1. Random Graph Generation

1.1 Random Permutation of n Objects

Most of the random graph generation procedures in this chapter make use of a subprogram that will generate a random permutation of n objects. A random permutation of the set of integers {1, 2, …, n} is produced by a sequence of random interchanges.

Procedure parameters:

void randomPermutation (*n, ran, perm*)

n: int;
entry: number of objects to be permuted.
ran: java.util.Random;
entry: a `pseudorandom` number generator of type *java.util.Random* that has already been initialized by a "seed" of type *long*.
perm: int[*n*+1];
exit: *perm[i]* is the random permutation at *i*, *i* = 1,2,…,*n*.

```
public static void randomPermutation(int n, Random ran, int perm[])
{
  int i,j,k;

  for (i=1; i<=n; i++)
    perm[i] = i;
  for (i=1; i<=n; i++) {
    j = (int)(i + ran.nextDouble() * (n + 1 - i));
    k = perm[i];
    perm[i] = perm[j];
    perm[j] = k;
  }
}
```

Example:

Generate a random permutation of 9 elements.

```
package GraphAlgorithms;
import java.util.Random;

public class Test_randomPermutation extends Object {

  public static void main(String args[]) {

    int n = 9;
    long seed = 1;
    int perm[] = new int[n+1];
```

```
    Random ran = new Random(seed);

    GraphAlgo.randomPermutation(n, ran, perm);
    System.out.println("A random permuation of " + n + " elements:");
    for (int i=1; i<=n; i++)
      System.out.print(" " + perm[i]);
    System.out.println();
  }
}
```

Output:

```
A random permuation of 9 elements:
 7 5 4 2 9 6 3 1 8
```

1.2 Random Graph

The following procedure generates a random graph with some given number of nodes and edges. The type of random graph can be one of the following:

a. Simple (no loops or parallel edges) or nonsimple
b. Directed or undirected
c. Directed acyclic
d. Weighted or unweighted

In generating a random graph, pairs of random integers in the appropriate range are generated. For directed graphs, the pairs are ordered. For a simple graph, loops and parallel edges are excluded. In the case of a directed acyclic graph, a random permutation of n objects (perm[1], perm[2], …, perm[n]) is first generated. Then m edges in the form of (perm[i],perm[j]) with i < j are generated.

Procedure parameters:

int randomGraph (*n, m, seed, simple, directed, acyclic, weighted, minweight, maxweight, nodei, nodej, weight*)

randomGraph: int;

	exit:	the method returns the following error code: 0: solution found with normal execution 1: value of m is too large, should be at most $n*(n-1)/2$ for simple undirected graph or directed acyclic graph, and $n*(n-1)$ for simple directed graph.
n:	int;	
	entry:	number of nodes of the graph. Nodes of the graph are labeled from 1 to n.
m:	int;	
	entry:	number of edges of the graph. If m is greater than the maximum number of edges in a graph then a complete graph is generated.

seed: long;
entry: seed for initializing the random number generator.
simple: boolean;
entry: *simple* = true if the graph is simple, false otherwise.
directed: boolean;
entry: *directed* = true if the graph is directed, false otherwise.
acyclic: boolean;
entry: *acyclic* = true if it is a directed acyclic graph, false otherwise.
weighted: boolean;
entry: *weighted* = true if the graph is weighted, false otherwise.
minweight: int;
entry: minimum weight of the edges;
if *weighted* = false then this value is ignored.
maxweight: int;
entry: maximum weight of the edges;
if *weighted* = false then this value is ignored.
nodei, nodej: int[*m*+1];
exit: the i-th edge is from node *nodei[i]* to node *nodej[i]*,
for i = 1,2,...,*m*.
weight: int[*m*+1];
exit: *weight[i]* is the weight of the i-th edge, for i = 1,2,...,*m*;
if *weighted* = false then this array is ignored.

```
public static int randomGraph(int n, int m, long seed, boolean simple,
      boolean directed, boolean acyclic, boolean weighted, int minweight,
      int maxweight, int nodei[], int nodej[], int weight[])
{
  int maxedges,nodea,nodeb,numedges,temp;
  int dagpermute[] = new int[n + 1];
  boolean adj[][] = new boolean[n+1][n+1];
  Random ran = new Random(seed);

  // initialize the adjacency matrix
  for (nodea=1; nodea<=n; nodea++)
    for (nodeb=1; nodeb<=n; nodeb++)
      adj[nodea][nodeb] = false;
  numedges = 0;
  // check for valid input data
  if (simple) {
    maxedges = n * (n - 1);
    if (!directed) maxedges /= 2;
    if (m > maxedges) return 1;
  }
  if (acyclic) {
    maxedges = (n * (n - 1)) / 2;
    if (m > maxedges) return 1;
    randomPermutation(n,ran,dagpermute);
  }
  while (numedges < m) {
```

```
    nodea = ran.nextInt(n) + 1;
    nodeb = ran.nextInt(n) + 1;
    if (simple || acyclic)
      if (nodea == nodeb) continue;
    if ((simple && (!directed)) || acyclic)
      if (nodea > nodeb) {
        temp = nodea;
        nodea = nodeb;
        nodeb = temp;
      }
    if (acyclic) {
      nodea = dagpermute[nodea];
      nodeb = dagpermute[nodeb];
    }
    if ((!simple) || (simple && (!adj[nodea][nodeb]))) {
      numedges++;
      nodei[numedges] = nodea;
      nodej[numedges] = nodeb;
      adj[nodea][nodeb] = true;
      if (weighted)
        weight[numedges] = (int)(minweight +
              ran.nextDouble() * (maxweight + 1 - minweight));
    }
  }
  return 0;
}
```

Example:

Generate a random simple undirected nonacyclic weighted graph with 5 nodes and 8 edges.

```
package GraphAlgorithms;

public class Test_randomGraph extends Object {

  public static void main(String args[]) {

    int k;
    int n = 5;
    int m = 8;
    long seed = 1;
    boolean simple=true, directed=false, acyclic=false, weighted=true;
    int minweight = 90;
    int maxweight = 99;
    int nodei[] = new int[m+1];
    int nodej[] = new int[m+1];
    int weight[] = new int[m+1];

    k = GraphAlgo.randomGraph(n,m,seed,simple,directed,acyclic,
```

```
            weighted,minweight,maxweight,nodei,nodej,weight);
    if (k != 0)
      System.out.println("Invalid input data, error code = " + k);
    else {
      System.out.println("List of edges:\n   from to  weight");
      for (k=1; k<=m; k++)
        System.out.println("     " + nodei[k] + "   " + nodej[k] +
                           "    " + weight[k]);
    }
  }
}
```

Output:

```
List of edges:
   from to  weight
     1   4    94
     2   5    99
     4   5    99
     3   5    99
     2   3    93
     1   5    96
     1   3    91
     1   2    95
```

1.3 Random Bipartite Graph

A *bipartite graph* is an undirected graph where the nodes can be partitioned into two sets such that no edge connects nodes in the same set. The following procedure generates a random simple bipartite graph. It is optional to specify the required number of edges. The method generates random edges connecting the nodes of the two sets.

Procedure parameters:

void randomBipartiteGraph (*n1, n2, m, seed, nodei, nodej*)

n1: int;
entry: number of nodes in the first set of the bipartite graph.

n2: int;
entry: number of nodes in the second set of the bipartite graph. Nodes of the graph are labeled from 1 to *n1*+*n2*.

m: int;
entry: required number of edges of the graph.
If *m* is zero then a random bipartite graph will be generated with random edges connecting the nodes in the first set to

the nodes in the second set, and the total number of edges will be returned in *nodei[0]*.

seed: long;
entry: seed for initializing the random number generator.

nodei, nodej: int[*m*+1] for *m* > 0 ; int[*n1***n2*+1] for *m* = 0;
exit: The i-th edge is from node *nodei[i]* to node *nodej[i]*, for i = 1,2,..., *nodei[0]*.

```
public static void randomBipartiteGraph(int n1, int n2, int m,
                         long seed, int nodei[], int nodej[])
{
  int n,nodea,nodeb,nodec,numedges;
  boolean adj[][] = new boolean[n1+n2+1][n1+n2+1];
  boolean temp;
  Random ran = new Random(seed);

  n = n1 + n2;
  // initialize the adjacency matrix
  for (nodea=1; nodea<=n; nodea++)
    for (nodeb=1; nodeb<=n; nodeb++)
      adj[nodea][nodeb] = false;

  if (m != 0) {
    if (m > n1 * n2) m = n1 * n2;
    numedges = 0;
    // generate a simple bipartite graph with exactly m edges
    while (numedges < m) {
      // generate a random integer in interval [1, n1]
      nodea = (int)(1 + ran.nextDouble() * n1);
      // generate a random integer in interval [n1+1, n]
      nodeb = (int)(n1 + 1 + ran.nextDouble() * n2);
      if (!adj[nodea][nodeb]) {
        // add the edge (nodei,nodej)
        adj[nodea][nodeb] = adj[nodeb][nodea] = true;
        numedges++;
      }
    }
  }
  else {
    // generate a random adjacency matrix with edges from
    // nodes of group [1, n1] to nodes of group [n1+1, n]
    for (nodea=1; nodea<=n1; nodea++)
      for (nodeb=n1+1; nodeb<=n; nodeb++)
        adj[nodea][nodeb] = adj[nodeb][nodea] =
                    (ran.nextInt(2) == 0) ? false : true;
  }
  // random permutation of rows and columns
  for (nodea=1; nodea<=n; nodea++) {
    nodec = (int)(nodea + ran.nextDouble() * (n + 1 - nodea));
    for (nodeb=1; nodeb<=n; nodeb++) {
```

```
      temp = adj[nodec][nodeb];
      adj[nodec][nodeb] = adj[nodea][nodeb];
      adj[nodea][nodeb] = temp;
    }
    for (nodeb=1; nodeb<=n; nodeb++) {
      temp = adj[nodeb][nodec];
      adj[nodeb][nodec] = adj[nodeb][nodea];
      adj[nodeb][nodea] = temp;
    }
  }
  numedges = 0;
  for (nodea=1; nodea<=n; nodea++)
    for (nodeb=nodea+1; nodeb<=n; nodeb++)
      if (adj[nodea][nodeb]) {
        numedges++;
        nodei[numedges] = nodea;
        nodej[numedges] = nodeb;
      }
  nodei[0] = numedges;
}
```

Example:

Generate a random bipartite graph of 6 edges with 3 nodes in the first set and 4 nodes in the second set.

```
package GraphAlgorithms;

public class Test_randomBipartiteGraph extends Object {

  public static void main(String args[]) {

    int n1 = 3;
    int n2 = 4;
    int m = 6;
    long seed = 1;
    int nodei[] = new int[m+1];
    int nodej[] = new int[m+1];

    GraphAlgo.randomBipartiteGraph(n1,n2,m,seed,nodei,nodej);
    System.out.println("List of edges:\n   from to");
    for (int k=1; k<=nodei[0]; k++)
      System.out.println("     " + nodei[k] + "   " + nodej[k]);
  }
}
```

Output:

```
List of edges:
   from to
     1   2
     1   6
     3   6
     4   5
     5   6
     5   7
```

1.4 Random Regular Graph

The following procedure generates a random simple undirected regular graph with a given number of nodes *n* and the degree *d* of each node. An implementation in programming language C is given in [JK91].

Step 1. Start with a graph with n nodes and no edges.
Step 2. If every node in the graph is of degree *d* then stop.
Step 3. Randomly choose two nonadjacent nodes u and v, each of degree less than *d*. If such nodes u and v are not found then proceed to Step 5.
Step 4. Add the edge (u,v) to the graph, and go back to Step 2.
Step 5. If there is one node of degree less than *d* then proceed to Step 7.
Step 6. From the graph, randomly choose two adjacent nodes r and s each of degree less than *d*. Randomly choose two adjacent nodes p and q such that p is not adjacent to r, and q is not adjacent to s. Remove the edge (p,q) and add the two edges (p,r) and (q,s) to the graph. Go back to Step 2.
Step 7. There is exactly one node r of degree less than *d*. Randomly choose two adjacent nodes p and q that are not adjacent to r. Remove the edge (p,q) and add the two edges (p,r) and (q,r) to the graph. Go back to Step 2.

Procedure parameters:

int randomRegularGraph (*n, degree, seed, nodei, nodej*)

randomRegularGraph: int;
exit: the method returns the following error code:
0: solution found with normal execution
1: invalid input, if *degree* is odd then *n* must be even
2: value of *n* should be greater than *degree*

n: int;
entry: number of nodes of the graph.
Nodes of the graph are labeled from 1 to *n*.

degree: int;
entry: required degree of each node.
If the value of *degree* is odd then the value *n* must be even.

seed: long;
entry: seed for initializing the random number generator.

nodei, nodej: int[(*n***degree*)/2 + 1];
exit: the i-th edge is from node *nodei[i]* to node *nodej[i]*, for i = 1, 2, …, (*n***degree*)/2.

```
public static int randomRegularGraph(int n, int degree, long seed,
                                     int nodei[], int nodej[])
{
  int i,j,numedges,p,q,r=0,s=0,u,v=0;
  int permute[] = new int[n + 1];
  int deg[] = new int[n + 1];
  boolean adj[][] = new boolean[n+1][n+1];
  boolean more;
  Random ran = new Random(seed);

  // initialize the adjacency matrix
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      adj[i][j] = false;
  // initialize the degree of each node
  for (i=1; i<=n; i++)
    deg[i] = 0;
  // check input data consistency
  if ((degree % 2) != 0)
    if ((n % 2) != 0) return 1;
  if (n <= degree) return 2;
  // generate the regular graph
  iterate:
  while (true) {
    randomPermutation(n,ran,permute);
    more = false;
    // find two non-adjacent nodes each has less than required degree
    u = 0;
    for (i=1; i<=n; i++)
      if (deg[permute[i]] < degree) {
         v = permute[i];
         more = true;
         for (j=i+1; j<=n; j++) {
           if (deg[permute[j]] < degree) {
             u = permute[j];
             if (!adj[v][u]) {
               // add edge (u,v) to the random graph
               adj[v][u] = adj[u][v] = true;
               deg[v]++;
               deg[u]++;
               continue iterate;
             }
             else {
               // both r & s are less than the required degree
               r = v;
               s = u;
```

```
          }
        }
      }
    }
  if (!more) break;
  if (u == 0) {
    r = v;
    // node r has less than the required degree,
    // find two adjacent nodes p and q non-adjacent to r.
    for (i=1; i<=n-1; i++) {
      p = permute[i];
      if (r != p)
        if (!adj[r][p])
          for (j=i+1; j<=n; j++) {
            q = permute[j];
            if (q != r)
              if (adj[p][q] && (!adj[r][q])) {
                // add edges (r,p) & (r,q), delete edge (p,q)
                adj[r][p] = adj[p][r] = true;
                adj[r][q] = adj[q][r] = true;
                adj[p][q] = adj[q][p] = false;
                deg[r]++;
                deg[r]++;
                continue iterate;
              }
          }
    }
  }
  else {
    // nodes r and s of less than required degree, find two
    // adjacent nodes p & q such that (p,r) & (q,s) are not edges.
    for (i=1; i<=n; i++) {
      p = permute[i];
      if ((p != r) && (p != s))
        if (!adj[r][p])
          for (j=1; j<=n; j++) {
            q = permute[j];
            if ((q != r) && (q != s))
              if (adj[p][q] && (!adj[s][q])) {
                // remove edge (p,q), add edges (p,r) & (q,s)
                adj[p][q] = adj[q][p] = false;
                adj[r][p] = adj[p][r] = true;
                adj[s][q] = adj[q][s] = true;
                deg[r]++;
                deg[s]++;
                continue iterate;
              }
          }
    }
  }
```

```
  }
  numedges = 0;
  for (i=1; i<=n; i++)
    for (j=i+1; j<=n; j++)
      if (adj[i][j]) {
        numedges++;
        nodei[numedges] = i;
        nodej[numedges] = j;
      }
  return 0;
}
```

Example:

Generate a random regular graph with 8 nodes of degree 3.

```
package GraphAlgorithms;

public class Test_randomRegularGraph extends Object {

  public static void main(String args[]) {

    int k;
    int n = 8;
    int degree = 3;
    long seed = 1;
    int edges = (n * degree) / 2;
    int nodei[] = new int[edges+1];
    int nodej[] = new int[edges+1];

    k = GraphAlgo.randomRegularGraph(n,degree,seed,nodei,nodej);
    if (k != 0)
      System.out.println("invalid input data, error code = " + k);
    else {
      System.out.print("List of edges:\n from: ");
      for (k=1; k<=edges; k++)
        System.out.print("  " + nodei[k]);
      System.out.print("\n   to: ");
      for (k=1; k<=edges; k++)
        System.out.print("  " + nodej[k]);
      System.out.println();
    }
  }
}
```

Output:

```
List of edges:
 from:   1  1  1  2  2  3  3  4  4  5  5  6
   to:   2  3  8  5  7  4  6  6  8  7  8  7
```

1.5 Random Spanning Tree

The following procedure generates a random undirected spanning tree with a given number of nodes n by means of the greedy method. The method first generates a random permutation of n objects (perm[1], perm[2], …, perm[n]). Start with a partial tree consisting of a single node perm[1]. Suppose the current nodes in the partial tree are perm[1], perm[2],…, perm[i]. Randomly choose a node k from the partial tree, then include the node perm[i+1] and the edge (perm[i+1], k) into the partial tree. Terminate when all the n nodes are included in the partial tree.

Procedure parameters:

void randomSpanningTree (*n, seed, weighted, minweight, maxweight, nodei, nodej, weight*)

n:	int;	
	entry:	number of nodes of the tree. Nodes of the tree are labeled from 1 to *n*.
seed:	long;	
	entry:	seed for initializing the random number generator.
weighted:	boolean;	
	entry:	*weighted* = true if the tree is weighted, false otherwise.
minweight:	int;	
	entry:	minimum weight of the edges; if *weighted* = false then this value is ignored.
maxweight:	int;	
	entry:	maximum weight of the edges; if *weighted* = false then this value is ignored.
nodei, nodej:	int[*n*];	
	exit:	the i-th edge is from node *nodei[i]* to node *nodej[i]*, for i = 1,2,…,*n*–1.
weight:	int[*n*];	
	exit:	*weight[i]* is the weight of the i-th edge, for i = 1,2,…,*n*–1; if *weighted* = false then this array is ignored.

```
public static void randomSpanningTree(int n, long seed, boolean weighted,
     int minweight, int maxweight, int nodei[], int nodej[], int weight[])
{
  int nodea,nodeb,numedges;
  int permute[] = new int[n + 1];
  Random ran = new Random(seed);

  // generate a random permutation of n objects
  randomPermutation(n,ran,permute);

  numedges = 0;
```

```
  // add n-1 random edges by the greedy method
  for (nodea=2; nodea<=n; nodea++) {
    nodeb = ran.nextInt(nodea - 1) + 1;
    numedges++;
    nodei[numedges] = permute[nodea];
    nodej[numedges] = permute[nodeb];
    if (weighted)
      weight[numedges] = (int)(minweight +
                         ran.nextDouble() * (maxweight + 1 - minweight));
  }
}
```

Example:

Generate a weighted random spanning tree of 8 nodes with edge weights in the range of [90, 99].

```
package GraphAlgorithms;

public class Test_randomSpanningTree extends Object {

  public static void main(String args[]) {

    int n = 8;
    long seed = 1;
    boolean weighted=true;
    int minweight = 90;
    int maxweight = 99;
    int nodei[] = new int[n];
    int nodej[] = new int[n];
    int weight[] = new int[n];

    GraphAlgo.randomSpanningTree(n,seed,weighted,minweight,maxweight,
                                 nodei,nodej,weight);
    System.out.println("List of edges:\n   from to  weight");
    for (int k=1; k<=n-1; k++)
      System.out.println("     " + nodei[k] + "    " + nodej[k] +
                         "    " + weight[k]);
  }
}
```

Output:

```
List of edges:
   from to  weight
     4   6    99
     2   6    93
     5   4    91
     8   4    95
     1   2    95
     3   8    96
     7   8    97
```

1.6 Random Labeled Tree

The following procedure generates a random labeled tree with a given number of nodes n. The method first constructs a Prüfer code by choosing n–2 integers $[p_1,p_2,\ldots,p_{n-2}]$ randomly among integers 1,2,…,n. Then a random labeled tree T is obtained by converting the Prüfer code [NW78]. The Prüfer code conversion starts with an empty tree T and makes use of two arrays L_1 and L_2. L_1 is initialized to contain n integers [1,2,…,n], and L_2 initially contains the n–2 integers of the Prüfer code $[p_1,p_2,\ldots, p_{n-2}]$. Let w be the smallest number in L_1 but not in L_2. Add the edge (w, p_1) to T. Delete w from L_1 and p_1 from L_2. Again let w be the smallest number in L_1 but not in L_2. Add the edge (w, p_2) to T. Delete w from L_1 and p_2 from L_2. This process stops when L_2 is empty and L_1 contains exactly two elements u and v. The tree T is completed by adding the edge (u,v).

Procedure parameters:

void randomLabeledTree (*n, seed, sol*)

n:	int;	
	entry:	number of nodes of the tree, $n > 2$. Nodes of the tree are labeled from 1 to *n*.
seed:	long;	
	entry:	seed for initializing the random number generator.
sol:	int[*n*+1];	
	exit:	the i-th edge of the tree is (i, sol[i]), i = 1,2,…,*n*–1.

```
public static void randomLabeledTree(int n, long seed, int sol[])
{
  int i,idxa,idxb,idxc,idxd,idxe,nminus2;
  int neighbor[] = new int[n + 1];
  int prufercode[] = new int[n];
  Random ran = new Random(seed);

  if (n <= 2) return;
  // select n-2 integers at random in [1,n]
```

```
  for (i=1; i<=n-2; i++)
    prufercode[i] = (int)(1 + ran.nextDouble() * n);
  // compute the tree from the Pruefer code
  for (i=1; i<=n; i++)
    sol[i] = 0;
  nminus2 = n - 2;
  for (i=1; i<=nminus2; i++) {
    idxc = prufercode[n-1-i];
    if (sol[idxc] == 0) prufercode[n-1-i] = -idxc;
    sol[idxc] = -1;
  }
  idxb = 1;
  prufercode[n-1] = n;
  idxa = 0;
  while (true) {
    if (sol[idxb] != 0) {
      idxb++;
      continue;
    }
    idxd = idxb;
    while (true) {
      idxa++;
      idxe = Math.abs(prufercode[idxa]);
      sol[idxd] = idxe;
      if (idxa == n-1) {
        for (i=1; i<=nminus2; i++)
          prufercode[i] = Math.abs(prufercode[i]);
        return;
      }
      if (prufercode[idxa] > 0) break;
      if (idxe > idxb) {
        sol[idxe] = 0;
        break;
      }
      idxd = idxe;
    }
  }
}
```

Example:

Generate a random labeled tree with 7 nodes.

```
package GraphAlgorithms;

public class Test_randomLabeledTree extends Object {

  public static void main(String args[]) {

    int n = 7;
```

```
    long seed = 1;
    int sol[] = new int[n+1];

    GraphAlgo.randomLabeledTree(n,seed,sol);
    System.out.print("List of edges:\n from: ");
    for (int k=1; k<=n-1; k++)
      System.out.print("  " + k);
    System.out.print("\n   to: ");
    for (int k=1; k<=n-1; k++)
      System.out.print("  " + sol[k]);
    System.out.println();
  }
}
```

Output:

```
List of edges:
 from:   1  2  3  4  5  6
   to:   6  3  7  3  2  7
```

1.7 Random Unlabeled Rooted Tree

The following procedure generates a random unlabeled rooted tree with a given number of nodes n. The method calculates the numbers T_1 , T_2 , …, T_n, where T_i is the number of trees on i nodes:

$$(i-1)T_i = \sum_{v=1}^{\infty}\sum_{u=1}^{\infty} dT_{i-uv}T_v, \qquad T_1 = 1$$

A pair of positive integers u and v is chosen with a priori probability:

$$\frac{dT_{i-uv}T_v}{(i-1)T_i}$$

A random tree R_1 on n-uv nodes is chosen with probability $1/T_{n-uv}$, and a random R_2 on v nodes is chosen with probability $1/T_v$. Make u copies of R_2. A random unlabeled rooted tree on n nodes is obtained by joining the root of R_1 to the roots of each of the copies of R_2 [NW78].

Procedure parameters:

void randomUnlabeledRootedTree (*n, seed, sol*)

n:	int;	
	entry:	number of nodes of the tree, $n > 2$. Nodes of the tree are labeled from 1 to *n*.
seed:	long;	
	entry:	seed for initializing the random number generator.

sol: int[*n*+1];
exit: the i-th edge of the tree is (i, sol[i]), i = 1,2,...,*n*–1.

```
public static void randomUnlabeledRootedTree(int n, long seed, int sol[])
{
  int count,v,total,prod,curnum,nextlastroot,numt;
  int stackcounta,stackcountb,prob,p,q,r;
  int rightroot=0,u=0,w=0;
  int numtrees[] = new int[n + 1];
  int aux1[] = new int[n+1];
  int aux2[] = new int[n+1];
  boolean iter;
  Random ran = new Random(seed);

  // calculate numtrees[p], the number of trees on p nodes
  count = 1;
  numtrees[1] = 1;
  while (n > count) {
    total = 0;
    for (p=1; p<=count; p++) {
      q = count + 1;
      prod = numtrees[p] * p;
      for (r=1; r<=count; r++) {
        q -= p;
        if (q <= 0) break;
        total += numtrees[q] * prod;
      }
    }
    count++;
    numtrees[count] = total / (count - 1);
  }
  curnum = n;
  stackcounta = 0;
  stackcountb = 0;
  while (true) {
    if(curnum <= 2) {
      // a new tree placed in "sol", link to left neighbor
      sol[stackcountb+1] = rightroot;
      rightroot = stackcountb + 1;
      stackcountb += curnum;
      if (curnum > 1) sol[stackcountb] = stackcountb - 1;
      while (true) {
        curnum = aux2[stackcounta];
        if (curnum != 0) break;
        // stack counter is decremented as (u,v) is read
        u = aux1[stackcounta];
        stackcounta--;
        w = stackcountb - rightroot + 1;
        nextlastroot = sol[rightroot];
        numt = rightroot + (u - 1) * w - 1;
```

```
        if (u != 1) {
          // make u copies of the last tree
          for (p=rightroot; p<=numt; p++) {
            sol[p+w] = sol[p] + w;
            if((p-1)-((p-1)/w)*w == 0) sol[p+w] = nextlastroot;
          }
        }
        stackcountb = numt + w;
        if (stackcountb == n) return;
        rightroot = nextlastroot;
      }
      aux2[stackcounta] = 0;
      continue;
    }
    // choose a pair (u,v) with a priori probability
    prob = (int)((curnum - 1) * numtrees[curnum] * ran.nextDouble());
    v = 0;
    iter = true;
    while (iter) {
      v++;
      prod = v * numtrees[v];
      w = curnum;
      u = 0;
      iter = false;
      do {
        u++;
        w -= v;
        if (w < 1) {
          iter = true;
          break;
        }
        prob -= numtrees[w] * prod;
      } while (prob >= 0);
    }
    stackcounta++;
    aux1[stackcounta] = u;
    aux2[stackcounta] = v;
    curnum = w;
  }
}
```

Example:

Generate a random unlabeled rooted tree with 5 nodes.

```
package GraphAlgorithms;

public class Test_randomUnlabeledRootedTree extends Object {

  public static void main(String args[]) {
```

```
    int n = 5;
    long seed = 1;
    int sol[] = new int[n+1];

    GraphAlgo.randomUnlabeledRootedTree(n,seed,sol);
    System.out.println("List of edges:\n   from to");
    for (int k=2; k<=n; k++)
      System.out.println("     " + k + "   " + sol[k]);
  }
}
```

Output:

```
List of edges:
   from to
     2   1
     3   2
     4   2
     5   4
```

1.8 Random Connected Graph

The following procedure [JK91] generates a random undirected connected simple graph with some given number of nodes and edges. A random spanning tree is first generated. Then random edges are added until the required number of edges is reached.

Procedure parameters:

int randomConnectedGraph (*n, m, seed, weighted, minweight, maxweight, nodei, nodej, weight*)

randomConnectedGraph: int;
exit: the method returns the following error code:
0: solution found with normal execution
1: value of m is too small, should be at least $n-1$
2: value of m is too large, should be at most $n*(n-1)/2$

n: int;
entry: number of nodes of the graph.
Nodes of the graph are labeled from 1 to n.

m: int;
entry: number of edges of the graph.
If m is less than $n-1$ then m will be set to $n-1$.
If m is greater than the maximum number of edges in a graph then a complete graph is generated.

seed: long;
entry: seed for initializing the random number generator.
weighted: boolean;
entry: *weighted* = true if the graph is weighted, false otherwise.
minweight: int;
entry: minimum weight of the edges;
if *weighted* = false then this value is ignored.
maxweight: int;
entry: maximum weight of the edges;
if *weighted* = false then this value is ignored.
nodei, nodej: int[*n*];
exit: the i-th edge is from node *nodei[i]* to node *nodej[i]*,
for i = 1,2,...,*n*–1.
weight: int[*n*];
exit: *weight[i]* is the weight of the i-th edge, for i = 1,2,...,*n*–1;
if *weighted* = false then this array is ignored.

```
public static int randomConnectedGraph(int n, int m, long seed,
                boolean weighted, int minweight, int maxweight,
                int nodei[], int nodej[], int weight[])
{
  int maxedges,nodea,nodeb,numedges,temp;
  int permute[] = new int[n + 1];
  boolean adj[][] = new boolean[n+1][n+1];
  Random ran = new Random(seed);

  // initialize the adjacency matrix
  for (nodea=1; nodea<=n; nodea++)
    for (nodeb=1; nodeb<=n; nodeb++)
      adj[nodea][nodeb] = false;
  numedges = 0;
  // check for valid input data
  if (m < (n - 1)) return 1;
  maxedges = (n * (n - 1)) / 2;
  if (m > maxedges) return 2;

  // generate a random spanning tree by the greedy method
  randomPermutation(n,ran,permute);
  for (nodea=2; nodea<=n; nodea++) {
    nodeb = ran.nextInt(nodea - 1) + 1;
    numedges++;
    nodei[numedges] = permute[nodea];
    nodej[numedges] = permute[nodeb];
    adj[permute[nodea]][permute[nodeb]] = true;
    adj[permute[nodeb]][permute[nodea]] = true;
    if (weighted)
      weight[numedges] = (int)(minweight +
                          ran.nextDouble() * (maxweight + 1 - minweight));
  }
  // add the remaining edges randomly
```

```
  while (numedges < m) {
    nodea = ran.nextInt(n) + 1;
    nodeb = ran.nextInt(n) + 1;
    if (nodea == nodeb) continue;
    if (nodea > nodeb) {
      temp = nodea;
      nodea = nodeb;
      nodeb = temp;
    }
    if (!adj[nodea][nodeb]) {
      numedges++;
      nodei[numedges] = nodea;
      nodej[numedges] = nodeb;
      adj[nodea][nodeb] = true;
      if (weighted)
        weight[numedges] = (int)(minweight +
                           ran.nextDouble() * (maxweight + 1 - minweight));
    }
  }
  return 0;
}
```

Example:

Generate a random undirected connected simple graph of 8 nodes and 10 edges with edge weights in the range of [90, 99].

```
package GraphAlgorithms;

public class Test_randomConnectedGraph extends Object {

  public static void main(String args[]) {

    int k;
    int n = 8;
    int m = 10;
    long seed = 1;
    boolean weighted=true;
    int minweight = 90;
    int maxweight = 99;
    int nodei[] = new int[m+1];
    int nodej[] = new int[m+1];
    int weight[] = new int[m+1];

    k = GraphAlgo.randomConnectedGraph(n,m,seed,weighted,minweight,
                                       maxweight,nodei,nodej,weight);
    if (k != 0)
      System.out.println("Invalid input data, error code = " + k);
    else {
      System.out.println("List of edges:\n from to  weight");
```

```
      for (k=1; k<=m; k++)
        System.out.println("    " + nodei[k] + "    " + nodej[k] +
                           "     " + weight[k]);
      System.out.println();
    }
  }
}
```

Output:

```
List of edges:
 from to  weight
   4   6    99
   2   6    93
   5   4    91
   8   4    95
   1   2    95
   3   8    96
   7   8    97
   2   5    98
   4   7    90
   5   7    90
```

1.9 Random Hamilton Graph

The following procedure [JK91] generates a random simple Hamilton graph with some given number of nodes and edges. The graph can be directed or undirected. The method generates a random permutation of n objects (perm[1], perm[2], …, perm[n]). The graph is initialized with the Hamilton cycle (perm[1], perm[2]), (perm[2], perm[3]), …, (perm[n–1], perm[n]), (perm[n], perm[1]). Then random edges are added into the graph until the required number of edges is reached.

Procedure parameters:

int randomHamiltonGraph (*n, m, seed, directed, weighted, minweight, maxweight, nodei, nodej, weight*)

randomHamiltonGraph: int;
exit: the method returns the following error code:
0: solution found with normal execution
1: value of *m* is too small, should be at least *n*
2: value of *m* is too large, should be at most $n*(n-1)/2$ for simple undirected graph, and $n*(n-1)$ for simple directed graph.

n: int;
entry: number of nodes of the graph.
Nodes of the graph are labeled from 1 to *n*.

m: int;
entry: number of edges of the graph.
If *m* is greater than the maximum number of edges in a graph then a complete graph is generated.
If *m* is less than *n* then *m* is set equal to *n*.

seed: long;
entry: seed for initializing the random number generator.

directed: boolean;
entry: *directed* = true if the graph is directed, false otherwise.

weighted: boolean;
entry: *weighted* = true if the graph is weighted, false otherwise.

minweight: int;
entry: minimum weight of the edges;
if *weighted* = false then this value is ignored.

maxweight: int;
entry: maximum weight of the edges;
if *weighted* = false then this value is ignored.

nodei, nodej: int[*m*+1];
exit: the i-th edge is from node *nodei[i]* to node *nodej[i]*, for i = 1,2,...,*m*. The Hamilton cycle is given by the first *n* elements of these two arrays.

weight: int[*m*+1];
exit: *weight[i]* is the weight of the i-th edge, for i = 1,2,...,*m*;
if *weighted* = false then this array is ignored.

```
public static int randomHamiltonGraph(int n, int m, long seed,
            boolean directed, boolean weighted, int minweight,
            int maxweight, int nodei[], int nodej[], int weight[])
{
  int k,maxedges,nodea,nodeb,numedges,temp;
  int permute[] = new int[n + 1];
  boolean adj[][] = new boolean[n+1][n+1];
  Random ran = new Random(seed);

  // initialize the adjacency matrix
  for (nodea=1; nodea<=n; nodea++)
    for (nodeb=1; nodeb<=n; nodeb++)
      adj[nodea][nodeb] = false;
  // adjust value of m if needed
  if (m < n) return 1;
  maxedges = n * (n - 1);
  if (!directed) maxedges /= 2;
  if (m > maxedges) return 2;
  numedges = 0;
  // generate a random permutation
  randomPermutation(n,ran,permute);
  // obtain the initial cycle
  for (k=1; k<=n; k++) {
    if (k == n) {
      nodea = permute[n];
```

```
      nodeb = permute[1];
    }
    else {
      nodea = permute[k];
      nodeb = permute[k + 1];
    }
    numedges++;
    nodei[numedges] = nodea;
    nodej[numedges] = nodeb;
    adj[nodea][nodeb] = true;
    if (!directed) adj[nodeb][nodea] = true;
    if (weighted)
      weight[numedges] = (int)(minweight +
                          ran.nextDouble() * (maxweight + 1 - minweight));
  }
  // add the remaining edges randomly
  while (numedges < m) {
    nodea = ran.nextInt(n) + 1;
    nodeb = ran.nextInt(n) + 1;
    if (nodea == nodeb) continue;
    if ((nodea > nodeb) && (!directed)) {
      temp = nodea;
      nodea = nodeb;
      nodeb = temp;
    }
    if (!adj[nodea][nodeb]) {
      numedges++;
      nodei[numedges] = nodea;
      nodej[numedges] = nodeb;
      adj[nodea][nodeb] = true;
      if (weighted)
        weight[numedges] = (int)(minweight +
                            ran.nextDouble() * (maxweight + 1 - minweight));
    }
  }
  return 0;
}
```

Example:

Generate a random simple Hamilton graph of 7 nodes and 10 edges with edge weights in the range of [90, 99].

```
package GraphAlgorithms;

public class Test_randomHamiltonGraph extends Object {

  public static void main(String args[]) {

    int k;
```

```
    int n = 7;
    int m = 10;
    long seed = 1;
    boolean weighted=true;
    boolean directed=true;
    int minweight = 90;
    int maxweight = 99;
    int nodei[] = new int[m+1];
    int nodej[] = new int[m+1];
    int weight[] = new int[m+1];

    k = GraphAlgo.randomHamiltonGraph(n,m,seed,directed,weighted,minweight,
                                      maxweight,nodei,nodej,weight);
    if (k != 0)
      System.out.println("Invalid input data, error code = " + k);
    else {
      System.out.println("List of edges:\n from to  weight");
      for (k=1; k<=m; k++)
        System.out.println("   " + nodei[k] + "   " + nodej[k] +
                           "    " + weight[k]);
    }
  }
}
```

Output:

```
List of edges:
 from to  weight
   6   4    99
   4   2    99
   2   5    99
   5   7    93
   7   1    93
   1   3    92
   3   6    95
   6   3    97
   3   4    91
   4   7    91
```

1.10 Random Maximum Flow Network

The following procedure [JK91] generates a random simple weighted directed graph of n nodes in which node 1 (the source) has no incoming edges and node n (the sink) has no outgoing edges. The procedure first attempts to generate random paths until either the required number of edges is reached or every node is on a directed path from the source to the sink. If each node has already been included in some directed path from the source to the sink then additional edges

(p,q), where 0<p<n and 1<q≤n, are generated randomly until the required number of edges is reached.

Procedure parameters:

void randomMaximumFlowNetwork (*n, m, seed, minweight,maxweight, nodei, nodej, weight*)

n: int;
entry: number of nodes of the network.
Nodes of the graph are labeled from 1 to *n*.
Node 1 is the source and node *n* is the sink.

m: int;
entry: number of required edges of the directed graph.

seed: long;
entry: seed for initializing the random number generator.

minweight: int;
entry: minimum weight of the edges.

maxweight: int;
entry: maximum weight of the edges.

nodei, nodej: int[*m*+1];
exit: *nodei[0]* returns the actual number of edges generated.
If $m \le (n * n - 3 * n + 3)$ then *nodei[0]* = m, otherwise
nodei[0] = $(n * n - 3 * n + 3)$.
The i-th edge is from node *nodei[i]* to node *nodej[i]*,
for i = 1,2,…,*nodei[0]*.

weight: int[*m*+1];
exit: *weight[i]* is the weight of the i-th edge, for i=1,2,…,*nodei[0]*.

```
public static void randomMaximumFlowNetwork(int n, int m, long seed,
                          int minweight, int maxweight, int nodei[],
                          int nodej[], int weight[])
{
  int i,maxedges,nodea,nodeb,numedges,source,sink;
  boolean adj[][] = new boolean[n+1][n+1];
  boolean marked[] = new boolean[n+1];
  boolean more;
  Random ran = new Random(seed);

  if ((n <= 1) || (m < 1)) return;
  // initialize the adjacency matrix
  for (nodea=1; nodea<=n; nodea++)
    for (nodeb=1; nodeb<=n; nodeb++)
      adj[nodea][nodeb] = false;
  // check for valid input data
  maxedges = n * n - 3 * n + 3;
  if (m > maxedges) m = maxedges;
  nodei[0] = m;

  // node 1 is the source and node n is the sink
```

```
source = 1;
sink = n;
numedges = 0;
// initially every node is not on some path from source to sink
for (i=1; i<=n; i++)
  marked[i] = false;
// include each node on some path from source to sink */
marked[source] = true;
do {
  // choose an edge from source to some node not yet included
  do {
    // generate a random integer in interval [2,n]
    nodeb = (int)(2 + ran.nextDouble() * (n-1));
  } while (marked[nodeb]);
  marked[nodeb] = true;
  // add the edge from source to nodeb
  adj[1][nodeb] = true;
  numedges++;
  weight[numedges] = (int)(minweight +
                     ran.nextDouble() * (maxweight + 1 - minweight));
  nodei[numedges] = 1;
  nodej[numedges] = nodeb;
  if (numedges == m) return;
  // add an edge from current node to a node other than the sink
  if (nodeb != sink) {
    nodea = nodeb;
    marked[sink] = false;
    while (true) {
      do {
        // generate a random integer in interval [2,n]
        nodeb = (int)(2 + ran.nextDouble() * (n-1));
      } while (marked[nodeb]);
      marked[nodeb] = true;
      if (nodeb == sink) break;
      // add an edge from nodea to nodeb
      adj[nodea][nodeb] = true;
      numedges++;
      weight[numedges] = (int)(minweight +
                         ran.nextDouble() * (maxweight + 1 - minweight));
      nodei[numedges] = nodea;
      nodej[numedges] = nodeb;
      if (numedges == m) return;
      nodea = nodeb;
    }
    // add an edge from nodea to sink
    adj[nodea][sink] = true;
    numedges++;
    weight[numedges] = (int)(minweight +
                       ran.nextDouble() * (maxweight + 1 - minweight));
    nodei[numedges] = nodea;
```

```
      nodej[numedges] = sink;
      if (numedges == m) return;
    }
    more = false;
    for (i=1; i<n; i++)
      if (!marked[i]) {
        more = true;
        break;
      }
  } while (more);
  // add additional edges if needed
  while (numedges < m) {
    // generate a random integer in interval [1,n-1]
    nodea = (int)(1 + ran.nextDouble() * (n-1));
    // generate a random integer in interval [2,n]
    nodeb = (int)(2 + ran.nextDouble() * (n-1));
    if (!adj[nodea][nodeb]  &&  (nodea != nodeb)) {
      // add an edge from nodea to nodeb
      adj[nodea][nodeb] = true;
      numedges++;
      weight[numedges] = (int)(minweight +
                         ran.nextDouble() * (maxweight + 1 - minweight));
      nodei[numedges] = nodea;
      nodej[numedges] = nodeb;
    }
  }
}
```

Example:

Generate a random maximum flow network of 8 nodes and 10 edges with edge weights in the range of [90, 99]. Node 1 is the source and node 8 is the sink.

```
package GraphAlgorithms;

public class Test_randomMaximumFlowNetwork extends Object {

  public static void main(String args[]) {

    int n = 8;
    int m = 10;
    long seed = 1;
    int minweight = 90;
    int maxweight = 99;
    int nodei[] = new int[m+1];
    int nodej[] = new int[m+1];
    int weight[] = new int[m+1];

    GraphAlgo.randomMaximumFlowNetwork(n,m,seed,minweight,maxweight,
                                       nodei,nodej,weight);
```

```
    System.out.println("List of edges:\n from to  weight");
    for (int k=1; k<=nodei[0]; k++)
      System.out.println("   " + nodei[k] + "   " + nodej[k] +
                         "    " + weight[k]);
  }
}
```

Output:

```
List of edges:
 from to  weight
   1   7    94
   7   3    93
   3   8    90
   1   4    93
   4   5    91
   5   6    91
   6   2    96
   2   8    92
   4   3    98
   1   6    96
```

1.11 Random Isomorphic Graphs

The following procedure generates a pair of random isomorphic graphs with some given number of nodes and edges. The graph can be simple or nonsimple, directed or undirected. The method generates a random graph first, then a random permutation of n objects (perm[1], perm[2], …, perm[n]). The second isomorphic graph is obtained by renaming the vertices of the first random graph by the random permutation. The node i of the first random graph corresponds to the node perm[i] in the second graph.

Procedure parameters:

int randomIsomorphicGraphs (*n, m, seed, simple, directed, firsti, firstj, secondi, secondj, map*)

randomIsomorphicGraph: int;
exit: the method returns the following error code:
0: solution found with normal execution
1: value of *m* is too large, should be at most $n*(n-1)/2$ for simple undirected graph, and $n*(n-1)$ for simple directed graph.

n: int;
entry: number of nodes of each graph.
Nodes of each graph are labeled from 1 to *n*.

m:	int;	
	entry:	number of edges of each graph. If *m* is greater than the maximum number of edges in a graph then a complete graph is generated.
seed:	long;	
	entry:	seed for initializing the random number generator.
simple:	boolean;	
	entry:	*simple* = true if the graphs are simple, false otherwise.
directed:	boolean;	
	entry:	*directed* = true if the graphs are directed, false otherwise.
firsti, firstj:	int[*m*+1];	
	exit:	the k-th edge of the first graph is from node *firsti[k]* to node *firstj[k]*, for k = 1,2,…,*m*.
secondi, secondj:		int[*m*+1];
	exit:	the k-th edge of the second graph is from node *secondi[k]* to node *secondj[k]*, for k = 1,2,…,*m*.
map:	int[*n*+1];	
	exit:	in the graph isomorphism, node i of the first graph is renamed to node *map[i]* in the second graph, for i=1,2,…,*n*.

```
public static int randomIsomorphicGraphs(int n, int m, long seed,
       boolean simple, boolean directed, int firsti[], int firstj[],
       int secondi[], int secondj[], int map[])
{
  int k;
  Random ran = new Random(seed);

  // generate a random graph
  k = randomGraph(n,m,seed,simple,directed,false,false,0,0,firsti,firstj,map);
  if (k != 0) return k;
  // generate a random permutation
  randomPermutation(n,ran,map);
  // rename the vertices to obtain the isomorphic graph
  for (int i=1; i<=m; i++) {
    secondi[i] = map[firsti[i]];
    secondj[i] = map[firstj[i]];
  }
  return k;
}
```

Example:

Generate a pair of random isomorphic graphs with 5 nodes and 7 edges.

```
package GraphAlgorithms;

public class Test_randomIsomorphicGraphs extends Object {

  public static void main(String args[]) {
```

```
    int k;
    int n = 5;
    int m = 7;
    long seed = 1;
    boolean simple=true, directed=false;
    int map[] = new int[n+1];
    int firsti[] = new int[m+1];
    int firstj[] = new int[m+1];
    int secondi[] = new int[m+1];
    int secondj[] = new int[m+1];

    k = GraphAlgo.randomIsomorphicGraphs(n,m,seed,simple,directed,
                                firsti,firstj,secondi,secondj,map);
    if (k != 0)
      System.out.println("Invalid input data, error code = " + k);
    else {
      System.out.println("List of edges:\n  First Graph    Second Graph" +
                         "\n   from to         from to ");
      for (k=1; k<=m; k++)
        System.out.println("     " + firsti[k] + "   " + firstj[k] +
                           "           " + secondi[k] + "   " + secondj[k]);
      System.out.println("\n  Node mapping:\n   First Graph   Second Graph");
      for (k=1; k<=n; k++)
        System.out.println("        " + k + "           " + map[k]);
    }
  }
}
```

Output:

```
List of edges:
  First Graph    Second Graph
   from to         from to
     1   4           4   1
     3   4           2   1
     2   5           3   5
     4   5           1   5
     3   5           2   5
     2   3           3   2
     1   5           4   5

  Node mapping:
   First Graph   Second Graph
        1           4
        2           3
        3           2
        4           1
        5           5
```

1.12 Random Isomorphic Regular Graphs

The following procedure generates a pair of random isomorphic simple undirected regular graphs with some given number of nodes and the degree of each node. The method generates a random graph first, then a random permutation of n objects (perm[1], perm[2], …, perm[n]). The second isomorphic graph is obtained by renaming the vertices of the first random graph by the random permutation. The node i of the first random graph corresponds to the node perm[i] in the second graph.

Procedure parameters:

int randomIsomorphicRegularGraphs (*n, degree, seed, firsti, firstj,secondi, secondj, map*)

randomIsomorphicRegularGraphs: int;
exit: the method returns the following error code:
0: solution found with normal execution
1: invalid input, if *degree* is odd then *n* must be even
2: value of *n* should be greater than *degree*

n: int;
entry: number of nodes of each graph.
Nodes of each graph are labeled from 1 to *n*.

degree: int;
entry: required degree of each node.
If the value of *degree* is odd then the value *n* must be even.

seed: long;
entry: seed for initializing the random number generator.

firsti, firstj: int[(*n***degree*)/2 + 1];
exit: the k-th edge of the first graph is from node *firsti[k]* to node *firstj[k]*, for k = 1, 2, …, (*n***degree*)/2.

secondi, secondj: int[(*n***degree*)/2 + 1];
exit: the k-th edge of the second graph is from node *secondi[k]* to node *secondj[k]*, for k = 1, 2, …, (*n***degree*)/2.

map: int[*n*+1];
exit: in the graph isomorphism, node i of the first graph is renamed to node *map[i]* in the second graph, for i = 1,2,…,*n*.

```
public static int randomIsomorphicRegularGraphs(int n, int degree, long seed,
          int firsti[], int firstj[], int secondi[], int secondj[], int map[])
{
  int k;
  Random ran = new Random(seed);

  // generate a random regular graph
  k = randomRegularGraph(n,degree,seed,firsti,firstj);
  if (k != 0) return k;
  // generate a random permutation
  randomPermutation(n,ran,map);
```

```
  // rename the vertices to obtain the isomorphic graph
  for (int i=1; i<=(n*degree)/2; i++) {
    secondi[i] = map[firsti[i]];
    secondj[i] = map[firstj[i]];
  }
  return k;
}
```

Example:

Generate a pair of random isomorphic simple undirected regular graphs with 6 nodes of degree 3.

```
package GraphAlgorithms;

public class Test_randomIsomorphicRegularGraphs extends Object {

  public static void main(String args[]) {

    int k;
    int n = 6;
    int degree = 3;
    long seed = 1;
    int m = (n * degree) / 2;
    int map[] = new int[n+1];
    int firsti[] = new int[m+1];
    int firstj[] = new int[m+1];
    int secondi[] = new int[m+1];
    int secondj[] = new int[m+1];

    k = GraphAlgo.randomIsomorphicRegularGraphs(n,degree,seed,
                           firsti,firstj,secondi,secondj,map);
    if (k != 0)
      System.out.println("Invalid input data, error code = " + k);
    else {
      System.out.println("List of edges:\n  First Graph    Second Graph" +
                         "\n   from to         from to ");
      for (k=1; k<=m; k++)
        System.out.println("     " + firsti[k] + "    " + firstj[k] +
                           "           " + secondi[k] + "    " + secondj[k]);
      System.out.println("\n  Node mapping:\n   First Graph   Second Graph");
      for (k=1; k<=n; k++)
        System.out.println("        " + k + "           " + map[k]);
    }
  }
}
```

Output:

```
List of edges:
  First Graph    Second Graph
   from to         from to
     1   3           5   3
     1   4           5   2
     1   6           5   1
     2   3           4   3
     2   4           4   2
     2   5           4   6
     3   6           3   1
     4   5           2   6
     5   6           6   1


  Node mapping:
   First Graph   Second Graph
        1           5
        2           4
        3           3
        4           2
        5           6
        6           1
```