Centroid Decomposition

Priyanshu Das (AM.EN.H4CSE14003)

Amrita School of Engineering

# Introduction

Centroid Decomposition is a divide and conquer technique / Data Structure that aims to decompose a given tree (possibly skewed) into a tree which has at max log(N) height, where N is the number of nodes in the original tree. A major advantage of Centroid Decomposition, is that it drastically reduces the time complexity of point updates and point queries. This advantage arises due to the structure of the decomposed tree, which is more or less balanced, allowing us to store information regarding subtrees in the internal nodes. Information such as distance between a pair of nodes in the original tree can be computed, stored and accessed when needed to be used in conjunction with the decomposed tree using techniques described in this article.

# Centroid of a tree

Given a tree with N nodes, a node is called the centroid, if upon removing it, we get a forest of trees with each tree having a size (number of nodes) <= N/2.

**Theorem**: Given a tree with N nodes, there always exists a node C, such that upon removing this node, all the remaining components will have at max N/2 nodes.

**Proof:** If we select any node (c) in the tree, and if every subtree rooted at a neighbor of c, has size maximum N/2, then the selected node is the centroid and we are done. If this is not the case then one of the children subtrees have size > N/2.

If we select this child, then one of its subtrees (c) will definitely have size <= N/2, and hence we will never backtrack. Now, if this node is the Centroid, then we are done. If not, we repeat this process.

Since the number of nodes in the tree is finite and we never backtrack, this process must eventually end, and we would have a centroid.

# Centroid Tree

After we remove the centroid of the original tree, we get a forest, each with size <= N/2, each will have its own centroid. If we label the original centroid as the root, and the centroid of the resulting subtrees as it’s children, repeating the process recursively, we would have a tree with all N nodes, with all internal nodes (with the possible exception of the second last level) having >= 2 children.

IE we would have decomposed the original tree into another tree, with same number of nodes, but a maximum height of log2(N).

# Properties of Centroid Tree

* The tree formed will have all N nodes of the original tree.
* The height of the centroid tree is at most O(log(N)).
* The LCA of two nodes (a, b) in the centroid tree lies on the shortest path from (a, b) or (b, a) in the original tree.
* **The path between two nodes (p, q), can be reduced to path (p, lca(p,q) ) + path(q, lca(p, q)), in the decomposed tree.**

# Algorithm for Forming Centroid Tree

## Algorithm

In order to find the centroid of a given tree, we will use the theorem we just proved. We know that if we pick a node at random, and keep following the path whose subtree size is > n/2, we will eventually reach the centroid.

So, we pick a node at random, and follow the path with tree size greater than n/2, till we reach the centroid.

To form the centroid tree, we declare the found centroid to be the root (level 0) of the centroid tree, then we remove all incoming connections to this root, leaving behind a forest. For each of the trees in the forest, we find the centroid and declare them to be the children of the “parent” (the centroid which disconnected the trees in the forest) node in the centroid tree. We recursively repeat this process, till there are no more subtrees.

Step, by step:

1. Pick a node at random, find the centroid starting from that node
2. Once the first centroid is found, assign it as root of the centroid tree
3. Find the centroid for all the trees in the resulting forest, by starting from each of the children of the previously found centroid
4. assign these centroids as the children of the centroid that disconnected them
5. Repeat 3 – 5 for each centroid, till all nodes have been decomposed to the centroid tree

## Time Complexity

In the worst case, when we find the centroid for a tree, the resulting forest of trees will be having 2 trees, with sizes n/2 and n/2-1. This is the case if the tree in question is a linear tree.

For a linear tree, the worst starting point is either end of the linear tree. If we start at any end, the number of steps required to reach the centroid will be n/2.

Now, in the 1st level of the centroid tree (the level after the root), the subtrees represented by the centroids will have sizes n/2, let this be n’.

The number of steps required to reach the centroid in each will be n’/2 == n/4. Since there are 2 such subtrees, the steps taken = 2\*(N/4) = N/2.

We can see that this recursively repeats for k iterations, where 2^k = N, or k = log(N).

Thus, the worst case number of steps taken to decompose a tree is:

N/2 + N/2 + … + N/2 (k times) == O(N\*log(N))

## Pre-calculations Required

In order to efficiently perform queries on the centroid, we need to have a method of finding information, such as distance between two nodes in the original tree. In order to accomplish this, we need to have an efficient method of finding distance between two nodes in the original tree.

We know from the properties mentioned above, that the LCA of two nodes in the centroid lies on the shortest path between them. Thus, path from A->B can be decomposed to A->LCA(A, B) -> B, and vice versa. This relation is the keystone upon which the benefits of centroid decomposition rests.

We can’t store distance between every pair of nodes, as the memory required is too much (n^2). We instead use the following technique to find the distance between two nodes.

### Finding Distance between two nodes in the original tree

The distance between two nodes (u,v) can be found by the following relation:

Distance(u, v) == Level(u) + Level(v) - Level (lca(u,v))

where Level is precalculated as the level of the node in the original tree. Now, the problem comes with finding the lca(u, v). This can be done in log(N) steps, as described in the following subsection.

### Finding LCA of two nodes in the original tree

In order to reduce the time complexity of finding LCA In the original tree, we store the log(N) ancestors of every node.

For example: a node at level 20 will have stored the number of it’s parent, 2nd ancestor, 4th ancestor, 8th ancestor and 16th ancestor.

To find the Kth ancestor of node n, where K = 2^a + 2^b + 2^c +... (where a < b < c), it’s the same as finding the K’th ancestor of node n’, where n’ is the 2^a th ancestor of n, and K’ = 2^b + 2^c + …

Thus, finding the Kth ancestor of a node takes O(log(N)) time.

To find the LCA(a, b), we first bring a and b to the same level.

* let diff = level(b) – level (a) (level(a) < level(b))
* let a = parent(a, diff)

Then find the lowest level at which parent(level, a) == parent(level, b)

* Iterate for I = log(N) to I = 0
* if (par[I][a] != par[l][b]) a = par[I][a], b = par[I][b]
* After iteration ends, return par[I][a]

The above gives the lca, because if 2^kth ancestor of a and b are equal, and 2^k-1th ancestor are not equal, then the lca lies between 2^kth ancestor of a and 2^k-1 th ancestor of a, or 2^k-1 to 2^k-2 th ancestor of (2^k-1 th ancestor of a).

# Sample Problems

## Ciel The Commander (Codeforces)

Problem Link : <http://codeforces.com/problemset/problem/321/C>

### Problem Statement:

Now Fox Ciel becomes a commander of Tree Land. Tree Land, like its name said, has *n* cities connected by *n* - 1 undirected roads, and for any two cities there always exists a path between them.

Fox Ciel needs to assign an officer to each city. Each officer has a rank — a letter from 'A' to 'Z'. So there will be 26 different ranks, and 'A' is the topmost, so 'Z' is the bottommost.

There are enough officers of each rank. But there is a special rule must obey: if *x* and *y* are two distinct cities and their officers have the same rank, then on the simple path between *x* and *y* there must be a city *z* that has an officer with higher rank. The rule guarantee that a communications between same rank officers will be monitored by higher rank officer.

Help Ciel to make a valid plan, and if it's impossible, output "Impossible!"

**Input**

The first line contains an integer *n* (2 ≤ *n* ≤ 105) — the number of cities in Tree Land.

Each of the following *n* - 1 lines contains two integers *a* and *b* (1 ≤ *a*, *b* ≤ *n*, *a* ≠ *b*) — they mean that there will be an undirected road between *a* and *b*. Consider all the cities are numbered from 1 to *n*.

It guaranteed that the given graph will be a tree.

**Output**

If there is a valid plane, output *n* space-separated characters in a line — *i*-th character is the rank of officer in the city with number *i*.

Otherwise output "Impossible!"

### Problem Analysis

One key point to remember in this problem is that there can be maximum one node with rank ‘A’ in the tree, as according to the constraints, two cities with the same rank general must have a city lying on the path between them which has a general of higher rank. Since the highest rank is ‘A’, we cannot have more than one general of rank ‘A’.

If we consider the specific property of Centroid Tree, that the height of the tree is at max log(N), we get that height will be maximum 20. Now, consider this property of the centroid tree: Any path between two nodes at level k, has to pass through at least one node with level < k. Thus, we have the answer. As the height of the centroid tree will always be less than the number of characters available, we can simply decompose the tree and label all nodes at level k with ‘A’ + k, considering root to be level 0.

### Problem Solution

This problem requires a very basic application of Centroid Decomposition. We just need to decompose the tree, find the level of each node in the centroid tree, and label the nodes according to their levels.

### Code :

#include <bits/stdc++.h>

#define pf printf

#define sf scanf

#define ll long long int

#define mp make\_pair

#define pb push\_back

#define ins insert

#define loop(i, l, r) for(int i = l; i < r; i++)

#define max\_n 100001

#define max\_lg 20

#define inf\_val 9999999

using namespace std;

int lvl[max\_n+1], par[max\_n+1];

set<int> conn[max\_n+1];

int siz[max\_n+1];

vector<int> Primes;

bool not\_prime[max\_n+2];

ll n, pairs;

int t\_dist[max\_lg][max\_n+1];

//============================================================//

//-------------------------Decomposition----------------------//

int no\_nodes;

int find\_size(int node, int par)

{

siz[node] = 1;

no\_nodes++;

for(auto it = conn[node].begin(); it != conn[node].end(); it++)

if(\*it != par)

siz[node] += find\_size(\*it, node);

return siz[node];

}

int find\_centr(int node, int par)

{

for(auto it = conn[node].begin(); it != conn[node].end(); it++)

if(\*it != par && siz[\*it] > no\_nodes/2)

return find\_centr(\*it, node);

return node;

}

void decompose(int node, int depth)

{

ll ans = 0;

no\_nodes = 0;

find\_size(node, node);

int centr = find\_centr(node, node);

int u = centr;

par[u] = node;

lvl[u] = depth;

for(auto it=conn[u].begin(); it!=conn[u].end(); it++)

{

conn[\*it].erase(u);

decompose(\*it, depth+1);

}

conn[u].clear();

}

int main()

{

//--------------------------------------------------------------//

//----------------------- Input --------------------------------//

sf("%lld", &n);

loop(i, 0, n-1)

{

int a, b;

sf("%d%d", &a, &b);

conn[a].insert(b);

conn[b].insert(a);

}

decompose(1, 0);

int flag = 0;

for(int i = 1; i <= n; i++)

if(lvl[i]+'A' > 'Z')

flag = 1;

if(flag)

cout << "Impossible!";

else

{

for(int i = 1; i <= n; i++)

{

char tmp = lvl[i] + 'A';

pf("%c ", tmp);

}

}

return 0;

}

### Result: All test cases passed.



## Xenia and Tree (Codeforces)

Problem Link : <http://codeforces.com/contest/342/problem/E>

### Problem Statement:

Given N and M, where N is the number of nodes in the tree and M is the number of queries. Every node can be either “blue” or “red”. Initially every node is “blue”, except for vertex 1, which is coloured “red”.

N-1 lines follow describing the edges in the tree, the edge is described as two integers, a and b (separated by a space), meaning that there is an edge between a and b.

Then M queries follow. Each query has two numbers Ti and Vi. Ti is the type of the query, and Vi is the vertex.

* If Ti is 1, then color the vertex Vi as “red”.
* If Ti is 2, then print the minimum distance between Vi and a “red” node.

Constraints:

(2 <= N <= 105, 1<= M <= 105, 1<= ai,bi <=N, 1<= ti <=2, 1<= vi <= N)

### Problem Analysis

The most obvious and naïve approach to this problem would be to store all the “RED” nodes, and for a query, find the distance between that node and all the “RED” nodes. However, the time required to find the distance between two nodes in a tree is the same as finding their LCA : O(logN). Since the queries are Q, the complexity of this approach becomes O(Q2log(N)), as in case of alternating update and query, there will be O(Q2) distance queries.

Now, what we need to reduce is the number of distance calculation required each query. So, we try to organize the tree in such a way, that when we have a query, we only need to compare distances for a few select nodes. This is where Centroid Decomposition comes in.  
  
We decompose the tree, this is done in O(NlogN) operations. For every node in the centroid tree, we maintain its distance to the closest “RED” node in its subtree. Now, when a node is turned “RED”, the value stored at that node, as well as the value stored at its ancestors in the centroid tree may change. So, for every update, we need to update the value stored in that node and in it’s ancestors. This takes O(log2N) operations, as there are max logN ancestors and each distance comparison takes O(logN) time.

Next, for a query for a given node u, there can be three cases:-

1. Closest Node is in the subtree of u (in the Centroid Tree).
2. Closest Node is an ancestor of u (in the Centroid Tree).
3. Closest Node is the child of one of the ancestors of u, but not (1).

In case 1, the result will be the value stored at the node in the Centroid Tree.

In case 2, the answer will be the minimum of the distance from u to all of its ancestors (whose value is 0) in the given tree.

In case 3, the answer will be the minimum of the (dist(u, v) + val(v)) for all v ancestors of u.

In fact case 1, case 2 and case 3 can be combined, and the answer can be expressed as :-

Min( dist(u,v)+val(v) ) for all v==u or v == ancestor(u)

So, for a given query, we need to do maximum logN distance calculations, which means the number of operations is O(log2N).

Thus, the time complexity of the solution using Centroid Decomposition is O(Q\*log2N).

### Problem Solution

After we perform Centroid Decomposition of the given tree, the answer to query for any node will be the min (dist(node, ancestor) + ans(ancestor)), for all ancestors of the given node.

Whenever a node is updated, we check and update the answer all it’s O(logN) ancestors as well, ie ans[ancestor] = min(ans[ancestor], dist(node, ancestor)).

Distance between a node and it’s ancestor is calculated

Implementation:

Assume any node (say node 1) to be the root. Then, for every node, let’s find and store their level, marking the root (node 1) as level 0. Also, for every node we maintain a list of the node’s log(N) parents (Parent, 2nd ancestor, 4th ancestor,…, 2^log(N)th ancestor).

We can do this with a DFS, with DP. Pseudo Code:-

pre\_process()

DP[1][1] = 1; // DP[i][j] stores the 2^ith ancestor of j  
level[1] = 0;  
dfs\_pre(1);  
for(i=1; i < log(N); i++)  
 for(j = 1; j < N; j++)  
 DP[i][j] = DP[i-1][DP[i-1][j]];

dfs\_pre(int n)  
 for(x connected to n, x!= parent(x))  
 level[x] = level[n]+1;  
 DP[1][x] = n;  
 dfs\_pre(x);

Then we perform Centroid Decomposition of the given tree.

Now, we perform Centroid Decomposition of the tree.

As mentioned above, we perform the update for a node, by updating that node and it’s O(logn) ancestors, and for the queries, we calculate the min of all possible answers from the ancestors of the query node.

### Code :

#include <bits/stdc++.h>

#define sf scanf

#define pf printf

#define ll long long int

#define mp make\_pair

#define pb push\_back

#define max\_n 100000

#define max\_lg 20

#define loop(n) for(int i = 0; i < n; i++)

using namespace std;

typedef struct nod

{

set<int> conn;

bool color; // 0-BLUE, 1-RED

int size;

} nod;

typedef struct node

{

bool color;

int par;

int lvl, ans, num;

int dist[17];

} node;

node C\_tree[max\_n+1];

node \*root;

int n, m;

nod arr[max\_n+1];

int visited[max\_n+1], level[max\_n+1], DP[max\_lg+1][max\_n+1];

//----------------------------------------------------------//

//--------------\_Pre\_Process\_------------------------------//

//dfs\_pre(), finds and sets lvl of each node in original graph and DP[0][n's child]

void dfs\_pre(int n)

{

for(auto it = arr[n].conn.begin(); it != arr[n].conn.end(); it++)

{

if(\*it != DP[1][n])

{

DP[1][\*it] = n;

level[\*it] = level[n]+1;

dfs\_pre(\*it);

}

}

}

//Sets DP[logN][N]

void pre\_process()

{

DP[1][1] = 0;

level[1] = 0;

dfs\_pre(1);

for(int i = 2; i <= max\_lg; i++)

for(int j = 1; j <= max\_n; j++)

DP[i][j] = DP[i-1][DP[i-1][j]];

}

//lca() finds lca for two nodes in log(N) time

int lca(int a, int b) //a is lower level

{

if(level[a] > level[b]) swap(a, b);

int diff = level[b] - level[a];

for(int i = 1; i <= max\_lg; i++)

if(diff&(1<<(i-1)))

b = DP[i][b];

if(a==b)

return a;

for(int i = max\_lg; i > 0; i--)

if(DP[i][a] != DP[i][b])

{

a = DP[i][a];

b = DP[i][b];

}

return DP[1][a];

}

//returns distance between two nodes in the original tree

int distance(int u, int v)

{

return level[u] + level[v] - 2\*level[lca(u, v)];

}

//---------------------------------------------------------//

//--------------DECOMPOSITION-----------------------------//

int n\_nodes; //number of nodes

void dfs\_size(int n, int p) //find size of sub-tree at n, with parent p

{

arr[n].size = 1;

n\_nodes++;

for(auto it = arr[n].conn.begin(); it != arr[n].conn.end(); it++)

{

if(\*it != p)

{

dfs\_size(\*it, n);

arr[n].size += arr[\*it].size;

}

}

}

int dist;

int dfs\_centr(int n, int p) // finds centroid

{

dist++;

for(auto it = arr[n].conn.begin(); it != arr[n].conn.end(); it++)

{

if(\*it != p && arr[\*it].size > n\_nodes/2)

{

return dfs\_centr(\*it, n);

}

}

return n;

}

void decompose(int n, int p) // decomp. tree rooted at n, with par centroid p

{

n\_nodes = 0;

dist = 0;

dfs\_size(n,n);

int centroid = dfs\_centr(n,n);

memset(C\_tree[centroid].dist, 0, sizeof(C\_tree[centroid].dist));

if(p == -1)

{

p = centroid;

C\_tree[centroid].lvl = 0;

C\_tree[centroid].dist[C\_tree[centroid].lvl] = 0;

}

else

{

C\_tree[centroid].lvl = C\_tree[p].lvl+1;

C\_tree[centroid].dist[C\_tree[centroid].lvl] = 0;

for(int lv = C\_tree[p].lvl; lv >= 0; lv--)

{

C\_tree[centroid].dist[lv] = C\_tree[p].dist[lv] + dist;

}

}

C\_tree[centroid].par = p;

for(auto it = arr[centroid].conn.begin(); it != arr[centroid].conn.end(); it++)

{

arr[\*it].conn.erase(centroid);

decompose(\*it, centroid);

}

arr[n].conn.clear();

}

//------------------------------------------------------------//

//---------------\_UPDATE\_AND\_QUERY\_---------------------------//

void update(int n)

{

int tmp = n;

C\_tree[n].ans = 0;

while(1)

{

C\_tree[tmp].ans = min(C\_tree[tmp].ans, distance(n, tmp));

if(C\_tree[tmp].lvl == 0)

break;

tmp = C\_tree[tmp].par;

}

}

int query(int n)

{

int tmp = n;

int curr\_ans = C\_tree[n].ans;

while(1)

{

curr\_ans = min(curr\_ans, distance(n, tmp) + C\_tree[tmp].ans);

if(C\_tree[tmp].lvl == 0)

break;

tmp = C\_tree[tmp].par;

}

return curr\_ans;

}

//----------------- Main Function --------------------------------//\*/

int main()

{

memset(visited, 0, sizeof(visited));

sf("%d %d", &n, &m);

loop(n-1)

{

int n1, n2;

sf("%d %d", &n1, &n2);

arr[n1].conn.insert(n2);

arr[n1].color = 0; arr[n1].size = 0;

arr[n2].conn.insert(n1);

arr[n1].color = 0; arr[n1].size = 0;

}

loop(n)

{

C\_tree[i+1].num = i+1;

}

pre\_process();

decompose(1, -1);

loop(n)

C\_tree[i+1].ans = 99999999;

update(1);

loop(m)

{

int ti, vi;

sf("%d%d", &ti, &vi);

if(ti == 1)

update(vi);

else

pf("%d\n", query(vi));

}

return 0;

}

### Result: All test cases passed.



## Prime Distance on a Tree (Codechef)

Problem Link : <https://www.codechef.com/problems/PRIMEDST>

### Problem Statement:

You are given a tree. If we select 2 distinct nodes uniformly at random, what's the probability that the distance between these 2 nodes is a prime number?

**Input**

The first line contains a number **N**: the number of nodes in this tree.  
The following **N**-1 lines contain pairs ai and bi which means there is an edge with length 1 between them.

**Output**

Output a real number denote the probability we want.  
The difference between calculated answer and standard answer shouldn’t be more than 10-6.

**Constraints**

**(2** ≤ **N** ≤ **50,000), (1<= ai, bi <= N)**

### Problem Analysis

The problem can be reduced to: Find the number of unique pairs in the tree, such that the distance between them is a prime number. This is so because the probability of the distance between any two chosen nodes in the tree is:

**(pairs (u,v) such that dist(u,v) = P)/(NC2)**

where P is any Prime Number. Now, finding the distance between all possible pairs is not feasible, as NC2 is the number of possible ways of choosing pairs. This means that the number of operations required would be O(NC2\*log(N)), which is > (25\*1010) for N = 50,000.

However, we notice that we just need to find the number of pairs which have prime distance. So, first we find all prime numbers under N (O(NlogN)). The number of prime numbers under N is ~ N/(logN – 1). For N = 5\*104, this is ~3450.

Now, if we consider the property of centroid tree:

**“Every path (u,v) in the original tree can be broken down into a concatenation of two paths in the Centroid Tree : dist(u, v) == dist(u, lca(u, v)) + dist(v, lca(u,v))”**

For each centroid, we find the no of nodes at distance "i" from the centroid in its part and store it in dist[i].

We then remove the centroid and move to each part one by one which the centroid decomposes the tree into, and remove the contribution of all the nodes in that part in the dist array. For each node in that part, we find the no of nodes lying in other parts which are at prime distance, by iterating over all the primes and adding dist[Prime[j] - distance(i, centroid)] for each node i . Then again add the contributions of all the nodes in this part to dist and move on to the next part.

Since the contribution of each node to dist is added and removed only once, the overall complexity is O(N\*P\*logN) where P is the no of primes <=N .

### 

### Code :

#include <bits/stdc++.h>

#define pf printf

#define sf scanf

#define ll long long int

#define mp make\_pair

#define pb push\_back

#define ins insert

#define loop(i, l, r) for(int i = l; i < r; i++)

#define max\_n 50001

#define max\_lg 20

#define inf\_val 9999999

using namespace std;

int lvl[max\_n+1];

set<int> conn[max\_n+1];

int siz[max\_n+1];

vector<int> Primes;

bool not\_prime[max\_n+2];

ll n, pairs;

int t\_dist[max\_lg][max\_n+1];

//============================================================//

//-------------------------Decomposition----------------------//

int no\_nodes;

int find\_size(int node, int par)

{

siz[node] = 1;

no\_nodes++;

for(auto it = conn[node].begin(); it != conn[node].end(); it++)

if(\*it != par)

siz[node] += find\_size(\*it, node);

return siz[node];

}

int find\_centr(int node, int par)

{

for(auto it = conn[node].begin(); it != conn[node].end(); it++)

if(\*it != par && siz[\*it] > no\_nodes/2)

return find\_centr(\*it, node);

return node;

}

void add\_dist(int u, int p, int depth, int dist, int add)

{

t\_dist[depth][dist] += add;

for(auto it = conn[u].begin(); it != conn[u].end(); it++)

if(\*it != p)

add\_dist(\*it, u, depth, dist+1, add);

}

ll no\_pairs(int u, int p, int depth, int dist)

{

ll ans = 0;

for(int pr = 0; pr < Primes.size(); pr++)

{

int req = Primes[pr]-dist;

if(req <0) continue;

if(!t\_dist[depth][req]) break;

if(Primes[pr] != dist)

ans += t\_dist[depth][req];

else

ans += 2\*t\_dist[depth][req];

}

for(auto it = conn[u].begin(); it != conn[u].end(); it++)

if(\*it != p)

ans += no\_pairs(\*it, u, depth, dist+1);

return ans;

}

void decompose(int node, int depth)

{

ll ans = 0;

no\_nodes = 0;

find\_size(node, node);

int centr = find\_centr(node, node);

int u = centr;

add\_dist(centr, centr, depth, 0, 1);

for(auto it=conn[u].begin(); it!=conn[u].end(); it++)

{

add\_dist(\*it, u, depth, 1, -1);

ans += no\_pairs(\*it, u, depth, 1);

add\_dist(\*it, u, depth, 1, 1);

}

pairs += ans/2;

for(auto it=conn[u].begin(); it!=conn[u].end(); it++)

{

conn[\*it].erase(u);

decompose(\*it, depth+1);

}

for(int i = 0; i < n && t\_dist[depth][i]; i++)

t\_dist[depth][i] = 0;

}

//===================== Main Function ===========================//

//===============================================================//

int main()

{

//---------------------- Prime Sieve ---------------------------//

for(int i = 2; i <= max\_n; i++)

if(!not\_prime[i])

{

Primes.pb(i);

for(ll j = i\*i; j <= max\_n && j > 0; j += i)

not\_prime[j] = 1;

}

//--------------------------------------------------------------//

//----------------------- Input --------------------------------//

sf("%lld", &n);

loop(i, 0, n-1)

{

int a, b;

sf("%d%d", &a, &b);

conn[a].insert(b);

conn[b].insert(a);

}

decompose(1, 0);

double den = (double)n\*(n-1)\*0.5;

double ret = (double)pairs/den;

pf("%0.11lf", ret);

return 0;

}

//--------------------------------------------------------------//

### Result: All test cases passed

