too_lazy2name ICPC Team Notebook (2018-19)

Contents

1 Shortcuts

1.1 Template CPP

```cpp
#include <bits/stdc++.h>
using namespace std;
/* Template file for Online Algorithmic
   Competitions */
/* Typedefs */
    /* Basic types */
    typedef long long           ll;
    typedef long double         ld;
    typedef unsigned long long ull;
    /* STL containers */
    typedef vector <int>    vi;
    typedef vector <ll>     vll;
    typedef pair <int, int> pii;
    typedef pair <ll, ll>   pll;
    typedef vector < pii >  vpii;
    typedef vector < pll >  vpll;
    typedef vector <string> vs;
    typedef vector < vi >   vvi;
    typedef vector < vll >  vvll;
    typedef vector < vpii > vvpii;
    typedef set <int>       si;
/* Macros */
    /* Loops */
    #define fl(i, a, b)       for(int i(a); i
        <= (b); i ++)
    #define rep(i, n)         fl(i, 1, n)
    #define loop(i, n)        fl(i, 0, n - 1)
    #define rfl(i, a, b)      for(int i(a); i
        >= (b); i --)
    #define rrep(i, n)        rfl(i, n, 1)
    /* Algorithmic functions */
    #define srt(v)            sort((v).begin(),
         (v).end())
    /* STL container methods */
    #define pb                push_back
    #define mp                make_pair
    #define eb                emplace_back
    /* String methods */
    #define dig(i)            (s[i] - '0')
    #define slen(s)           s.length()
    /* Shorthand notations */
    #define fr                first
    #define sc                second
    #define re                return
    #define sz(x)             ((int) (x).size()
        )
    #define all(x)            (x).begin(), (x).
        end()
    #define sqr(x)            ((x) * (x))
    #define fill(x, y)        memset(x, y,
        sizeof(x))
    #define clr(a)            fill(a, 0)
    #define endl              '\n'
    /* Mathematical */
    #define IINF              0x3f3f3f3f
    #define LLINF
        1000111000111000111LL
```

```
    #define PI 3.14159265358979323
    /* Debugging purpose */
    #define trace1(x)                cerr << #x << ": " << x << endl
    #define trace2(x, y)             cerr << #x << ": " << x << " | " << #y << ": " << y << endl
    #define trace3(x, y, z)          cerr << #x << ": " << x << " | " << #y << ": " << y << " | " << #z << ": " << z << endl
    #define trace4(a, b, c, d)       cerr << #a << ": " << a << " | " << #b << ": " << b << " | " << #c << ": " << c << " | " << #d << ": " << d << endl
    #define trace5(a, b, c, d, e)    cerr << #a << ": " << a << " | " << #b << ": " << b << " | " << #c << ": " << c << " | " << #d << ": " << d << " | " << #e << ": " << e << endl
    #define trace6(a, b, c, d, e, f) cerr << #a << ": " << a << " | " << #b << ": " << b << " | " << #c << ": " << c << " | " << #d << ": " << d << " | " << #e << ": " << e << " | " << #f << ": " << f << endl
    /* Fast Input Output */
    #define FAST_IO                  ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0)
/* Constants */
    const ll MOD = 1000000007LL;
    const ll MAX = 100010LL;
/* Templates */
template<class T> T abs(T x) { re x > 0 ? x : -x; }
template<typename T> T gcd(T a, T b){ if(b == 0) return a; return gcd(b, a % b); }
template<typename T> T power(T x, T y, ll m = MOD){T ans = 1; x %= m; while(y > 0){ if(y & 1LL) ans = (ans * x)%m; y >>= 1LL; x = (x*x)%m; } return ans%m; }

int main(){
    #ifndef ONLINE_JUDGE
    freopen("/Users/sahilbansal/Desktop/input.txt","r",stdin);
    freopen("/Users/sahilbansal/Desktop/output.txt","w",stdout);
    freopen("/Users/sahilbansal/Desktop/error.txt","w",stderr);
    #endif
    FAST_IO;

    return 0;
}
```

2 Data Structures

2.1 Segment Tree

```
void build(int node, int start, int end)
{
    if(start == end)
    {
        // Leaf node will have a single element
        tree[node] = A[start];
    }
    else
    {
        int mid = (start + end) / 2;
        // Recurse on the left child
        build(2*node, start, mid);
        // Recurse on the right child
        build(2*node+1, mid+1, end);
        // Internal node will have the sum of both of its children
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}
void update(int node, int start, int end, int idx, int val)
{
    if(start == end)
    {
        // Leaf node
        A[idx] += val;
        tree[node] += val;
    }
    else
    {
        int mid = (start + end) / 2;
        if(start <= idx and idx <= mid)
```

```
        {
            // If idx is in the left child,
               recurse on the left child
            update(2*node, start, mid, idx,
               val);
        }
        else
        {
            // if idx is in the right child,
               recurse on the right child
            update(2*node+1, mid+1, end, idx,
                val);
        }
        // Internal node will have the sum of
           both of its children
        tree[node] = tree[2*node] + tree[2*
           node+1];
    }
}
int query(int node, int start, int end, int l
   , int r)
{
    if(r < start or end < l)
    {
        // range represented by a node is
           completely outside the given range
        return 0;
    }
    if(l <= start and end <= r)
    {
        // range represented by a node is
           completely inside the given range
        return tree[node];
    }
    // range represented by a node is
       partially inside and partially outside
        the given range
    int mid = (start + end) / 2;
    int p1 = query(2*node, start, mid, l, r);
    int p2 = query(2*node+1, mid+1, end, l, r
       );
    return (p1 + p2);
}
void updateRange(int node, int start, int end
   , int l, int r, int val)
{
    // out of range
    if (start > end or start > r or end < l)
        return;
    // Current node is a leaf node
    if (start == end)
    {
        // Add the difference to current node
        tree[node] += val;
        return;
    }
    // If not a leaf node, recur for children
       .
    int mid = (start + end) / 2;
    updateRange(node*2, start, mid, l, r, val
       );
    updateRange(node*2 + 1, mid + 1, end, l,
       r, val);
    // Use the result of children calls to
       update this node
    tree[node] = tree[node*2] + tree[node
       *2+1];
}
void updateRange(int node, int start, int end
   , int l, int r, int val)
{
    if(lazy[node] != 0)
    {
        // This node needs to be updated
        tree[node] += (end - start + 1) *
           lazy[node];    // Update it
        if(start != end)
        {
            lazy[node*2] += lazy[node];
                                  // Mark child
                as lazy
            lazy[node*2+1] += lazy[node];
                                // Mark child
               as lazy
        }
        lazy[node] = 0;
           // Reset it
    }
    if(start > end or start > r or end < l)
                    // Current segment is not
        within range [l, r]
        return;
    if(start >= l and end <= r)
    {
        // Segment is fully within range
```

```
        tree[node] += (end - start + 1) * val;
        if(start != end)
        {
            // Not leaf node
            lazy[node*2] += val;
            lazy[node*2+1] += val;
        }
        return;
    }
    int mid = (start + end) / 2;
    updateRange(node*2, start, mid, l, r, val);      // Updating left child
    updateRange(node*2 + 1, mid + 1, end, l, r, val);   // Updating right child
    tree[node] = tree[node*2] + tree[node*2+1];        // Updating root with max value
}
int queryRange(int node, int start, int end, int l, int r)
{
    if(start > end or start > r or end < l)
        return 0;           // Out of range
    if(lazy[node] != 0)
    {
        // This node needs to be updated
        tree[node] += (end - start + 1) * lazy[node];          // Update it
        if(start != end)
        {
            lazy[node*2] += lazy[node];           // Mark child as lazy
            lazy[node*2+1] += lazy[node];     // Mark child as lazy
        }
        lazy[node] = 0;                    // Reset it
    }
    if(start >= l and end <= r)      // Current segment is totally within range [l, r]
        return tree[node];
    int mid = (start + end) / 2;
    int p1 = queryRange(node*2, start, mid, l, r);         // Query left child
    int p2 = queryRange(node*2 + 1, mid + 1, end, l, r); // Query right child
    return (p1 + p2);
}
```

2.2 Fenwick Tree

```
class FenwickTree {                 // remember that index 0 is not used
private: vi ft; int n;            // recall that vi is: typedef vector<int> vi;
public: FenwickTree(int _n) : n(_n) { ft.assign(n+1, 0); }    // n+1 zeroes
  FenwickTree(const vi& f) : n(f.size()-1) {
    ft.assign(n+1, 0);
    for (int i = 1; i <= n; i++) {                           // O(n)
      ft[i] += f[i];
        // add this value
      if (i+LSOne(i) <= n)     // if index i has parent in the updating tree
        ft[i+LSOne(i)] += ft[i]; } }         // add this value to that parent
  int rsq(int j) {                                   // returns RSQ(1, j)
    int sum = 0; for (; j; j -= LSOne(j)) sum += ft[j];
    return sum; }
  int rsq(int i, int j) { return rsq(j) - rsq(i-1); }  // returns RSQ(i, j)
  // updates value of the i-th element by v (v can be +ve/inc or -ve/dec)
  void update(int i, int v) {
    for (; i <= n; i += LSOne(i)) ft[i] += v; }   // note: n = ft.size()-1
  int select(int k) { // O(log^2 n)
    int lo = 1, hi = n;
    for (int i = 0; i < 30; i++) { // 2^30 > 10^9 > usual Fenwick Tree size
      int mid = (lo+hi) / 2;                    // Binary Search the Answer
      (rsq(1, mid) < k) ? lo = mid : hi = mid; }
    return hi; }
};
```

```
class RUPQ : FenwickTree {      // RUPQ variant
    is a simple extension of PURQ
public:
  RUPQ(int n) : FenwickTree(n) {}
  int point_query(int i) { return rsq(i); }
  void range_update(int i, int j, int v) {
     update(i, v), update(j+1, -v); }
};
```

2.3 Union Find

```
class UnionFind {
   // OOP style
private:
  vi p, rank, setSize;
     // remember: vi is vector<int>
  int numSets;
public:
  UnionFind(int N) {
    setSize.assign(N, 1); numSets = N; rank.
       assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i
       ++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i
      : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return
     findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) { numSets--;
      int x = findSet(i), y = findSet(j);
      // rank is used to keep the tree short
      if (rank[x] > rank[y]) { p[y] = x;
         setSize[x] += setSize[y]; }
      else                   { p[x] = y;
         setSize[y] += setSize[x];
                                 if (rank[x] ==
                                    rank[y])
                                   rank[y]++;
                                   } } }
  int numDisjointSets() { return numSets; }
  int sizeOfSet(int i) { return setSize[
     findSet(i)]; }
};
```

3 Bit Manipulation

3.1 Bit Manipulation

```
#define isOn(S, j) (S & (1<<j))
#define setBit(S, j) (S |= (1<<j))
#define clearBit(S, j) (S &= ~(1<<j))
#define toggleBit(S, j) (S ^= (1<<j))
#define lowBit(S) (S & (-S))
#define setAll(S, n) (S = (1<<n)-1)

#define modulo(S, N) ((S) & (N-1))    //
   returns S % N, where N is a power of 2
#define isPowerOfTwo(S) (!(S & (S-1)))
#define nearestPowerOfTwo(S) ((int)pow(2.0, (
   int)((log((double)S) / log(2.0)) + 0.5)))
#define turnOffLastBit(S) ((S) & (S-1))
#define turnOnLastZero(S) ((S) | (S+1))
#define turnOffLastConsecutiveBits(S) ((S) &
   (S+1))
#define turnOnLastConsecutiveZeroes(S) ((S) |
    (S-1))
```

4 Graph Algorithms

4.1 Shortest Path

```
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1e9
int main() {
    int V, E, s; scanf("%d %d %d", &V, &E, &s
       );
    vector<vii> AL(V, vii());       // assign
       blank vectors of ii-s to AL
    for (int i = 0; i < E; i++) {
    int u, v, w; scanf("%d %d %d", &u, &v, &w
       );
    AL[u].emplace_back(v, w);
                                  // directed
       graph
    }
    // Dijkstra routine
```

```
vi dist(V, INF); dist[s] = 0;            // INF = 1B to avoid overflow
priority_queue<ii, vector<ii>, greater<ii>> pq; pq.push({0, s});
                              // to sort the^ pairs by increasing distance from s
while (!pq.empty()) {
  // main loop
int d, u; tie(d, u) = pq.top(); pq.pop();
    // get shortest unvisited u
if (d > dist[u]) continue;          // this is a very important check
for (auto &v : AL[u]) {
                           // all outgoing edges from u
  if (dist[u]+v.second < dist[v.first]) {
    dist[v.first] = dist[u]+v.second;
                           // relax operation
    pq.push({dist[v.first], v.first});
} } }      // this variant can cause duplicate items in the priority queue
for (int i = 0; i < V; i++) // index + 1 for final answer
printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

// Bellman Ford routine
vi dist(V, INF); dist[s] = 0;
for (int i = 0; i < V-1; i++) // relax all E edges V-1 times, total O(VE)
  for (int u = 0; u < V; u++)
                             // these two loops = O(E)
    if (dist[u] != INF)  // important: do not propagate if dist[u] == INF
      for (auto &v : AL[u])   // we can record SP spanning here if needed
        dist[v.first] = min(dist[v.first], dist[u]+v.second);    // relax
bool hasNegativeCycle = false;
for (int u = 0; u < V; u++) if (dist[u] != INF) // one more pass to check
  for (auto &v : AL[u])
    if (dist[v.first] > dist[u]+v.second)
                                  // should be false
      hasNegativeCycle = true;  // if true, then negative cycle exists!
  printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");
  if (!hasNegativeCycle)
    for (int i = 0; i < V; i++)
      printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);
  return 0;
}
```

4.2 Warshall

```
int V, E; scanf("%d %d", &V, &E);
for (int i = 0; i < V; i++) {
  for (int j = 0; j < V; j++)
    AM[i][j] = INF;
  AM[i][i] = 0;
}
for (int i = 0; i < E; i++) {
  int u, v, w; scanf("%d %d %d", &u, &v, &w);
  AM[u][v] = w; // directed graph
}
for (int k = 0; k < V; k++) // common error: remember that loop order is k->i->j
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      AM[i][j] = min(AM[i][j], AM[i][k]+AM[k][j]);
```

4.3 Matching

```
vi match, vis;
  // global variables
vector<vi> AL;
int Aug(int L) {      // return 1 if there exists an augmenting path from L
  if (vis[L]) return 0;
                                       // return 0 otherwise
  vis[L] = 1;
  for (auto &R : AL[L])
```

```
    if (match[R] == -1 || Aug(match[R])) {
      match[R] = L;
      return 1;
        // found 1 matching
    }
  return 0;
    // no matching
}
bool isprime(int v) {
  int primes[10] =
    {2,3,5,7,11,13,17,19,23,29};
  for (int i = 0; i < 10; i++)
    if (primes[i] == v)
      return true;
  return false;
}
int main() {
  int V = 5, Vleft = 3;                                          // we
    ignore vertex 0
  AL.assign(V, vi());
  AL[1].push_back(3); AL[1].push_back(4);
  AL[2].push_back(3);
  // build unweighted bipartite graph with
    directed edge left->right set
  unordered_set<int> freeV;
  for (int L = 0; L < Vleft; L++)
    freeV.insert(L); // assume all vertices
      on left set are free initially
  match.assign(V, -1);    // V is the number
    of vertices in bipartite graph
  int MCBM = 0;
  // Greedy pre-processing for trivial
    Augmenting Paths
  // try commenting versus un-commenting this
     for-loop
  for (int L = 0; L < Vleft; L++) {
                                          // O(V^2)
    vi candidates;
    for (auto &R : AL[L])
      if (match[R] == -1)
        candidates.push_back(R);
    if (candidates.size() > 0) {
      MCBM++;
      freeV.erase(L);                     // L is
        matched, no longer a free vertex
      int a = rand()%candidates.size();   //
        randomize this greedy matching
      match[candidates[a]] = L;
    }
  }
  for (auto &f : freeV) {       // for each of
      the k remaining free vertices
    vis.assign(Vleft, 0);
      // reset before each recursion
    MCBM += Aug(f);           // once f is
      matched, f remains matched till end
  }
  printf("Found %d matchings\n", MCBM);
  return 0;
}
```

4.4 Max Flow

```
#define MAX_V 100 // enough for sample graph
   in Figure 4.24/4.25/4.26/UVa 259
int V, k, vertex, weight;
int res[MAX_V][MAX_V], mf, f, s, t;
                        // global variables
vector<vii> AL;                  // res and
   AdjList contain the same flow graph
vi p;
void augment(int v, int minEdge) {  //
   traverse BFS spanning tree from s->t
  if (v == s) { f = minEdge; return; }  //
     record minEdge in a global var f
  else if (p[v] != -1) { augment(p[v], min(
     minEdge, res[p[v]][v]));
                             res[p[v]][v] -= f;
                               res[v][p[v]] += f
                               ; } }
int main() {
  scanf("%d %d %d", &V, &s, &t);
  memset(res, 0, sizeof res);
  AL.assign(V, vii());
  for (int u = 0; u < V; u++) {
    int k; scanf("%d", &k);
    while (k--) {
      int v, w; scanf("%d %d", &v, &w);
      res[u][v] = w;
      AL[u].emplace_back(v, 1);
                                 // to record
         structure
      AL[v].emplace_back(u, 1);
         // do not forget the back edge
    }
```

```cpp
  }
  mf = 0;
     // mf stands for max_flow
  while (1) {                          // an O(
     VE^2) Edmonds Karp's algorithm
    f = 0;
    // run BFS, compare with the original BFS
       shown in Section 4.2.2
    bitset<MAX_V> vis; vis[s] = true;
       // we change vi dist to bitset!
    queue<int> q; q.push(s);
    p.assign(MAX_V, -1);       // record the
       BFS spanning tree, from s to t!
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break; // immediately stop
         BFS if we already reach sink t
      for (auto v : AL[u])              //
         use AL for neighbor enumeration
        if (res[u][v.first] > 0 && !vis[v.
           first])
          vis[v.first] = true, q.push(v.first
             ), p[v.first] = u;
    }
    augment(t, INF);    // find the min edge
       weight 'f' in this path, if any
    if (f == 0) break; // we cannot send any
       more flow ('f' = 0), terminate
    mf += f; }           // we can still send
       a flow, increase the max flow!
  printf("%d\n", mf);
                              // this is the
     max flow value
  return 0;
}
```

4.5 Strongly Connected Components

```cpp
//Implementation of Strongly connected
   components using Kosaraju Algorithm

const int MAX = 2e5 + 5;

//Complexity : O(V + E)
class StronglyConnected {
private:
        int V, E, cnt;
        stack<int> S;
        bool visited[MAX];
        vector<int> adj[MAX];
        vector<int> trans[MAX];
        vector<int> components[MAX];
public:
        StronglyConnected(int n, int m) {
                V = n;
                E = m;
                cnt = 0;
        }
        void clear() {
                for(int i=1; i<=V; ++i) {
                        adj[i].clear();
                        trans[i].clear();
                        components[i].clear()
                           ;
                }
        }
        void set_visited() {
                for(int i=1; i<=V; ++i) {
                        visited[i] = false;
                }
        }
        void add_edge(int a, int b) {
                adj[a].push_back(b);
                trans[b].push_back(a);
        }
        void dfs1(int u) {
                visited[u] = true;
                for(size_t i=0; i<adj[u].size
                   (); ++i) {
                        if (visited[adj[u][i
                           ]] == false) {
                                dfs1(adj[u][i
                                   ]);
                        }
                }
                S.push(u);
        }
        void dfs2(int u) {
                visited[u] = true;
                components[cnt].push_back(u);
                for(size_t i=0; i<trans[u].
                   size(); ++i) {
                        if(visited[trans[u][i
                           ]] == false) {
                                dfs2(trans[u
                                   ][i]);
                        }
                }
```

```
        }
        void scc() {
                set_visited();
                for(int i=1; i<=V; ++i) {
                        if(!visited[i]) {
                                dfs1(i);
                        }
                }
                set_visited();
                cnt = 0;
                while(!S.empty()) {
                        int v = S.top();
                        S.pop();
                        if (visited[v] ==
                           false) {
                                dfs2(v);
                                cnt += 1;
                        }
                }
        }
        bool is_scc() {
                return (cnt == 1);
        }
        int no_of_scc() {
                return cnt;
        }
        void print() {
                for(int i=0; i<cnt; ++i) {
                        printf("Component %d
                           : ", i+1);
                        for(size_t j=0; j<
                           components[i].size
                           (); ++j) {
                                printf("%d ",

                                   components
                                   [i][j]);
                        }
                        printf("\n");
                }
        }
};
```

5 Number Theory

5.1 Number Theory

```
typedef map<int, int> mii;
```

```
ll _sieve_size;
bitset<10000010> bs;                        //
   10^7 should be enough for most cases
vll primes;                    // compact list of
   primes in form of vector<long long>
// first part
void sieve(ll upperbound) {               //
   create list of primes in [0..upperbound]
  _sieve_size = upperbound+1;
                              // add 1 to include
     upperbound
  bs.set();

     // set all bits to 1
  bs[0] = bs[1] = 0;
                                            //
     except index 0 and 1
  for (ll i = 2; i < _sieve_size; i++) if (bs
     [i]) {
    // cross out multiples of i <=
       _sieve_size starting from i*i
    for (ll j = i*i; j < _sieve_size; j += i)
        bs[j] = 0;
    primes.push_back(i);         // also add
       this vector containing list of primes
} }

   // call this method in main method

bool isPrime(ll N) {                      // a
   good enough deterministic prime tester
  if (N < _sieve_size) return bs[N];
                         // now O(1) for small
     primes
  for (int i = 0; (i < primes.size()) && (
     primes[i]*primes[i] <= N); i++)
    if (N%primes[i] == 0) return false;
  return true;                      // it takes
      longer time if N is a large prime!
}                              // note: only work for
    N <= (last prime in vi "primes")^2

vi primeFactors(ll N) {    // remember: vi is
   vector of integers, ll is long long
  vi factors;                        // vi `
     primes' (generated by sieve) is optional
  ll PF_idx = 0, PF = primes[PF_idx];      //
     using PF = 2, 3, 4, ..., is also ok
  while ((N != 1) && (PF*PF <= N)) {    //
     stop at sqrt(N), but N can get smaller
```

```
    while (N%PF == 0) { N /= PF; factors.push_back(PF); }      // remove this PF
    PF = primes[++PF_idx];                                     // only consider primes!
  }
  if (N != 1) factors.push_back(N);      // special case if N is actually a prime
  return factors;             // if pf exceeds 32-bit integer, you have to change vi
}
ll numPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF*PF <= N)) {
    while (N%PF == 0) { N /= PF; ans++; }
    PF = primes[++PF_idx];
  }
  return ans + (N != 1);
}
ll numDiffPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF*PF <= N)) {
    if (N%PF == 0) ans++;                                      // count this pf only once
    while (N%PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  return ans + (N != 1);
}
ll sumPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF*PF <= N)) {
    while (N%PF == 0) { N /= PF; ans += PF; }
    PF = primes[++PF_idx];
  }
  return ans + (N != 1) * N;
}
ll numDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;         // start from ans = 1
  while (N != 1 && (PF*PF <= N)) {
    ll power = 0;                                        // count the power
    while (N%PF == 0) { N /= PF; power++; }
    ans *= (power+1);                                    // according to the formula
    PF = primes[++PF_idx];
  }
  return (N != 1) ? 2*ans : ans;    // (last factor has pow = 1, we add 1 to it)
}
ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;         // start from ans = 1
  while (N != 1 && (PF*PF <= N)) {
    ll power = 0;
    while (N%PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power+1.0) - 1) / (PF-1);    // formula
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N-1);  // last one
  return ans;
}
ll EulerPhi(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = N;         // start from ans = N
  while (N != 1 && (PF * PF <= N)) {
    if (N % PF == 0) ans -= ans / PF;                    // only count unique factor
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  return (N != 1) ? ans - ans/N : ans;                   // last factor
}
```

5.2 Extended Euclidean Function

```
int gcd (int a, int b, int & x, int & y) {
        if (a == 0) {
                x = 0; y = 1;
                return b;
        }
        int x1, y1;
```

```
        int d = gcd (b%a, a, x1, y1);
        x = y1 - (b / a) * x1;
        y = x1;
        return d;
}
```

5.3 Modular Inverse

```
// Function to find modular inverse of a
   under modulo m
// Assumption: m is prime
void modInverse(int a, int m)
{
    int g = gcd(a, m);
    if (g != 1)
        cout << "Inverse doesn't exist";
    else
    {
        // If a and m are relatively prime,
           then modulo inverse
        // is a^(m-2) mode m
        cout << "Modular multiplicative
           inverse is "
             << power(a, m-2, m);
    }
}
// Function to find modulo inverse of a
// Works when m and a are coprime
void modInverse(int a, int m)
{
    int x, y;
    int g = gcdExtended(a, m, &x, &y);
    if (g != 1)
        cout << "Inverse doesn't exist";
    else
    {
        // m is added to handle negative x
        int res = (x%m + m) % m;
        cout << "Modular multiplicative
           inverse is " << res;
    }
}
// A naive method to find modulor
   multiplicative inverse of
// 'a' under modulo 'm'
int modInverse(int a, int m)
{
    a = a%m;
    for (int x=1; x<m; x++)
       if ((a*x) % m == 1)
          return x;
}
```

5.4 nCr Modulo P

```
// Returns n^(-1) mod p (used Fermat's little
    theorem)
ll modInverse(ll n, ll p){
    return power(n, p-2, p);
}
// Returns nCr % p using Fermat's little
   theorem.
ll nCrModP(ll n, ll r, ll p){
   // Base case
   if(r == 0)
      return 1;
    // Fill factorial array so that we can
       find all factorial of r, n and n - r
    ll fact[n + 1];
    fact[0] = 1;
    fl(i, 1, n + 1){
        fact[i] = (fact[i - 1] * i) % p;
    }
    return (fact[n] * modInverse(fact[r], p)
       % p * modInverse(fact[n - r], p) % p)
       % p;
}
```

6 String Matching

6.1 KMP

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N]; // T = text, P =
   pattern
int b[MAX_N], n, m; // b = back table, n =
   length of T, m = length of P
void naiveMatching() {
  for (int i = 0; i < n; i++) { // try all
     potential starting indices
    bool found = true;
    for (int j = 0; j < m && found; j++) //
       use boolean flag 'found'
```

```
      if (i+j >= n || P[j] != T[i+j]) // if mismatch found
        found = false; // abort this, shift starting index i by +1
    if (found) // if P[0..m-1] == T[i..i+m-1]
      printf("P is found at index %d in T\n", i);
} }
void kmpPreprocess() { // call this before calling kmpSearch()
  int i = 0, j = -1; b[0] = -1; // starting values
  while (i < m) { // pre-process the pattern string P
    while (j >= 0 && P[i] != P[j]) j = b[j]; // if different, reset j using b
    i++; j++; // if same, advance both pointers
    b[i] = j; // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2, 3, 4
} }         // in the example of P = "SEVENTY SEVEN" above
void kmpSearch() { // this is similar as kmpPreprocess(), but on string T
  int i = 0, j = 0; // starting values
  while (i < n) { // search through string T
    while (j >= 0 && T[i] != P[j]) j = b[j]; // if different, reset j using b
    i++; j++; // if same, advance both pointers
    if (j == m) { // a match found when j == m
      printf("P is found at index %d in T\n", i-j);
      j = b[j]; // prepare j for the next possible match
} } }
```

6.2 Z-Algorithm

```
//The Z-function for this string is an array of length n where the i-th element is equal to the greatest number
//of characters starting from the position i that coincide with the first characters of s.
vector<int> z_function(string &s)
{
        int n=s.size();
        vector<int> z(n);
        for(int i=1,l=0,r=0;i<n;i++)
        {
                if(i<=r)
                        z[i]=min(r-i+1, z[i-l]);
                while(i+z[i]<n && s[z[i]]==s[i+z[i]])
                        z[i]++;
                if(i+z[i]-1>r)
                        l=i, r=i+z[i]-1;
        }
        return z;
}
```

6.3 Trie

```
//Trie implementation for finding xor maximisation & minimisation
const int MAX = 1<<20;
const int LN = 20;

struct node {
        node *child[2];
};
static node trie_alloc[MAX*LN] = {};
static int trie_sz = 0;

node *trie;

node *get_node() {
        node *temp = trie_alloc + (trie_sz++);
        temp->child[0] = NULL;
        temp->child[1] = NULL;
        return temp;
}
//O(log A_MAX)
void insert(node *root, int n) {
        for(int i = LN-1; i >= 0; --i) {
                int x = (n&(1<<i)) ? 1 : 0;
                if (root->child[x] == NULL) {
                        root->child[x] = get_node();
                }
                root = root->child[x];
        }
}
```

```
//O(log A_MAX)
int query_min(node *root, int n) {
        int ans = 0;
        for(int i = LN-1; i >= 0; --i) {
                int x = (n&(1<<i)) ? 1 : 0;
                assert(root != NULL);
                if (root->child[x] != NULL) {
                        root = root->child[x];
                }
                else {
                        ans ^= (1<<i);
                        root = root->child[1^x];
                }
        }
        return ans;
}
//O(log A_MAX)
int query_max(node *root, int n) {
        int ans = 0;
        for(int i = LN-1; i >= 0; --i) {
                int x = (n&(1<<i)) ? 1 : 0;
                assert(root != NULL);
                if (root->child[1^x] != NULL) {
                        ans ^= (1<<i);
                        root = root->child[1^x];
                }
                else {
                        root = root->child[x];
                }
        }
        return ans;
}
```

7 LCA and LCS

7.1 LCA

```
int depth[maxn],s[maxn],table[maxn][20] = {0};
vi graph[maxn];
pii edges[maxn];
void dfs1(int x) {
        loop(i,graph[x].size()) {
                if(graph[x][i]!=table[x][0]) {
                        depth[graph[x][i]] = depth[x] + 1;
                        table[graph[x][i]][0] = x;
                        dfs1(graph[x][i]);
                }
        }
}
void build_table(int n) {
        rep(i,19) {
                rep(j,n) {
                        table[j][i] = table[table[j][i-1]][i-1];
                }
        }
}
int lca(int x,int y) {
        if(depth[x]>depth[y]) swap(x,y);
        for(int i=19;~i;i--) {
                if(depth[table[y][i]]>=depth[x]) y = table[y][i];
        }
        //cout<<y<<endl;
        if(x==y) return x;
        for(int i=19;~i;i--) {
                if(table[x][i]!=table[y][i]) {
                        x = table[x][i];
                        y = table[y][i];
                }
        }
        return table[x][0];
}
void dfs2(int x) {
        loop(i,graph[x].size()) {
                if(graph[x][i]!=table[x][0]) dfs2(graph[x][i]),s[x]+=s[graph[x][i]];
        }
}
int main() {
        int n;
        cin>>n;
```

```
        rep(i,n-1) {
                int x,y;
                cin>>x>>y;
                graph[x].pb(y);
                graph[y].pb(x);
                edges[i] = {x,y};
        }
        dfs1(1);
        build_table(n);
        int m;
        cin>>m;
        loop(i,m) {
                int x,y;
                cin>>x>>y;
                s[x]++;
                s[y]++;
                s[lca(x,y)]-=2;
        }
        dfs2(1);
        rep(i,n-1) {
                if(depth[edges[i].fr]>depth[
                   edges[i].sc])
                {
                        cout<<s[edges[i].fr
                           ]<<' ';
                }
                else cout<<s[edges[i].sc]<<'
                   ';
        }
        cout<<endl;
        return 0;
}
```

7.2 LCS

```
// Given a list of numbers of length n, this
   routine extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
//   INPUT: a vector of integers
//   OUTPUT: a vector containing the longest
   increasing subsequence

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

#define STRICTLY_INCREASNG

VI LongestIncreasingSubsequence(VI v) {
  VPII best;
  VI dad(v.size(), -1);

  for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
    PII item = make_pair(v[i], 0);
    VPII::iterator it = lower_bound(best.
       begin(), best.end(), item);
    item.second = i;
#else
    PII item = make_pair(v[i], i);
    VPII::iterator it = upper_bound(best.
       begin(), best.end(), item);
#endif
    if (it == best.end()) {
      dad[i] = (best.size() == 0 ? -1 : best.
         back().second);
      best.push_back(item);
    } else {
      dad[i] = it == best.begin() ? -1 : prev
         (it)->second;
      *it = item;
    }
  }

  VI ret;
  for (int i = best.back().second; i >= 0; i
     = dad[i])
    ret.push_back(v[i]);
  reverse(ret.begin(), ret.end());
  return ret;
}
```