too_lazy2name ICPC Team Notebook (2018-19)

Contents

1 Shortcuts

1.1 Template CPP

```
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef unsigned long long ull;

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;

#define fl(i, a, b) for (int i(a); i <= (b); i ++)
#define rep(i, n)                for (int i = 1; i <= (n); i ++)
#define loop(i, n)               for (int i = 0; i < (n); i ++)
#define rfl(i, a, b)    for (int i(a); i >= (b); i --)
#define rrep(i, n)               for (int i = (n); i >= 1; i --)

#define all(v)  (v).begin(), (v).end()
#define srt(v)  sort (all (v))

#define pb push_back
#define mp make_pair

#define dig(i)  (s[i] - '0')
#define slen(s) s.length()

#define fr first
#define sc second

#define sz(x) ((int) (x).size())
#define fill (x, y)     memset(x, y, sizeof(x))
#define clr(a)  fill(a, 0)
#define endl '\n'

#define PI 3.14159265358979323

#define trace1(x1)
                cerr << #x1 << ": " << x1 << endl;
#define trace2(x1, x2)
            cerr << #x1 << ": " << x1 << " | " << #x2 << ": " << x2 << endl;
#define trace3(x1, x2, x3)
            cerr << #x1 << ": " << x1 << " | " << #x2 << ": " << x2 << " | " << #x3 << "
```

```
    : " << x3 << endl;
#define FAST_IO ios_base::sync_with_stdio(
    false); cin.tie(0); cout.tie(0)
const ll MOD = 1000000007LL;
const ll MAX = 100010LL;
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("input.txt", "r", stdin);
        freopen("output.txt", "w", stdout);
        freopen("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
    }
  }
  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

```
//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.
void getZarr(string str, int Z[]);

// prints all occurrences of pattern in text
   using Z algo
void search(string text, string pattern)
{
    // Create concatenated string "P$T"
    string concat = pattern + "$" + text;
    int l = concat.length();

    // Construct Z array
    int Z[l];
    getZarr(concat, Z);

    // now looping through Z array for
       matching condition
    for (int i = 0; i < l; ++i)
    {
        // if Z[i] (matched region) is equal
           to pattern
        // length we got the pattern
        if (Z[i] == pattern.length())
            cout << "Pattern found at index "
                << i - pattern.length() -1 <<
                    endl;
    }
}
// Fills Z array for given string str[]
void getZarr(string str, int Z[])
{
    int n = str.length();
    int L, R, k;

    // [L,R] make a window which matches with
       prefix of s
    L = R = 0;
    for (int i = 1; i < n; ++i)
    {
        // if i>R nothing matches so we will
           calculate.
        // Z[i] using naive way.
        if (i > R)
        {
            L = R = i;

            // R-L = 0 in starting, so it
               will start
            // checking from 0'th index. For
               example,
            // for "ababab" and i = 1, the
               value of R
            // remains 0 and Z[i] becomes 0.
               For string
            // "aaaaaa" and i = 1, Z[i] and R
                become 5
            while (R<n && str[R-L] == str[R])
                R++;
            Z[i] = R-L;
            R--;
        }
        else
        {
            // k = i-L so k corresponds to
               number which
            // matches in [L,R] interval.
            k = i-L;

            // if Z[k] is less than remaining
                interval
            // then Z[i] will be equal to Z[k
               ].
            // For example, str = "ababab", i
                = 3, R = 5
            // and L = 2
            if (Z[k] < R-i+1)
                Z[i] = Z[k];

            // For example str = "aaaaaa" and
                i = 2, R is 5,
            // L is 0
            else
            {
                // else start from R and
                   check manually
                L = i;
```

```
                while (R<n && str[R-L] == str
                    [R])
                    R++;
                Z[i] = R-L;
                R--;
            }
        }
    }
}
```

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 LIS

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 LIS

```
// 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;
}
```

8 Computational Geometry

8.1 Points and Lines

```
#include <bits/stdc++.h>
using namespace std;

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0) // important constant;
   alternative #define PI (2.0 * acos(0.0))

double DEG_to_RAD(double d) { return d*PI /
   180.0; }

double RAD_to_DEG(double r) { return r*180.0
   / PI; }

// struct point_i { int x, y; };    // basic
   raw form, minimalist mode
struct point_i { int x, y;     // whenever
   possible, work with point_i
  point_i() { x = y = 0; }
                          // default
     constructor
  point_i(int _x, int _y) : x(_x), y(_y) {}
     };          // user-defined

struct point { double x, y;   // only used if
    more precision is needed
  point() { x = y = 0.0; }
                          // default
     constructor
  point(double _x, double _y) : x(_x), y(_y)
     {}         // user-defined
  bool operator < (point other) const { //
     override less than operator
    if (fabs(x-other.x) > EPS)
                         // useful for
       sorting
      return x < other.x;          // first
         criteria , by x-coordinate
    return y < other.y; }          // second
       criteria, by y-coordinate
  // use EPS (1e-9) when testing equality of
     two floating points
  bool operator == (point other) const {
   return (fabs(x-other.x) < EPS && (fabs(y-
      other.y) < EPS)); } };

double dist(point p1, point p2) {
                  // Euclidean distance
                      // hypot(dx, dy)
                         returns sqrt(dx * dx
                          + dy * dy)
  return hypot(p1.x-p2.x, p1.y-p2.y); }
                    // return double

// rotate p by theta degrees CCW w.r.t origin
   (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);    //
     multiply theta with PI / 180.0
  return point(p.x * cos(rad) - p.y*sin(rad),
               p.x * sin(rad) + p.y*cos(rad))
```

```
                    ; }
struct line { double a, b, c; };          //
    a way to represent a line

// the answer is stored in the third
    parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l
    ) {
  if (fabs(p1.x-p2.x) < EPS)
                        // vertical line is
      fine
    l = {1.0, 0.0, -p1.x};
                                  // default
       values
  else
    l = {-(double)(p1.y-p2.y) / (p1.x-p2.x),
         1.0,                 // IMPORTANT: we
            fix the value of b to 1.0
         -(double)(l.a*p1.x) - p1.y}; }

// not needed since we will use the more
    robust form: ax + by + c = 0
struct line2 { double m, c; };      //
    another way to represent a line

int pointsToLine2(point p1, point p2, line2 &
    l) {
 if (abs(p1.x-p2.x) < EPS) {            //
     special case: vertical line
   l.m = INF;                       // l
      contains m = INF and c = x_value
   l.c = p1.x;                      // to denote
      vertical line x = x_value
   return 0;   // we need this return
      variable to differentiate result
 }
 else {
   l.m = (double)(p1.y-p2.y) / (p1.x-p2.x);
   l.c = p1.y - l.m*p1.x;
   return 1;     // l contains m and c of the
      line equation y = mx + c
} }
bool areParallel(line l1, line l2) {       //
    check coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.
     b-l2.b) < EPS); }

bool areSame(line l1, line l2) {           //
    also check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c-l2
     .c) < EPS); }

// returns true (+ intersection point) if two
    lines are intersect
bool areIntersect(line l1, line l2, point &p)
    {
  if (areParallel(l1, l2)) return false;
                  // no intersection
  // solve system of 2 linear algebraic
     equations with 2 unknowns
  p.x = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b
     - l1.a*l2.b);
  // special case: test for vertical line to
     avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a*p.x + l1
     .c);
  else                      p.y = -(l2.a*p.x + l2
     .c);
  return true; }

struct vec { double x, y;  // name: 'vec' is
    different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {}
      };

vec toVec(point a, point b) {          //
    convert 2 points to vector a->b
  return vec(b.x-a.x, b.y-a.y); }

vec scale(vec v, double s) {           //
    nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x*s, v.y*s); }
                        // shorter.same.longer

point translate(point p, vec v) {          //
    translate p according to v
  return point(p.x+v.x, p.y+v.y); }

// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line
    &l) {
  l.a = -m;
     // always -m
  l.b = 1;
     // always 1
  l.c = -((l.a*p.x) + (l.b*p.y)); }
                          // compute this
void closestPoint(line l, point p, point &ans
    ) {
```

```
  line perpendicular;          //
     perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {              //
     special case 1: vertical line
    ans.x = -(l.c);   ans.y = p.y;
      return; }
  if (fabs(l.a) < EPS) {            //
     special case 2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);
      return; }
  pointSlopeToLine(p, 1/l.a, perpendicular);
                // normal line
  // intersect line l with this perpendicular
      line
  // the intersection point is the closest
     point
  areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &
    ans) {
  point b;
  closestPoint(l, p, b);
     // similar to distToLine
  vec v = toVec(p, b);
                                  // create a
     vector
  ans = translate(translate(p, v), v); }
            // translate p twice

// returns the dot product of two vectors a
    and b
double dot(vec a, vec b) { return (a.x*b.x +
    a.y*b.y); }

// returns the squared value of the
    normalized vector
double norm_sq(vec v) { return v.x*v.x + v.y*
    v.y; }

// returns the distance from p to the line
    defined by
// two points a and b (a and b must be
    different)
// the closest point is stored in the 4th
    parameter (byref)
double distToLine(point p, point a, point b,
    point &c) {
  // formula: c = a + u*ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));
                          // translate a to c
  return dist(p, c); }           // Euclidean
     distance between p and c

// returns the distance from p to the line
    segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th
    parameter (byref)
double distToLineSegment(point p, point a,
    point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);
                          // closer to a
    return dist(p, a); }          // Euclidean
       distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);
                          // closer to b
    return dist(p, b); }          // Euclidean
       distance between p and b
  return distToLine(p, a, b, c); }
     // run distToLine as above

double angle(point a, point o, point b) {  //
    returns angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa)*
     norm_sq(ob))); }

// returns the cross product of two vectors a
    and b
double cross(vec a, vec b) { return a.x*b.y -
    a.y*b.x; }

//// another variant
// returns 'twice' the area of this triangle
    A-B-c
// int area2(point p, point q, point r) {
//   return p.x * q.y - p.y * q.x +
//          q.x * r.y - q.y * r.x +
//          r.x * p.y - r.y * p.x;
// }
// note: to accept collinear points, we have
    to change the '> 0'
// returns true if point r is on the left
```

```
   side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > -
     EPS; }
// returns true if point r is on the same
   line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))
     ) < EPS; }
```

8.2 Circles

```
int insideCircle(point_i p, point_i c, int r)
    { // all integer version
  int dx = p.x - c.x, dy = p.y - c.y;
  int Euc = dx * dx + dy * dy, rSq = r * r;
                  // all integer
  return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;
     } //inside/border/outside
bool circle2PtsRad(point p1, point p2, double
    r, point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
              (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) *
      h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) *
      h;
  return true; }          // to get the other
     center, reverse p1 and p2
```

8.3 Triangles

```
double perimeter(double ab, double bc, double
    ca) {
  return ab + bc + ca; }
double perimeter(point a, point b, point c) {
  return dist(a, b) + dist(b, c) + dist(c, a)
     ; }
double area(double ab, double bc, double ca)
   {
  // Heron's formula, split sqrt(a * b) into
     sqrt(a) * sqrt(b); in implementation
  double s = 0.5 * perimeter(ab, bc, ca);
  return sqrt(s) * sqrt(s - ab) * sqrt(s - bc
     ) * sqrt(s - ca); }
double area(point a, point b, point c) {
  return area(dist(a, b), dist(b, c), dist(c,
      a)); }
double rInCircle(double ab, double bc, double
    ca) {
return area(ab, bc, ca) / (0.5 * perimeter(ab
   , bc, ca)); }
double rInCircle(point a, point b, point c) {
return rInCircle(dist(a, b), dist(b, c), dist
   (c, a)); }
// assumption: the required points/lines
   functions have been written
// returns 1 if there is an inCircle center,
   returns 0 otherwise
// if this function returns 1, ctr will be
   the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3,
   point &ctr, double &r) {
r = rInCircle(p1, p2, p3);
if (fabs(r) < EPS) return 0;
                  // no inCircle center
line l1, l2;                        // compute
   these two angle bisectors
double ratio = dist(p1, p2) / dist(p1, p3);
point p = translate(p2, scale(toVec(p2, p3),
   ratio / (1 + ratio)));
pointsToLine(p1, p, l1);
ratio = dist(p2, p1) / dist(p2, p3);
p = translate(p1, scale(toVec(p1, p3), ratio
   / (1 + ratio)));
pointsToLine(p2, p, l2);
areIntersect(l1, l2, ctr);          // get
   their intersection point
return 1; }
double rCircumCircle(double ab, double bc,
   double ca) {
return ab * bc * ca / (4.0 * area(ab, bc, ca)
   ); }
```

```
double rCircumCircle(point a, point b, point
   c) {
return rCircumCircle(dist(a, b), dist(b, c),
   dist(c, a)); }

// assumption: the required points/lines
   functions have been written
// returns 1 if there is a circumCenter
   center, returns 0 otherwise
// if this function returns 1, ctr will be
   the circumCircle center
// and r is the same as rCircumCircle
int circumCircle(point p1, point p2, point p3
   , point &ctr, double &r){
double a = p2.x - p1.x, b = p2.y - p1.y;
double c = p3.x - p1.x, d = p3.y - p1.y;
double e = a * (p1.x + p2.x) + b * (p1.y + p2
   .y);
double f = c * (p1.x + p3.x) + d * (p1.y + p3
   .y);
double g = 2.0 * (a * (p3.y - p2.y) - b * (p3
   .x - p2.x));
if (fabs(g) < EPS) return 0;

ctr.x = (d*e - b*f) / g;
ctr.y = (a*f - c*e) / g;
r = dist(p1, ctr);  // r = distance from
   center to 1 of the 3 points
return 1; }

// returns true if point d is inside the
   circumCircle defined by a,b,c
int inCircumCircle(point a, point b, point c,
    point d) {
return (a.x - d.x) * (b.y - d.y) * ((c.x - d.
   x) * (c.x - d.x) + (c.y - d.y) * (c.y - d.
   y)) +
        (a.y - d.y) * ((b.x - d.x) * (b.x - d.
           x) + (b.y - d.y) * (b.y - d.y)) * (
           c.x - d.x) +
        ((a.x - d.x) * (a.x - d.x) + (a.y - d.
           y) * (a.y - d.y)) * (b.x - d.x) * (
           c.y - d.y) -
        ((a.x - d.x) * (a.x - d.x) + (a.y - d.
           y) * (a.y - d.y)) * (b.y - d.y) * (
           c.x - d.x) -
        (a.y - d.y) * (b.x - d.x) * ((c.x - d.
           x) * (c.x - d.x) + (c.y - d.y) * (c
           .y - d.y)) -
        (a.x - d.x) * ((b.x - d.x) * (b.x - d.
           x) + (b.y - d.y) * (b.y - d.y)) * (
           c.y - d.y) > 0 ? 1 : 0;
}

bool canFormTriangle(double a, double b,
   double c) {
return (a + b > c) && (a + c > b) && (b + c >
    a); }
```

8.4 Polygon

```
// returns the perimeter, which is the sum of
    Euclidian distances
// of consecutive line segments (polygon
   edges)
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)
    // remember that P[0] = P[n-1]
    result += dist(P[i], P[i+1]);
  return result; }

// returns the area
double area(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)
                  // Shoelace formula
    result += (P[i].x*P[i+1].y - P[i+1].x*P[i
       ].y); // if all points are int
  return fabs(result)/2.0; }     // result
     can be int(eger) until last step

double area_alternative(const vector<point> &
   P) {
  double result = 0.0; point O(0.0, 0.0);
  for (int i = 0; i < (int)P.size()-1; i++)
    result += cross(toVec(O, P[i]), toVec(O,
       P[i+1]));
  return fabs(result) / 2.0; }

// returns true if we always make the same
   turn while examining
// all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false;   // a point/sz
     =2 or a line/sz=3 is not convex
```

```
  bool firstTurn = ccw(P[0], P[1], P[2]);
                  // remember one result
  for (int i = 1; i < sz-1; i++)
     // then compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 :
        i+2]) != firstTurn)
      return false;              // different
         sign -> this polygon is concave
  return true; }
                                          // this
     polygon is convex

// returns true if point p is in either
   convex/concave polygon P
bool inPolygon(point pt, const vector<point>
   &P) {
  if ((int)P.size() < 3) return false;
                    // avoid point or line
  double sum = 0;     // assume the first
     vertex is equal to the last vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
        sum += angle(P[i], pt, P[i+1]);
                               // left turn/
           ccw
    else sum -= angle(P[i], pt, P[i+1]); }
                         // right turn/cw
  return fabs(sum) > PI; }   // 360d -> in, 0
     d -> out, we have large margin

// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q,
   point A, point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs(a * p.x + b * p.y + c);
  double v = fabs(a * q.x + b * q.y + c);
  return point((p.x * v + q.x * u) / (u+v), (
     p.y * v + q.y * u) / (u+v)); }

// cuts polygon Q along the line formed by
   point a -> point b
// (note: the last point must be the same as
   the first point)
vector<point> cutPolygon(point a, point b,
   const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a
       , Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(
       toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);
              // Q[i] is on the left of ab
    if (left1 * left2 < -EPS)          // edge
       (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i
         +1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());          // make P'
       s first point = P's last point
  return P; }

vector<point> CH_Andrew(vector<point> &Pts) {
  int n = Pts.size(), k = 0;
  vector<point> H(2*n);
  sort(Pts.begin(), Pts.end());          //
     sort the points lexicographically
  for (int i = 0; i < n; i++) {
                                 // build lower
     hull
    while (k >= 2 && ccw(H[k-2], H[k-1], Pts[
       i]) <= 0) k--;
    H[k++] = Pts[i];
  }
  for (int i = n-2, t = k+1; i >= 0; i--) {
                    // build upper hull
    while (k >= t && ccw(H[k-2], H[k-1], Pts[
       i]) <= 0) k--;
    H[k++] = Pts[i];
  }
  H.resize(k);
  return H;
}

point pivot(0, 0);
vector<point> CH_Graham(vector<point> &Pts) {
  vector<point> P(Pts);        // copy all
     points so that Pts is not affected
  int i, j, n = (int)P.size();
  if (n <= 3) {              // corner cases: n
     =1=point, n=2=line, n=3=triangle
    if (!(P[0] == P[n-1])) P.push_back(P[0]);
       // safeguard from corner case
    return P; }
      // the CH is P itself
  // first, find P0 = point with lowest Y and
```

```
    if tie: rightmost X
int P0 = 0;
for (i = 1; i < n; i++)
   // O(n)
  if (P[i].y < P[P0].y || (P[i].y == P[P0].
     y && P[i].x > P[P0].x))
    P0 = i;
swap(P[0], P[P0]);
                                  // swap P
   [P0] with P[0]
// second, sort points by angle w.r.t.
   pivot P0, O(n log n) for this sort
pivot = P[0];                     // use
   this global variable as reference
sort(++P.begin(), P.end(), [](point a,
   point b) {  // we do not sort P[0]
  if (collinear(pivot, a, b))
                                  // special
      case
    return dist(pivot, a) < dist(pivot, b);
        // check which one is closer
  double d1x = a.x-pivot.x, d1y = a.y-pivot
     .y;
  double d2x = b.x-pivot.x, d2y = b.y-pivot
     .y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)
     ) < 0; }); // compare 2 angles

// third, the ccw tests, although complex,
   it is just O(n)
vector<point> S;
S.push_back(P[n-1]); S.push_back(P[0]); S.
   push_back(P[1]);   // initial S
i = 2;
   // then, we check the rest
while (i < n) {      // note: n must be >= 3
    for this method to work, O(n)
  j = (int)S.size()-1;
  if (ccw(S[j-1], S[j], P[i])) S.push_back(
     P[i++]);  // left turn, accept
  else S.pop_back(); }   // or pop the top
     of S until we have a left turn
return S; } // return the result, overall O
   (n log n) due to angle sorting
```