;  } |
| Insert at beginning:  void insertAtBeg(int x) {  Node1 temp = new Node1(x);  temp.next = head;  head = temp;  } | Insert at end:  void insertEnd(int x) {  Node1 temp = new Node1(x);  if (head == null) {  head = temp;  return;  }  Node1 curr = head;  while (curr.next != null) {  curr = curr.next;  }  curr.next = temp;  } |
| Middle node:  void middleNode(){  Node1 fast=head,slow=head;  if(head==null){  return;  }  while(fast!=null && fast.next!=null){  fast=fast.next.next;  slow=slow.next;  }  System.out.println("\n"+slow.data + " is middle node");  } | Print nth from end:  void printNthfromEnd(int n){  int len=0;  for(Node1 curr=head;curr!=null;curr=curr.next){  len++;  }  System.out.println(len);  if(len<n)return;  Node1 curr=head;  for(int i=0;i<len-n;i++){  curr=curr.next;  }  System.out.println("\n"+curr.data + " is "+n+"th node");  } |
| Detect loop:  boolean detectLoop(){  Node1 fast=head,slow=head;  while(fast!=null && fast.next!=null){  slow=slow.next;  fast=fast.next.next;  if(slow==fast){  return true;  }  }return false;  } | Detect and remove loop:  void detectandRemoveLoop(){  Node1 fast=head,slow=head;  while(fast!=null && fast.next!=null){  slow=slow.next;  fast=fast.next.next;  if(slow==fast){  break;  }  }  if(slow!=fast){  return;  }  slow=head;  while(slow.next!=fast.next){  slow=slow.next;  fast=fast.next;  }  fast.next=null;  } |
| No of nodes:  int getCount(Node node) {  Node current = node;  int count = 0;  while (current != null) {  count++;  current = current.next;  }  return count;  } | Get node:  int getNode() {  int c1 = getCount(head1);  int c2 = getCount(head2);  int d;  if (c1 > c2) {  d = c1 - c2;  return getIntesectionNode(d, head1, head2);  } else {  d = c2 - c1;  return getIntesectionNode(d, head2, head1);  }  } |
| Intersection of two LL:  int getIntesectionNode(int d, Node node1, Node node2) {  int i;  Node current1 = node1;  Node current2 = node2;  for (i = 0; i < d; i++) {  if (current1 == null) {  return -1;  }  current1 = current1.next;  }  while (current1 != null && current2 != null) {  if (current1.data == current2.data) {  return current1.data;  }  current1 = current1.next;  current2 = current2.next;  }  return -1;  } | Pairwise swap:  Node pairWiseSwap(Node node)  {  if (node == null || node.next == null) {  return node;  }  Node remaing = node.next.next;  Node newhead = node.next;  node.next.next = node;  node.next = pairWiseSwap(remaing);  return newhead;  } |
| Recursive reverse:  static Node reverse(Node head) {  if (head == null || head.next == null) {  return head;  }  Node smallAns = reverse(head.next);  head.next.next = head;  head.next = null;  return smallAns;  } | Reverse:  Node reverse(Node head) {  Node prev = null;  Node curr = head;  Node next;  while (curr != null) {  next = curr.next;  curr.next = prev;  prev = curr;  curr = next;  }  return prev;  } |
| Is palindrome:  static boolean isPalindrome(Node head) {  Node slow = head;  boolean ispalin = true;  Stack<Integer> stack = new Stack<Integer>();  while (slow != null) {  stack.push(slow.data);  slow = slow.ptr;  }  while (head != null) {  int i = stack.pop();  if (head.data == i) {  ispalin = true;  } else {  ispalin = false;  break;  }  head = head.ptr;  }  return ispalin;  } | Merge LL:  static Node sortedMerge(Node headA, Node headB) {  Node dummyNode = new Node(0);  Node tail = dummyNode;  while (true) {  if (headA == null) {  tail.next = headB;  break;  }  if (headB == null) {  tail.next = headA;  break;  }  if (headA.data <= headB.data) {  tail.next = headA;  headA = headA.next;  } else {  tail.next = headB;  headB = headB.next;  }  tail = tail.next;  }  return dummyNode.next;  } |

Generic Tree:

|  |  |
| --- | --- |
| private static void display(Node root) {  String str = root.data + "->";  for (Node t : root.children) {  str = str + t.data + ",";  }  str = str + ".";  System.out.println("" + str);  for (Node t : root.children) {  display(t);;  }  } | private static int size(Node root) {  int s = 0;  for (Node t : root.children) {  int cs = size(t);  s = s + cs;  }  s = s + 1;  return s;  } |
| private static int max(Node root) {  //int max=Integer.MIN\_VALUE;  int m = root.data;  for (Node t : root.children) {  int cm = max(t);  if (cm > m) {  m = cm;  }  }  return m;  } | private static int height(Node root) {  int ht = -1;  for (Node t : root.children) {  int chld = height(t);  ht = Math.max(ht, chld);  }  ht = ht + 1;  return ht;  } |
| private static void reverse(Node root) {  for(Node t:root.children){  reverse(t);  }  Collections.reverse(root.children);  } | Level order line by line:  private static void levelOrderlinebyline(Node root) {  Queue<Node> q = new LinkedList<>();  Queue<Node> cq = new LinkedList<>();  q.add(root);  while (q.size() > 0) {  Node front = q.poll();  System.out.print(front.data + " ");  for (Node t : front.children) {  cq.add(t);  }  if (q.size() == 0) {  q = cq;  cq = new LinkedList<>();  System.out.println("");  }  }  } |
| Level order zig jag:  private static void levelOrderZigZag(Node root) {  Stack<Node> ms = new Stack<>();  Stack<Node> cs = new Stack<>();  ms.push(root);  int level = 1;  while (ms.size() > 0) {  root = ms.pop();  System.out.print(root.data + " ");  if (level % 2 == 1) {  for (int i = 0; i < root.children.size(); i++) {  Node t = root.children.get(i);  cs.push(t);  }  } else {  for (int i = root.children.size() - 1; i >= 0; i--) {  Node t = root.children.get(i);  cs.push(t);  }  }  if (ms.size() == 0) {  ms = cs;  cs = new Stack<>();  level++;  System.out.println();  }  }  } |  |

Recursion:

|  |  |
| --- | --- |
| Coinchange:  private static void coinChange(int i, int[] coins, int amtsf, int tamt, String asf) {  if (i == coins.length) {  if (amtsf == tamt) {  System.out.println(asf + ".");  }  return;  }  coinChange(i + 1, coins, amtsf + coins[i], tamt, asf + coins[i] + "-");  coinChange(i + 1, coins, amtsf + 0, tamt, asf);  } | Display array:  private static void display(int[] arr, int idx) {  if(idx==arr.length){  return;  }  System.out.print(" "+arr[idx]);  display(arr, idx+1);  }    private static void displayRev(int[] arr, int idx) {  if(idx==arr.length){  return;  }  display(arr, idx+1);  System.out.print(" "+arr[idx]);    } |
| Remove X:  public static String removeX(String input, char a) {  if (input.length() == 0) {  return input;  }  String small = removeX(input.substring(1), a);  if (input.charAt(0) == a) {  return small;  } else {  return input.charAt(0) + small;  }  } | Remove consecutiv duplicates:  public static String removeConsecutiveDuplicates(String s) {  // Write your code here  if (s.length() == 1) {  return s;  }  String small = removeConsecutiveDuplicates(s.substring(1));  if (s.charAt(0) == small.charAt(0)) {  return small;  } else {  return s.charAt(0) + small;  }  } |
| Tiling:  private static int tiling(int n) {  if(n<=3){  return 1;  }else{  return tiling(n-1)+tiling(n-4);  }  } | Is array sorted:  private static boolean sorted(int[] arr, int i, int n) {  if (i == n - 1) {  return true;  }  if (arr[i] < arr[i + 1] && sorted(arr, i + 1, n)) {  return true;  }  return false;  } |
| Sum of digits:  static int sum = 0;    public static int sumOfDigits(int input) {  // Write your code here  if (input > 0) {  int lm = input % 10;  sum = sum + lm;  sumOfDigits(input / 10);  }  return sum;  } | Is string plindrome:  public static boolean isStringPalindrome(String input, int s, int e) {  if (s == e) {  return true;  }  if (input.charAt(s) != input.charAt(e)) {  return false;  }  if (s < e + 1) {  return isStringPalindrome(input, s + 1, e - 1);  }  return true;  } |
| Fact:  private static int fact(int i) {  if(i==1|| i==0){  return 1;  }  return i\*fact(i-1);  } | First,last and all index of array:  private static int first(int[] arr, int idx, int x) {  if (idx == arr.length) {  return -1;  }  if (arr[idx] == x) {  return idx;  } else {  int fiisa = first(arr, idx + 1, x);  return fiisa;  }  }  private static int last(int[] arr, int idx, int x) {  if (idx == arr.length) {  return -1;  }  int liisa = last(arr, idx + 1, x);  if (liisa == -1) {  if (arr[idx] == x) {  return idx;  } else {  return -1;  }  } else {  return liisa;  }  }  private static int[] allIndices(int[] arr, int x, int idx, int fsf) {  if(idx==arr.length){  return new int[fsf];  }  if(arr[idx]==x){  int[] iarr=allIndices(arr, x, idx + 1, fsf + 1);  iarr[fsf]=idx;  return iarr;  }else{  int[] iarr=allIndices(arr, x, idx + 1, fsf );  return iarr;  }    } |
| Max in array:  private static int max(int[] arr, int idx) {  if (idx == arr.length) {  return 0;  }  int p = max(arr, idx + 1);  if (p > arr[idx]) {  return p;  } else {  return arr[idx];  }  } | Power:  private static int power(int x, int n) {  if (n == 0) {  return 1;  }  int x1 = power(x, n - 1);  int xx = x \* x1;  return xx;  } |
| Power fast:  private static int power(int x, int n) {  if (n == 0) {  return 1;  }  int xpnb2 = power(x, n / 2);  int xn = xpnb2 \* xpnb2;  if (n % 2 == 1) {  xn = xn \* x;  }  return xn;  } | Print subsequence:  private static void printSS(String ques, String ans) {  if(ques.length()==0){  System.out.println(ans);  return;  }  char ch=ques.charAt(0);  String roq=ques.substring(1);    printSS(roq, ans+ch);  printSS(roq,ans+"");    } |
| Return subsequences:  private static ArrayList<String> gss(String str) {  if (str.length() == 0) {  ArrayList<String> bres = new ArrayList<>();  bres.add("");  return bres;  }  char ch = str.charAt(0);  String ros = str.substring(1);  ArrayList<String> rres = gss(ros);  ArrayList<String> mres = new ArrayList<>();  for (String p : rres) {  mres.add("" + p);  mres.add(ch + p);  }  return mres;  } | Subset sum:  private static void printSubsets(int[] arr, int idx, String set, int sos, int tar) {  if(idx==arr.length){  if(sos==tar) {  System.out.println(set+".");  }  return;  }  printSubsets(arr,idx+1,set+arr[idx]+",",sos+arr[idx],tar);  printSubsets(arr, idx+1, set, sos, tar);  } |
| Print encodings:  private static void printEncodings(String ques, String asf) {  if(ques.length()==0){  System.out.println(""+asf);  return;  }else if(ques.length()==1){  char ch=ques.charAt(0);  if(ch=='0'){  return;  }else{  int chv=ch-'0';  char code=(char)('a'+chv-1);  asf=asf+code;  System.out.println(asf);  }  }else {  char ch=ques.charAt(0);  String roq=ques.substring(1);    if(ch=='0'){  return;  }else{  int chv=ch-'0';  char code=(char)('a'+chv-1);  printEncodings(roq,asf+code);  }  String ch12=ques.substring(0,2);  String roq12=ques.substring(2);    int ch12v=Integer.parseInt(ch12);  if(ch12v<=26){  char code=(char)('a'+ch12v-1);  printEncodings(roq12,asf+code);  }    }  } | Get stair paths:  private static ArrayList<String> getStairPaths(int n) {  if (n == 0) {  ArrayList<String> bres = new ArrayList<String>();  bres.add("");  return bres;  } else if (n < 0) {  ArrayList<String> bres = new ArrayList<String>();  return bres;  }  ArrayList<String> paths1 = getStairPaths(n - 1);  ArrayList<String> paths2 = getStairPaths(n - 2);  ArrayList<String> paths3 = getStairPaths(n - 3);  ArrayList<String> paths = new ArrayList<String>();  for (String path : paths1) {  paths.add(1 + path);  }  for (String path : paths2) {  paths.add(2 + path);  }  for (String path : paths3) {  paths.add(3 + path);  }  return paths;  } |
| Print permutations:  private static void printPermut(String ques, String asf) {  if (ques.length() == 0) {  System.out.println("" + asf);  return;  }  for (int i = 0; i < ques.length(); i++) {  char ch = ques.charAt(i);  String qlpart = ques.substring(0, i);  String qrpart = ques.substring(i + 1);  String roq = qlpart + qrpart;  printPermut(roq, asf + ch);  }  } | Permutations using backtracking:  static void permutations(int[] boxes, int ci, int ti) {  if (ci > ti) {  for (int i = 0; i < boxes.length; i++) {  System.out.print(" " + boxes[i]);  }  System.out.println("");  return;  }  for (int i = 0; i < boxes.length; i++) {  if (boxes[i] == 0) {  boxes[i] = ci;  permutations(boxes, ci + 1, ti);  boxes[i] = 0;  }  }  } |
| Coin change combination (once per coin):  private static void coins(int i, int[] arr, int colsofar, int tar, String asf) {  if(i==arr.length){  if(colsofar==tar){  System.out.println(asf+".");  }  return;  }    coins(i+1,arr,colsofar+arr[i],tar,asf+arr[i]+"-");  coins(i+1,arr,colsofar+0,tar,asf);    } | Coin change combination (infinite per coin):  private static void coins(int i, int[] arr, int colsofar, int tar, String asf) {  if (i == arr.length) {  if (colsofar == tar) {  System.out.println(asf + ".");  }  return;  }  for (int j = tar / arr[i]; j >= 1; j--) {  String part = "";  for (int k = 0; k < j; k++) {  part += arr[i] + "-";  }  coins(i + 1, arr, colsofar + arr[i] \* j, tar, asf + part);  }  coins(i + 1, arr, colsofar + 0, tar, asf);  } |
| Combinations:  private static void combinations(int cb, int tb, int ssf, int ts, String asf) {  if (cb > tb) {  if (ssf == ts) {  System.out.println(asf);  }  return;  }  combinations(cb + 1, tb, ssf + 1, ts, asf + "i");  combinations(cb + 1, tb, ssf, ts, asf + "-");  } | Permutation:  static void permutations(int[] boxes, int ci, int ti) {  if (ci > ti) {  for (int i = 0; i < boxes.length; i++) {  System.out.print(boxes[i]);  }  System.out.println("");  return;  }  for (int i = 0; i < boxes.length; i++) {  if (boxes[i] == 0) {  boxes[i] = ci;  permutations(boxes, ci + 1, ti);  boxes[i] = 0;  }  }  } |
| Print permutations of string:  public static void printPermutations(String str, String asf) {    if(str.length()==0){  System.out.println(asf);  return;  }  for(int i=0;i<str.length();i++){  char ch=str.charAt(i);  String lpart=str.substring(0,i);  String rpart=str.substring(i+1);  String ros=lpart+rpart;  printPermutations(ros,asf+ch);  }  } | Queens permutation:  private static void queens(int qpsf, int tq, int[][] chess) {  if (qpsf == tq) {  for (int i = 0; i < chess.length; i++) {  for (int j = 0; j < chess[0].length; j++) {  if (chess[i][j] == 0) {  System.out.print("-\t");  } else {  System.out.print("q" + chess[i][j] + "\t");  }  }  System.out.println("");  }  System.out.println("");  return;  }  for (int i = 0; i < chess.length; i++) {  for (int j = 0; j < chess[0].length; j++) {  if (chess[i][j] == 0) {  chess[i][j] = qpsf + 1;  queens(qpsf + 1, tq, chess);  chess[i][j] = 0;  }  }  }  } |
| Queens permutation:  private static void queensPermut(int qpsf, int tq, int row, int col, String asf, boolean[] queens) {  if (row == tq) {  if (qpsf == tq) {  System.out.print(asf);  System.out.println("");  }  return;  }  int nr = 0, nc = 0;  String yasf = "", nasf = "";  String sep=" ";  if (col == tq - 1) {  nr = row + 1;  nc = 0;  sep="\n";  } else {  nr = row;  nc = col + 1;  sep="\t";  }    for (int i = 0; i < queens.length; i++) {  if(queens[i]==false){  queens[i]=true;  queensPermut(qpsf+1,tq,nr,nc,asf+"q"+(i+1)+sep,queens);  queens[i]=false;  }  }    queensPermut(qpsf+0,tq,nr,nc,asf+"-"+sep,queens);  } | Queen combination:  private static void queen(int qpsf, int tq, int row, int col, String asf) {  if (row == tq) {  if (qpsf == tq) {  System.out.println(asf);  }  return;  }  int nr = 0, nc = 0;  String yasf = "", nasf = "";  if (col == tq - 1) {  nr = row + 1;  nc = 0;  yasf = asf + "q\n";  nasf = asf + "-\n";  } else {  nr = row;  nc = col + 1;  yasf = asf + "q";  nasf = asf + "-";  }  queen(qpsf + 1, tq, nr, nc, yasf);  queen(qpsf + 0, tq, nr, nc, nasf);  } |
| Word break:  static void wordBreak(String str, String ans, HashSet<String> dict) {  if (str.length() == 0) {  System.out.println(ans);  return;  }  for (int i = 0; i < str.length(); i++) {  String left = str.substring(0, i + 1);  if (dict.contains(left)) {  String right = str.substring(i + 1);  wordBreak(right, ans + left + " ", dict);  }  }  } |  |

DP:

|  |  |
| --- | --- |
| Climb stairs:  public static void main(String[] args) {  int arr[] = {1, 5, 2, 3, 1};  int n = arr.length;  Integer dp[] = new Integer[n + 1];  dp[n] = 0;  for (int i = n - 1; i >= 0; i--) {  if (arr[i] > 0) {  int min = Integer.MAX\_VALUE;  for (int j = 1; j <= arr[i] & (i + j) < dp.length; j++) {  if (dp[i + j] != null) {  min = Math.min(min, dp[i + j]);  }  }  if (min != Integer.MAX\_VALUE) {  dp[i] = min + 1;  }  }  }    for (int i = 0; i < dp.length; i++) {  System.out.print(" " + dp[i]);  }  } | Coinchange combination:  public static void main(String[] args) {  int arr[] = {2, 3, 5};  int amt = 10;  int[] dp = new int[amt + 1];  dp[0] = 1;  for (int i = 0; i < arr.length; i++) {  for (int j = arr[i]; j < dp.length; j++) {  dp[j] = dp[j] + dp[j - arr[i]];  }  }  System.out.println("" + dp[amt]);  } |
| Coin change permutation:  public static void main(String[] args) {  int arr[] = {2, 3, 5,6};  int tar = 10;  int[] dp=new int[tar+1];  dp[0]=1;  for(int i=1;i<dp.length;i++){  for(int j=0;j<arr.length;j++){  if(arr[j]<=i){  dp[i]=dp[i]+dp[i-arr[j]];  }  }  }  System.out.println(""+dp[tar]);  } | Goldmine:  public static void main(String[] args) {  int grid[][] = {  {8, 2, 1, 6},  {6, 5, 5, 2},  {2, 1, 0, 3},  {7, 2, 2, 4}  };  int n = grid.length;  int m = grid[0].length;  int dp[][] = new int[n][m];  for (int j = grid[0].length - 1; j >= 0; j--) {  for (int i = grid.length - 1; i >= 0; i--) {  if (j == grid[0].length - 1) {  dp[i][j] = grid[i][j];  } else if (i == grid.length - 1) {  dp[i][j] = grid[i][j] + Math.max(dp[i][j + 1], dp[i - 1][j + 1]);  } else if (i == 0) {  dp[i][j] = grid[i][j] + Math.max(dp[i][j + 1], dp[i + 1][j + 1]);  } else {  dp[i][j] = grid[i][j] + Math.max(dp[i][j + 1], Math.max(dp[i - 1][j + 1], dp[i + 1][j + 1]));  }  }  }  int max = dp[0][0];  for (int i = 0; i < dp.length; i++) {  if (dp[i][0] > max) {  max = dp[i][0];  }  }  System.out.println("" + max); |
| Min cost path:  public static void main(String[] args) throws Exception {  Scanner s=new Scanner(System.in);  int n=s.nextInt();  int[][] arr=new int[n][3];  for (int i = 0; i < arr.length; i++) {  for (int j = 0; j < arr[0].length; j++) {  arr[i][j]=s.nextInt();  }  }  long[][] dp=new long[n][3];  dp[0][0]=arr[0][0];  dp[0][1]=arr[0][1];  dp[0][2]=arr[0][2];  for(int i=1;i<arr.length;i++){  dp[i][0]=arr[i][0]+Math.min(dp[i-1][1],dp[i-1][2]);  dp[i][1]=arr[i][1]+Math.min(dp[i-1][0],dp[i-1][2]);  dp[i][2]=arr[i][2]+Math.min(dp[i-1][0],dp[i-1][1]);  }  long ans=Math.min(dp[n-1][0], Math.min(dp[n-1][1], dp[n-1][2]));  System.out.println(ans);  } | Count binary strings with No 2 consecutive Zeroes:  public static void main(String[] args) {  int n=3;  int[] dp0=new int[n+1];  int[] dp1=new int[n+1];  dp0[1]=1;  dp1[1]=1;  for(int i=2;i<=n;i++){  dp1[i]=dp1[i-1]+dp0[i-1];  dp0[i]=dp1[i-1];  }  System.out.println(""+dp1[n]+dp0[n]);  } |
| Arrange buildings:  public static void main(String[] args) {  int n=5;  int ob=1;  int os=1;  for(int i=2;i<=n;i++){  int nb=os;  int ns=os+ob;  os=ns;  ob=nb;  }  int total=os+ob;  total=total\*total;  System.out.println(""+total);  } | Decode ways:  public static void main(String[] args) {  String str="21123";  int[] dp=new int[str.length()];  dp[0]=1;  for(int i=1;i<dp.length;i++){  if(str.charAt(i-1)=='0' && str.charAt(i)=='0'){  dp[i]=0;  }else if(str.charAt(i-1)=='0' && str.charAt(i)!='0'){  dp[i]=dp[i-1];  }else if(str.charAt(i-1)!='0' && str.charAt(i)=='0'){  if(str.charAt(i-1)=='1' || str.charAt(i-1)=='2'){  dp[i]=i>=2?dp[i-2]:1;  }else{  dp[i]=0;  }  }else{  if(Integer.parseInt(str.substring(i-1,i+1))<=26){  dp[i]=dp[i-1]+(i>=2?dp[i-2]:1);  }else{  dp[i]=dp[i-1];  }  }  }  System.out.println(dp[str.length()-1]);  } |
| Catalan:  public static void main(String[] args) {  int n = 11;  int dp[] = new int[n + 1];  dp[0] = 1;  dp[1] = 1;  for (int i = 2; i < dp.length; i++) {  for (int j = 0; j < i; j++) {  dp[i] = dp[j] + dp[i] \* dp[i - j - 1];  }  }  for (int i = 0; i < dp.length; i++) {  System.out.println("Catalan of" + i + " " + dp[i]);  }  System.out.println(""+Integer.MAX\_VALUE);  } | Count palindromic subsequences:  public static void main(String[] args) {  String str = "abccbc";  int[][] dp = new int[str.length()][str.length()];  for (int g = 0; g < str.length(); g++) {  for (int i = 0, j = g; j < dp[0].length; j++, i++) {  if (g == 0) {  dp[i][j]=1;  } else if (g == 1) {  dp[i][j]=str.charAt(i)==str.charAt(j)?3:2;  } else {  if(str.charAt(i)==str.charAt(j)){  dp[i][j]=dp[i][j-1]+dp[i+1][j]+1;  }else{  dp[i][j] = dp[i][j - 1] + dp[i + 1][j] - dp[i + 1][j - 1];  }  }  }  }  System.out.println(dp[0][str.length() - 1]);  } |
| Distinct subsequence:  public static void main(String[] args) {  String str="abcdcc";  int[] dp=new int[str.length()+1];  dp[0]=1;    HashMap<Character,Integer> lo=new HashMap<>();  for (int i = 1; i < dp.length; i++) {  dp[i]=dp[i-1]\*2;    char ch=str.charAt(i-1);  if(lo.containsKey(ch)){  int pos=lo.get(ch);  dp[i]=dp[i]-dp[pos-1];  }  lo.put(ch,i);  }    System.out.println(dp[str.length()]);  } | LIS:  public static void main(String[] args) {  int[] arr = {10, 22, 9, 33, 21, 50, 41, 60, 80, 1};  int n = arr.length;  int[] dp = new int[n];  dp[0] = 1;  for (int i = 1; i < dp.length; i++) {  int max = 0;  for (int j = 0; j < i; j++) {  if (arr[i] > arr[j]) {  if (dp[j] > max) {  max = dp[j];  }  }  }  dp[i] = max + 1;  }  int omax = 0;  for (int i = 0; i < dp.length; i++) {  if (dp[i] > omax) {  omax = dp[i];  }  }  System.out.print(" " + omax);  } |
| LCS:  public static void main(String[] args) {  String s1 = "abcd";  String s2 = "abbd";  int[][] dp = new int[s1.length() + 1][s2.length() + 1];  for (int i = dp.length - 2; i >= 0; i--) {  for (int j = dp[0].length - 2; j >= 0; j--) {  char c1 = s1.charAt(i);  char c2 = s2.charAt(j);  if (c1 == c2) {  dp[i][j] = 1 + dp[i + 1][j + 1];  } else {  dp[i][j] = Math.max(dp[i + 1][j], dp[i][j + 1]);  }  }  }  System.out.println(dp[0][0]);  } | Largest square submatrix:  public static void main(String[] args) {  int arr[][] = {{1, 1, 1, 1},  {1, 1, 1, 0},  {1, 1, 1, 0}  };  System.out.println(solution(arr));  }  private static int solution(int[][] arr) {  int dp[][] = new int[arr.length][arr[0].length];  int ans = Integer.MIN\_VALUE;  for (int i = dp.length - 1; i >= 0; i--) {  for (int j = dp[0].length - 1; j >= 0; j--) {  if (i == dp.length - 1 && j == dp[0].length - 1) {  dp[i][j] = arr[i][j];  } else if (i == dp.length - 1) {  dp[i][j] = arr[i][j];  } else if (j == dp[0].length - 1) {  dp[i][j] = arr[i][j];  } else {  if (arr[i][j] == 0) {  dp[i][j] = 0;  } else {  int min = Math.min(dp[i][j + 1], dp[i + 1][j]);  min = Math.min(min, dp[i + 1][j + 1]);  dp[i][j] = min + 1;  if (dp[i][j] > ans) {  ans = dp[i][j];  }  }  }  }  }  return ans;  } |
| Longest Common Substring:  public static void main(String[] args) {  String s1 = "baaabab";  String s2 = "baaab";  System.out.println(sol(s1,s2));  }  private static int sol(String s1, String s2) {  int[][] dp=new int[s1.length()+1][s2.length()+1];  int max=0;  for(int i=1;i<dp.length;i++){  for(int j=1;j<dp[0].length;j++){  char c1=s1.charAt(i-1);  char c2=s2.charAt(j-1);  if(c1!=c2){  dp[i][j]=0;  }else{  dp[i][j]=dp[i-1][j-1]+1;  }  if(dp[i][j]>max){  max=dp[i][j];  }  }  }  return max;  } | Longest palindromic subsequence:  public static void main(String[] args) {  String s = "abccbc";  int[][] dp = new int[s.length()][s.length()];  for (int g = 0; g < s.length(); g++) {  for (int i = 0, j = g; j < dp.length; i++, j++) {  if (g == 0) {  dp[i][j] = 1;  } else if (g == 1) {  dp[i][j] = s.charAt(i) == s.charAt(j) ? 2 : 1;  } else {  if (s.charAt(i) == s.charAt(j)) {  dp[i][j] = 2 + dp[i + 1][j - 1];  } else {  dp[i][j] = Math.max(dp[i][j - 1], dp[i + 1][j]);  }  }  }  }  System.out.println(dp[0][s.length() - 1]);  } |
| Longest Palindromic Substring:  public static void main(String[] args) {  String s = "abccbc";  int len = 0;  boolean[][] dp = new boolean[s.length()][s.length()];  for (int g = 0; g < s.length(); g++) {  for (int i = 0, j = g; j < dp.length; i++, j++) {  if (g == 0) {  dp[i][j] = true;  } else if (g == 1) {  if (s.charAt(i) == s.charAt(j)) {  dp[i][j] = true;  } else {  dp[i][j] = false;  }  } else {  if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1] == true) {  dp[i][j] = true;  } else {  dp[i][j] = false;  }  }  if (dp[i][j]) {  len = g + 1;  }  }  }  System.out.println("" + len);  } | Max sum increasing subseq:  public static void main(String[] args) {  int arr[] = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3};  int omax = Integer.MIN\_VALUE;  int dp[] = new int[arr.length];  for (int i = 0; i < dp.length; i++) {  Integer max = null;  for (int j = 0; j < i; j++) {  if (arr[j] <= arr[i]) {  if (max == null) {  max = dp[j];  } else if (dp[j] > max) {  max = dp[j];  }  }  }  if (max == null) {  dp[i] = arr[i];  } else {  dp[i] = max + arr[i];  }  if (dp[i] > omax) {  omax = dp[i];  }  }  System.out.println(  "" + omax);  } |
| Minimum Cost to Make Strings Identical:  public static void main(String[] args) {  String s1 = "cat";  String s2 = "cut";  int x = 1;  int y = 1;  System.out.println(sol(s1, s2, x, y));  }  private static int sol(String s1, String s2, int c1, int c2) {  int[][] dp = new int[s1.length() + 1][s2.length() + 1];  for (int i = dp.length - 1; i >= 0; i--) {  for (int j = dp[0].length - 1; j >= 0; j--) {  if (i == dp.length - 1 && j == dp[0].length - 1) {  dp[i][j] = 0;  } else if (i == dp.length - 1) {  dp[i][j] = 0;  } else if (j == dp[0].length - 1) {  dp[i][j] = 0;  } else {  if (s1.charAt(i) == s2.charAt(j)) {  dp[i][j] = 1 + dp[i + 1][j + 1];  } else {  dp[i][j] = Math.max(dp[i+1][j],dp[i][j+1]);  }  }  }  }    int lcs=dp[0][0];  int s1r=s1.length()-lcs;  int s2r=s2.length()-lcs;  int cost=s1r\*c1+s2r\*c2;  return cost;  } | Non Intersecting Chords Circle:  public static void main(String[] args) {  int n=6;  System.out.println(noofchord(n));  }  private static long noofchord(int n) {  long[] dp=new long[n+1];  dp[0]=1;  dp[1]=1;  for(int i=2;i<=n;i++){  int l=0;  int r=i-1;  while(l<=i-1){  dp[i]=dp[i]+dp[l]\*dp[r];  l++;  r--;  }  }  return dp[n];  } |
| Optimal strategy of a game:  public static void main(String[] args) {  int[] a = {20, 30, 2, 2, 2, 10};  optimal(a);  }  private static void optimal(int[] arr) {  int[][] dp = new int[arr.length][arr.length];  for (int g = 0; g < dp.length; g++) {  for (int i = 0, j = g; j < dp.length; i++, j++) {  if (g == 0) {  dp[i][j] = arr[i];  } else if (g == 1) {  dp[i][j] = Math.max(arr[i], arr[j]);  } else {  int val1 = arr[i] + Math.min(dp[i + 2][j], dp[i + 1][j - 1]);  int val2 = arr[i] + Math.min(dp[i + 1][j - 1], dp[i][j - 2]);  int val = Math.max(val1, val2);  dp[i][j] = val;  }  }  }  System.out.println("" + dp[0][arr.length - 1]);  } | Print all LIS:  public static class Pair {  int l, i, val;  String psf;  Pair(int l, int i, int val, String psf) {  this.l = l;  this.i = i;  this.val = val;  this.psf = psf;  }  }  public static void main(String[] args) {  int arr[] = {5, 6, 7, 8, 3};  solution(arr);  }  private static void solution(int[] arr) {  int[] dp = new int[arr.length];  int omax = 0;  int omi = 0;  for (int i = 0; i < dp.length; i++) {  int max = 0;  for (int j = 0; j < i; j++) {  if (arr[j] <= arr[i]) {  if (dp[j] > max) {  max = dp[j];  }  }  }  dp[i] = max + 1;  if (dp[i] > omax) {  omax = dp[i];  omi = i;  }  }  ArrayDeque<Pair> q = new ArrayDeque<>();  q.add(new Pair(omax, omi, arr[omi], arr[omi] + ""));  while (q.size() > 0) {  Pair rem = q.removeFirst();  if (rem.l == 1) {  System.out.println(rem.psf);  }  for (int j = 0; j < rem.i; j++) {  if (dp[j] == rem.l - 1 && arr[j] <= rem.val) {  q.add(new Pair(dp[j], j, arr[j], arr[j] + " -> " + rem.psf));  }  }  }  } |
| Print path maximum gold:  public static class Pair {  int j, i;  String psf;  Pair(String psf, int i, int j) {  this.j = j;  this.i = i;  this.psf = psf;  }  }  public static void main(String[] args) {  int arr[][] = {  {3, 2, 3, 1},  {2, 4, 6, 0},  {5, 0, 1, 3},  {9, 1, 5, 1}  };  int[][] dp = new int[arr.length][arr[0].length];  for (int j = arr[0].length - 1; j >= 0; j--) {  for (int i = 0; i < arr.length; i++) {  if (j == arr[0].length - 1) {  dp[i][j] = arr[i][j];  } else if (i == 0) {  dp[i][j] = arr[i][j] + Math.max(dp[i][j + 1], dp[i + 1][j + 1]);  } else if (i == arr.length - 1) {  dp[i][j] = arr[i][j] + Math.max(dp[i][j + 1], dp[i - 1][j + 1]);  } else {  dp[i][j] = arr[i][j] + Math.max(dp[i][j + 1], Math.max(dp[i - 1][j + 1], dp[i + 1][j + 1]));  }  }  }  int max = Integer.MIN\_VALUE;  for (int i = 0; i < dp.length; i++) {  if (dp[i][0] > max) {  max = dp[i][0];  }  }  System.out.println("" + max);  ArrayDeque<Pair> q = new ArrayDeque<>();  for (int i = 0; i < dp.length; i++) {  if (dp[i][0] > max) {  q.add(new Pair(i + "", i, 0));  }  }  while (q.size() > 0) {  Pair rem = q.removeFirst();  if (rem.j == arr[0].length - 1) {  System.out.println(rem.psf);  } else if (rem.i == 0) {  int g = Math.max(dp[rem.i][rem.j + 1], dp[rem.i + 1][rem.j + 1]);  if (g == dp[rem.i][rem.j + 1]) {  q.add(new Pair(rem.psf + "d2", rem.i, rem.j + 1));  }  if (g == dp[rem.i + 1][rem.j + 1]) {  q.add(new Pair(rem.psf + "d3", rem.i + 1, rem.j + 1));  }  } else if (rem.i == arr.length - 1) {  int g = Math.max(dp[rem.i][rem.j + 1], dp[rem.i - 1][rem.j + 1]);  if (g == dp[rem.i][rem.j + 1]) {  q.add(new Pair(rem.psf + "d2", rem.i, rem.j + 1));  }  if (g == dp[rem.i - 1][rem.j + 1]) {  q.add(new Pair(rem.psf + "d1", rem.i - 1, rem.j + 1));  }  } else {  int g = Math.max(dp[rem.i][rem.j + 1], Math.max(dp[rem.i - 1][rem.j + 1], dp[rem.i + 1][rem.j + 1]));  if (g == dp[rem.i - 1][rem.j + 1]) {  q.add(new Pair(rem.psf + "d1", rem.i - 1, rem.j + 1));  }  if (g == dp[rem.i][rem.j + 1]) {  q.add(new Pair(rem.psf + "d2", rem.i, rem.j + 1));  }  if (g == dp[rem.i + 1][rem.j + 1]) {  q.add(new Pair(rem.psf + "d3", rem.i + 1, rem.j + 1));  }  }  }  } | Burst baloons:  public static void main(String[] args) {  int[] arr={2,3,5};  System.out.println(sol(arr));  }  private static int sol(int[] arr) {  int[][] dp=new int[arr.length][arr.length];  for(int g=0;g<dp.length;g++){  for(int i=0,j=g;j<dp.length;i++,j++){  int max=Integer.MIN\_VALUE;  for(int k=i;k<=j;k++){  int left=k==i?0:dp[i][k-1];  int right=k==j?0: dp[k+1][j];  int val=(i==0?1:arr[i-1])\*arr[k]\*(j==arr.length-1?1:arr[j+1]);  int total=left+right+val;  if(total>max){  max=total;  }  }  dp[i][j]=max;  }  }  return dp[0][dp.length-1];  } |
| Combination of balanced parenthesis:  public static void main(String[] args) {  int n=4;  int[] dp=new int[n+1];  dp[0]=1;  dp[1]=1;  for (int i = 2; i <=n; i++) {  int inside=i-1;  int outside=0;  while(inside>=0){  dp[i]=dp[i]+dp[inside]\*dp[outside];  inside--;  outside++;  }  }  System.out.println(dp[n]);  bal(n);  }      static void bal(int n){  int[] dp=new int[n+1];    dp[0]=1;  dp[1]=1;    for(int i=2;i<=n;i++){  int inside=i-1;  int outside=0;    while(inside>=0){  dp[i]=dp[i]+dp[inside]\*dp[outside];    inside--;  outside++;  }  }  System.out.println(""+dp[n]);  } | Count distinct subsequences:  public static void main(String[] args) {  String str = "abcbac";  int[] dp = new int[str.length()+1];  dp[0] = 1;  HashMap<Character, Integer> hm = new HashMap<>();  for (int i = 1; i < dp.length; i++) {  dp[i] = 2 \* dp[i - 1];  char ch = str.charAt(i - 1);  if (hm.containsKey(ch)) {  int j = hm.get(ch);  dp[i] = dp[i] - dp[j - 1];  }  hm.put(ch, i);  }  System.out.println(dp[str.length()] - 1);  } |
| Count no of BST:  public static void main(String[] args) {  int n=4;  int[] dp=new int[n+1];  dp[0]=1;  dp[1]=1;    for (int i = 2; i <=n; i++) {  int l=0;  int r=i-1;  while(l<=i-1){  dp[i]+=dp[l]\*dp[r];  l++;  r--;  }  }    System.out.println(dp[n]);  } | Count palindromic substrings:  public static void main(String[] args) {  String s = "abccbc";  boolean[][] dp = new boolean[s.length()][s.length()];  int cnt = 0;  for (int g = 0; g < s.length(); g++) {  for (int i = 0, j = g; j < dp.length; i++, j++) {  if (g == 0) {  dp[i][j] = true;  } else if (g == 1) {  if (s.charAt(i) == s.charAt(j)) {  dp[i][j] = true;  } else {  dp[i][j] = false;  }  } else {  if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1] == true) {  dp[i][j] = true;  } else {  dp[i][j] = false;  }  }  if (dp[i][j]) {  cnt++;  }  }  }  System.out.println("" + cnt);  } |
| Count valleys and mountains:  public static void main(String[] args) {  int n=5;  int[] dp=new int[n+1];  dp[0]=1;  dp[1]=1;  for(int i=2;i<dp.length;i++){  int inside=i-1;  int outside=i-inside;  while(inside>=0){  dp[i]=dp[i]+dp[outside]\*dp[inside];  inside--;  outside++;  }  }  System.out.println(dp[n]);    } | Egg dropping:  public static void main(String[] args) {  int n = 4;  int k = 2;  System.out.println(eggdrop(n, k));  }  public static int eggdrop(int n, int k) {  int[][] dp = new int[n + 1][k + 1];  for (int i = 1; i <= n; i++) {  for (int j = 1; j <= k; j++) {  if (i == 1) {  dp[i][j] = j;  } else if (j == 1) {  dp[i][j] = 1;  } else {  int min = Integer.MAX\_VALUE;  for (int mj = j - 1, pj = 0; mj >= 0; mj--, pj++) {  int v1 = dp[i][mj];//egg survices  int v2 = dp[i - 1][pj];//egg breaks  int val = Math.max(v1, v2);  min = Math.min(val, min);  }  dp[i][j] = min + 1;  }  }  }  return dp[n][k];  } |
| Kadane max sum subarray:  public static void main(String[] args) {  int arr[] = {5, 6, 7, 4, 3, 6, 4};  System.out.println(sol(arr));  }  private static int sol(int[] arr) {  int csum = arr[0];  int osum = arr[0];  for (int i = 1; i < arr.length; i++) {  if (csum >= 0) {  csum += arr[i];  } else {  csum = arr[i];  }  if (csum > osum) {  osum = csum;  }  }  return osum;  } | Longest bitonic subsequences:  public static void main(String[] args) {  int arr[] = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3};  int n = arr.length;  int lis[] = new int[n];  for (int i = 0; i < arr.length; i++) {  int max = 0;  for (int j = 0; j < i; j++) {  if (arr[j] <= arr[i]) {  if (lis[j] > max) {  max = lis[j];  }  }  }  lis[i] = max + 1;  }  ///////////////  int lds[] = new int[n];  for (int i = arr.length - 1; i >= 0; i--) {  int max = 0;  for (int j = arr.length - 1; j > i; j--) {  if (arr[j] <= arr[i]) {  if (lds[j] > max) {  max = lds[j];  }  }  }  lds[i] = max + 1;  }  int omax = 0;  for (int i = 0; i < arr.length; i++) {  if (lis[i] + lds[i] - 1 > omax) {  omax = lis[i] + lds[i] - 1;  }  }  System.out.println("" + omax);  } |
| Arithmatic slices:  public static void main(String[] args) {  int[] arr = {2, 5, 9, 12, 15, 18, 22, 26, 30, 34, 36, 38, 40, 41};  System.out.println(solution(arr));  }  private static int solution(int[] arr) {  int[] dp = new int[arr.length];  int ans = 0;  for (int i = 2; i < arr.length; i++) {  if (arr[i] - arr[i - 1] == arr[i - 1] - arr[i - 2]) {  dp[i] = dp[i - 1] + 1;  ans = ans + dp[i];  }  }  return ans;  } | All path minimum jumps:  public static class Pair {  int i, s, j;  String psf;  Pair(int i, int s, int j, String psf) {  this.i = i;  this.s = s;  this.j = j;  this.psf = psf;  }  }  public static void main(String[] args) {  int arr[] = {3, 3, 0, 2, 1, 2, 4, 2, 0, 0};  solution(arr);  }  private static void solution(int[] arr) {  Integer[] dp = new Integer[arr.length];  dp[arr.length - 1] = 0;  for (int i = arr.length - 2; i >= 0; i--) {  int steps = arr[i];  int min = Integer.MAX\_VALUE;  for (int j = 1; j <= steps && i + j < arr.length; j++) {  if (dp[i + j] != null && dp[i + j] < min) {  min = dp[i + j];  }  }  if (min != Integer.MAX\_VALUE) {  dp[i] = min + 1;  }  }  System.out.println(dp[0]);  ArrayDeque<Pair> queue = new ArrayDeque<>();  queue.add(new Pair(0, arr[0], dp[0], 0 + ""));  while (queue.size() > 0) {  Pair rem = queue.removeFirst();  if(rem.j==0){  System.out.println(rem.psf+".");  }  for (int j = 1; j <= rem.s && rem.i + j < arr.length; j++) {  int ci = rem.i + j;  if (dp[ci] != null && dp[ci] == rem.j - 1) {  queue.add(new Pair(ci, arr[ci], dp[ci], rem.psf + "->" + ci));  }  }  }  } |
| Temple offerings:  public static void main(String[] args) {  int[] height = {2, 3, 5, 6, 4, 8, 9};  System.out.println(totalOfferings(height));  }  private static int totalOfferings(int[] height) {  int[] larr = new int[height.length];  larr[0] = 1;  for (int i = 1; i < height.length; i++) {  if (height[i] > height[i - 1]) {  larr[i] = larr[i - 1] + 1;  } else {  larr[i] = 1;  }  }  int[] rarr = new int[height.length];  rarr[rarr.length - 1] = 1;  for (int i = height.length - 2; i >= 0; i--) {  if (height[i] > height[i + 1]) {  rarr[i] = rarr[i + 1] + 1;  } else {  rarr[i] = 1;  }  }  int ans = 0;  for (int i = 0; i < height.length; i++) {  ans += Math.max(larr[i], rarr[i]);  }  return ans;  } | Wild card pattern matching:  public static void main(String[] args) {  String s1 = "baaabab";  String s2 = "ba\*a";  System.out.println(sol(s1, s2));  }  private static boolean sol(String str, String pattern) {  boolean[][] dp = new boolean[pattern.length() + 1][str.length() + 1];  for (int i = dp.length - 1; i >= 0; i--) {  for (int j = dp[0].length - 1; j >= 0; j--) {  if (i == dp.length - 1 && j == dp[0].length - 1) {  dp[i][j] = true;  } else if (i == dp.length - 1) {  dp[i][j] = false;  } else if (j == dp[0].length - 1) {  if (pattern.charAt(i) == '\*') {  dp[i][j] = dp[i + 1][j];  } else {  dp[i][j] = false;  }  } else {  if (pattern.charAt(i) == '?') {  dp[i][j] = dp[i + 1][j + 1];  } else if (pattern.charAt(i) == '\*') {  dp[i][j] = dp[i + 1][j] || dp[i][j + 1];  } else if (pattern.charAt(i) == str.charAt(j)) {  dp[i][j] = dp[i + 1][j + 1];  } else {  dp[i][j] = false;  }  }  }  }  return dp[0][0];  } |