UNIPD - SWERC Notebook

Contents

1 Utils

1.1 Union Find

```
struct UnionFind {
  vector<int> p, size;
  int numSets;
  UnionFind(int n) {
    p = vector<int>(n);
    size = vector<int>(n);
    numSets = n;
    for (int i = 0; i < N; ++i){
      p[i] = i; size[i] = 1;
    }
  }
  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 uniteSets(int i, int j){
    i = findSet(i);
    j = findSet(j);
    if(i==j)
      return;
    numSets--;
    if(size[i] > size[j]){
      int tmp = j;
      j = i; i = tmp;
    }
    size[j] += size[i];
    p[i] = j
    }
};
```

1.2 Sliding Window

```
static vector<int> slidingWindowMin(vector<int> arr, int n, int k){
  vector<int> sol(n-k+1);
  int c=0;
  deque<int> q(0);
  int i;
  for (i = 0; i < k; ++i) {
    // previous smaller elements are useless so remove from q
    while (!q.empty() && arr[i] <= arr[q.back()])
      q.pop_back(); // Remove from rear
    q.push_back(i);
  }
  // Process rest of the elements, i.e., from arr[k] to arr[n-1]
  for (; i < n; ++i) {
    // element at the front is the largest of prev window, so add it
    sol[c++] = arr[q.front()];

    // Remove the elements which are out of this window
    while ((!q.empty()) && q.front() <= i - k)
      q.pop_front();
    // Remove elements smaller than currently being added element
    while ((!q.empty()) && arr[i] <= arr[q.back()])
      q.pop_back();
    // Add current element at the rear of q
    q.push_back(i);
  }
  //add max element of last window
  sol[c] = arr[q.front()];
  return sol;
}
```

1.3 BinarySearch

```
int bound(int[] data, int element){
  int first = 0, last = data.length;
  int mid;
  while (first < last) {
    mid = first + ((last - first) >> 1);
    if (data[mid] < element) // <= for upper
      first = mid + 1;
    else
      last = mid;
  }
  return first;
}
```

1.4 Treap

```
Treap
struct item {
    int key, prior;
    item * l, * r;
    item() { }
    item (int key, int prior) : key(key), prior(prior), l(NULL), r(NULL) { }
};
typedef item * pitem;

void split (pitem t, int key, pitem & l, pitem & r) {
    if (!t)
        l = r = NULL;
    else if (key < t->key)
        split (t->l, key, l, t->l),  r = t;
    else
        split (t->r, key, t->r, r),  l = t;
}

void insert (pitem & t, pitem it) {
    if (!t)
        t = it;
    else if (it->prior > t->prior)
        split (t, it->key, it->l, it->r),  t = it;
    else
        insert (it->key < t->key ? t->l : t->r, it);
}

void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
}

void erase (pitem & t, int key) {
    if (t->key == key)
        merge (t, t->l, t->r);
    else
        erase (key < t->key ? t->l : t->r, key);
}

pitem unite (pitem l, pitem r) {
    if (!l || !r)  return l ? l : r;
    if (l->prior < r->prior)  swap (l, r);
    pitem lt, rt;
    split (r, l->key, lt, rt);
    l->l = unite (l->l, lt);
    l->r = unite (l->r, rt);
    return l;
}

int cnt (pitem t) {
```

```
    return t ? t->cnt : 0;
}

void upd_cnt (pitem t) {
    if (t)
        t->cnt = 1 + cnt(t->l) + cnt (t->r);
}
//it's sufficient to add calls of upd_cnt() to the end of insert, erase, split and merge to keep cnt values
      up-to-date

//random migliore
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
```

1.5 Lazy Segment Tree

```
int N,M,K;

struct segTree{
    struct Node{
        ll d, lazy;
    };
    vector<Node> data;
    int n;
    void init(int x){
        n = 1; while( n < x ) n *= 2;
        data.resize(n*2+10);
    }
    void propagate(int node, int nodeL, int nodeR){
        if( data[node].lazy == 0 ) return;
        ll len = nodeR - nodeL + 1;
        data[node].d += len*data[node].lazy;
        if( len > 1 ){
            data[node*2].lazy += data[node].lazy;
            data[node*2+1].lazy += data[node].lazy;
        }
        data[node].lazy = 0;
    }

    void update(int l, int r, ll val, int node, int nodeL, int nodeR){
        propagate(node, nodeL, nodeR);
        if( l > nodeR || r < nodeL ) return;
        if( l <= nodeL && nodeR <= r ){
            data[node].lazy += val;
            propagate(node,nodeL,nodeR);
            return;
        }
        update(l,r,val,node*2,nodeL,(nodeL+nodeR)/2);
        update(l,r,val,node*2+1,(nodeL+nodeR)/2+1,nodeR);
        data[node].d = data[node*2].d + data[node*2+1].d;
    }

    ll query(int l, int r, int node, int nodeL, int nodeR){
        propagate(node, nodeL, nodeR);
        if( l > nodeR || r < nodeL ) return 0;
        if( l <= nodeL && nodeR <= r ){
            return data[node].d;
        }
        ll sum = 0;
        sum += query(l,r,node*2,nodeL,(nodeL+nodeR)/2);
        sum += query(l,r,node*2+1,(nodeL+nodeR)/2+1,nodeR);
        return sum;
    }

};
```

2 DP

2.1 2D range sum

```
int max2DSum (vector<vector<int>> a){
  for(int i=0; i<a.size(); i++){
    for(int j=0; j<a[0].size(); j++){
      if(j>0) a[i][j] += a[i][j-1];
    }
  }
  int mx = 0;
  for(int l=0; l<a[0].size(); l++){
    for(int r=l; r<a[0].size(); r++){
      int sum = 0;
      for(int row=0; row<a.size(); row++){
        if(l>0) sum += a[row][r] - a[row][l-1];
        else  sum += a[row][r];
        // Kadane s algorithm on rows
        if (sum < 0)  sum = 0;
        mx = max(mx, sum);
      }
    }
  }
  return mx;
```

2.2 SOS DP

```
// O(4^n) approach to solve F[mask] = sum, over all subsets i of mask, of A[i]
for(int mask = 0;mask < (1<<N); ++mask){
  for(int i = 0;i < (1<<N); ++i){
    if((mask&i) == i){
      F[mask] += A[i];
    }
  }
}
// O(3^n) approach
for (int mask = 0; mask < (1<<n); mask++){
  F[mask] = A[0];
    // iterate over all the subsets of the mask
    for(int i = mask; i > 0; i = (i-1) & mask){
      F[mask] += A[i];
    }
}
//O(n 2^n) approach
for(int mask = 0; mask < (1<<N); ++mask){
  dp[mask][-1] = A[mask]; //handle base case separately (leaf states)
  for(int i = 0;i < N; ++i){
    if(mask & (1<<i))
      dp[mask][i] = dp[mask][i-1] + dp[mask^(1<<i)][i-1];
    else
      dp[mask][i] = dp[mask][i-1];
  }
  F[mask] = dp[mask][N-1];
}
//memory optimized, super easy to code.
for(int i = 0; i<(1<<N); ++i)
  F[i] = A[i];
for(int i = 0;i < N; ++i) for(int mask = 0; mask < (1<<N); ++mask){
  if(mask & (1<<i))
    F[mask] += F[mask^(1<<i)];
}
```

3 Graph

3.1 Graph traversal

```
vector<int> vis;
bool hasCycle = false;
void cycle_dfs(int v){
  vis[v] = 1;
  for(int i : adj[v]){
    if(vis[i] == 0)
      cycle_dfs(i);
    else if(vis[i] == 1)
      hasCycle = true;
  }
  vis[v] = 2;
}


bool k_partite(int v){
  vector<int> colOk(k+1);
  for(int u : adj[v]){
    colOk[color[u]] = 1;
  }
  for(int i=1; i<=k; i++){
    if(colOk[i] == 1)
      continue;
    color[v] = i;
    bool flg = false;
    for(int u : adj[v]){
      if(color[u] == 0){
        if(!k_partite(u)){
          flg = true;
          break;
        }
      }
```

```
    }
    if(flg)
      continue;
    return true;
  }
  color[v] = 0;
  return false;
}
```

3.2 Tarjan

```
int V,used[MAXV],disc[MAXV],low[MAXV];
stack<int> st;
vector<int> mam[MAXV];
vector<vector<int>> SCC;

void tarjanDfs(int n){
    static int time = 0;

    used[n] = 1;
    low[n] = disc[n] = ++time;
    st.push(n);

    for(int x:mam[n])
        if(used[x] == 0){
            tarjanDfs(x);
            low[n] = min(low[n],low[x]);
        }
        else if(used[x] == 1)
            low[n] = min(low[n],disc[x]);

    if(disc[n] == low[n]){
        SCC.push_back(vector<int>());
        while(st.top() != n){
            SCC.back().push_back(st.top());
            used[st.top()] = 2;
            st.pop();
        }
        SCC.back().push_back(n);
        used[n] = 2;
        st.pop();
    }
}

vector<vector<int>> tarjan(){
    for(int i=0;i<V;++i)
        if(!used[i])
            tarjanDfs(i);

    return SCC;
}
```

3.3 Articulation Points

```
int V,disc[MAXV],low[MAXV];
bool used[MAXV];
vector<int> mam[MAXV],AP;

void apDfs(int n, int par = -1, bool isRoot = true){
    static int time = 0;

    used[n] = true;
    low[n] = disc[n] = ++time;

    int nChild = 0;
    bool added = false;

    for(int x:mam[n])
        if(!used[x]){
            apDfs(x,n,false);
            ++nChild;
            low[n] = min(low[n],low[x]);
            if(!added && !isRoot && low[x]>disc[n]){
                added = true;
                AP.push_back(n);
            }
        }
        else if(x != par)
            low[n] = min(low[n],disc[x]);

    if(isRoot){
        if(nChild >= 2)
            AP.push_back(n);
    }
}

vector<int> articulationPoints(){
    for(int i=0;i<V;++i)
        if(!used[i])
            apDfs(i);

    return AP;
}
```

3.4 Bridges

```
int V,disc[MAXV],low[MAXV];
bool used[MAXV];
vector<int> mam[MAXV];
vector<pair<int,int>> bridges;

void bridgesDfs(int n, int par = -1){
    static int time = 0;

    used[n] = true;
    low[n] = disc[n] = ++time;

    for(int x:mam[n])
        if(!used[x]){
            bridgesDfs(x,n);
            low[n] = min(low[n],low[x]);
            if(low[x]>disc[n])
                bridges.push_back({n,x});
        }
        else if(x != par)
            low[n] = min(low[n],disc[x]);
}

vector<pair<int,int>> findBridges(){
    for(int i=0;i<V;++i)
        if(!used[i])
            bridgesDfs(i);

    return bridges;
}
```

3.5 Minimum Spanning Tree

```
typedef tuple<int, int, int> iii;
vector<iii> edgeList;
int n; // # of vertexes
int kruskal(){
  // how to add edges: edgeList.push_back({dist, i, j});
  sort(edgeList.begin(), edgeList.end());
  int cost = 0, count = 0, i = 0;
  UnionFind uf = new UnionFind(n);
  while(i < n && (count < (n-1))){
    auto [w, u, v] = edgeList[i];
    if(!uf.isSameSet(u, v)){
      cost += w;
      count++;
      uf.unionSets(u, v);
    }
  }
  return cost;
}
}
```

3.6 Dijkstra

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;
void dijkstra(int s, vector<int> &d, vector<int> &p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    d[s] = 0;
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;
    q.push({0, s});
    while (!q.empty()) {
        int v = q.top().second;
        int d_v = q.top().first;
        q.pop();
        if (d_v != d[v])
            continue;
```

```
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;
            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
                q.push({d[to], to});
            }
        }
    }
}
```

3.7 Floyd-Wharshall

```
const int INF = 1000000000;
vector<vector<int>> adjMat;
void floydWarshall(){
  //  classic floyd-warshall algorithm...
  for (int i = 0; i < n; i++){
    adjMat[i][i] = 0;
  }
  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if(adjMat[i][k]==INF || adjMat[k][j]==INF)
          continue;
        adjMat[i][j] = Math.min(adjMat[i][j], adjMat[i][k] + adjMat[k][j]);
      }
    }
  }
  // ...ends here
  for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++){
      for (int k = 0; adjMat[i][j] != -INF && k < n; k++) {
        if (adjMat[i][k] != INF && adjMat[k][j] != INF && adjMat[k][k] < 0)
        {
          adjMat[i][j] = -INF;
        }
      }
    }
  }
}
```

3.8 Dinic Max Flow

```

#define ll long long
#define pb push_back

struct FlowEdge {
    int v, u;
    ll cap, flo = 0;
    FlowEdge(int v, int u, ll cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
    const ll flow_inf = 1e18;
    vector<FlowEdge> edj;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> lvl, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        lvl.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, ll cap) {
        edj.pb({v, u, cap});
        edj.pb({u, v, 0});
        adj[v].pb(m);
        adj[u].pb(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edj[id].cap - edj[id].flo < 1)
                    continue;
                if (lvl[edj[id].u] != -1)
                    continue;
                lvl[edj[id].u] = lvl[v] + 1;
                q.push(edj[id].u);
            }
        }
        return lvl[t] != -1;
    }

    ll dfs(int v, ll pu) {
        if (pu == 0)
            return 0;
        if (v == t)
            return pu;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
            int id = adj[v][cid];
            int u = edj[id].u;
            if (lvl[v] + 1 != lvl[u] || edj[id].cap - edj[id].flo < 1)
                continue;
            ll tr = dfs(u, min(pu, edj[id].cap - edj[id].flo));
            if (tr == 0)
                continue;
            edj[id].flo += tr;
            edj[id ^ 1].flo -= tr;
            return tr;
        }
        return 0;
    }

    ll flow() {
        ll f = 0;
        while (true) {
            fill(lvl.begin(), lvl.end(), -1);
            lvl[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (ll pu = dfs(s, flow_inf)) {
                f += pu;
            }
        }
        return f;
    }
};
```

3.9 LCA

```
int par[maxn][logn], d[maxn];
vector<int> adj[maxn];

void dfs0(int v, int f, int dd){
    d[v] = dd;;
    int curr = par[v][0] = f;
    for(int i = 1; (1 << i) <= dd; i++){
        par[v][i] = par[curr][i - 1];
        curr = par[curr][i - 1];
    }
    for(auto u : adj[v])
        if(u != f)
            dfs0(u, v, dd + 1);
}

int lca(int a, int b){
    if(d[a] > d[b]) swap(a, b);
    int diff = d[b] - d[a];
    for(int i = 0; i < logn; i++)
        if(diff & (1 << i))
            b = par[b][i];
    if(a == b) return a;
    for(int i = logn - 1; i >= 0; i--)
        if(par[a][i] != par[b][i])
            a = par[a][i], b = par[b][i];
    return par[a][0];
}
```

3.10 Bipartite Matching

```
vector<vector<int>> edgeMat;
// edgeMat[i][j]==1 means there's an edge from appl i to job j
vector<int> visited, matchR;
static int n, m;
```

```

bool bpm(int u) {
  for (int v = 0; v < m; v++) {
    if ((edgeMat[u][v]==1) && (visited[v]==0)) {
      visited[v] = 1;
      if (matchR[v] < 0 || bpm(matchR[v])) {
        matchR[v] = u;
        return true;
      }
    }
  }
  return false;
}
// Returns maximum number of matching from n app to m jobs, O(VE)
int maxBipartiteMatching() {
  // The value of matchR[i] is the app assigned to job i
  matchR.assign(m, -1);
  int result = 0;
  for (int u = 0; u < n; u++) {
    visited = vector<int>(m);
    // Find if the applicant 'u' can get a job
    if (bpm(u))
      result++;
  }
  return result;
}
```

4 Math

4.1 Math Utils

```
ll gcd(ll a, ll b) {
  while(b > 0){
    ll tmp = a%b;
    a = b;
    b = tmp;
  }
  return a;
}

ll lcm(ll a, ll b){
  return (a / gcd(a, b)) * b;
}

ll mod(ll a, ll M){
  return (a%M + M)%M;
}

// Find the greatest common factor between two numbers
  ll gcf(ll a, ll b) {
      return b == 0 ? a : gcf(b, a % b);
  }

ll* extendedEuclid(ll a, ll b){ //finds x,y so that ax+by=gcd(a,b)
  ll xx = 0, y = 0;
  ll yy = 1, x = 1;
  while(b > 0){
    ll q = a / b;
    ll tmp = b;  b = a%b;  a = tmp;
    tmp = xx; xx = x - q*xx; x = tmp;
    tmp = yy; yy = y - q*yy; y = tmp;
  }
  return new ll[3]{a, x, y};
}

vector<ll> modularLinearSolver(ll a, ll b, ll M){ //solution to ax = b (mod M)
      vector<ll> sol;
  ll *tmp = extendedEuclid(a, M);
  ll g = tmp[0], x = tmp[1], y = tmp[2];
  if(b%g == 0){
    x = mod(x*(b/g), M);
    for(ll i=0; i<g; i++)
      sol.push_back(mod(x + i*(M/g), M));
  }
  return sol;
}

ll modInverse(ll a, ll M){ // returns b so that ab = 1 (mod n)
  ll *tmp = extendedEuclid(a, M);
  ll g = tmp[0], x = tmp[1], y = tmp[2];
  if(g > 1) return -1;
  return mod(x, M);
}

ll* linearDiophantine(ll a, ll b, ll c){ // computes x,y so that ax+by=c
  if(a==0 && b==0){
    if(c==0) return NULL;
    return new ll[2]{0,0};
  }
  if(a==0){
    if(c%b == 0) return NULL;
    return new ll[2]{0, c/b};
  }
  if(b==0){
    if(c%a == 0) return NULL;
    return new ll[2]{c/a, 0};
  }
  ll g = gcd(a, b);
  if(c % g != 0) return NULL;
  ll x = c / g * modInverse(a/g, b/g);
  ll y = (c - a*x) / b;
  return new ll[2]{x, y};
}

ll modMul(ll a, ll b,  ll M) {
  ll res = 0;
  a = a % M;
  while (b > 0) {
      if (b % 2 == 1)  {
    res = (res + a) % M;
      }
      a = (a * 2) % M;
      b /= 2;
  }
  return res % M;
}
```

4.2 NextCombination

```

// Use this method in combination with a do while loop to generate all the combinations
  // of a set choosing r elements in a iterative fashion. This method returns
  // false once the last combination has been generated.
  // NOTE: Originally the selection needs to be initialized to {0,1,2,3 ... r-1}
bool nextCombination(int v[], int n, int r) {
  //assert(r<=n);
  int i = r - 1;
  while (v[i] == n - r + i)
    if (--i < 0)
      return 0;
  v[i]++;
  for (int j = i + 1; j < r; j++)
    v[j] = v[i] + j - i;
  return 1;
}
```

4.3 Linear Solver

```
vector<ld> linearSolver(vector<vector<ld>> a) {
    int n = a.size();
    int m = a[0].size() - 1;
    vector<int> where(m,-1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs(a[i][col]) > abs(a[sel][col]))
                sel = i;
        if (abs(a[sel][col]) < eps)
          continue;
        for (int i=col; i<=m; ++i){
            ld tmp = a[sel][i];
            a[sel][i] = a[row][i];
            a[row][i] = tmp;
        }
        where[col] = row;
        for (int i=0; i<n; ++i)
            if (i != row) {
                ld c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                a[i][j] -= a[row][j] * c;
            }
        ++row;
    }
    vector<ld> ans(m);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
```

```
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs(sum - a[i][m]) > eps)
        return vector<long double>(); // no solution
    }
    for (int i=0; i<m; ++i)
      if (where[i] == -1)
          return vector<long double>{1.0/0.0}; // infinite solutions
    return ans; // one solution
}
```

4.4 Fast Matrix Exponentiation

```
vector<vector<int>> matrixPower(vector<vector<int>> base, int pow){
  vector<vector<int>> ans(base.size());
  for (int i = 0; i < N; i++){
    ans[i] = vector<int>(base.size());
    ans[i][i] = 1;// generate identity matrix
  }
  while ( pow != 0 ){// binary exponentiation
    if  ( (pow&1) != 0 )
      ans = multiplyMatrix(ans, base);
    base = multiplyMatrix(base, base);
    pow >>= 1;
  }
  return  ans;
}
```

4.5 Factorization

```
#define ll long long
int MAXN = 10000000;
vector<int> spf(MAXN);
void smallestPrimeFactor(){ //O(nloglogn) sieve
  spf[1] = 1; spf[2] = 2;
  for (int i=3; i<MAXN; i+=2)
    spf[i] = i;
  for (int i=4; i<MAXN; i+=2)
    spf[i] = 2;
  for (int i=3; i*i<MAXN; i++) {
    if (spf[i] == i){ // checking if i is prime
      for (int j=i*i; j<MAXN; j+=i) // marking SPF for all numbers divisible by i
        if (spf[j]==j)
          spf[j] = i;
    }
  }
}
vector<ll> fastFactorize(ll x){ // run spf first
  vector<ll> ret;
  while (x != 1) {
    ret.push_back(spf[x]);
    x = x / spf[x];
  }
  return ret;
}

vector<ll> factorize(ll x){ //O(sqrtN)
  vector<ll> ret;
  while(x%2==0){
    ret.push_back(2);
    x = x>>1;
  }
  for(ll i = 3; i*i <= x; i = i+2) {
    while(x%i == 0) {
      ret.push_back(i);
      x = x/i;
    }
  }
  return ret;
}

vector<ll> getDivisors(int n) { // O(sqrtN)
  vector<ll> sol;
  for (int i=1; i*i<=n; i++) {
    if(n%i==0) {
      if(n/i == i) // check if divisors are equal
        sol.push_back(i);
      else{
        sol.push_back(i);
        sol.push_back(n/i); // push the second divisor in the vector
      }
    }
  }
  return sol;
}
```

4.6 FFT

```
using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    if (n == 1)
        return;

    vector<cd> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++) {
        a0[i] = a[2*i];
        a1[i] = a[2*i+1];
    }
    fft(a0, invert);
    fft(a1, invert);

    double ang = 2 * PI / n * (invert ? -1 : 1);
    cd w(1), wn(cos(ang), sin(ang));
    for (int i = 0; 2 * i < n; i++) {
        a[i] = a0[i] + w * a1[i];
        a[i + n/2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;
            a[i + n/2] /= 2;
        }
        w *= wn;
    }
}

vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, 0);
    fft(fb, 0);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, 1);

    vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}
```

4.7 Simplex

```
public class Simplex {

  static final double EPS = 1e-9;

  // The matrix given as an argument represents the function to be maximized
  // and each of the constraints. Constraints and objective function must be
  // normalized first through the following steps:
  // 1) RHS must be non-negative so multiply any inequalities failing this by -1
  // 2) Add positive coefficient slack variable on LHS of any <= inequality
  // 3) Add negative coefficient surplus variable on LHS of any >= inequality
  // 4) Add positive coefficient artificial variable on LHS of any >= inequality and any = equality.
  //
  // If any artificial variables were added, perform simplex once, maximizing the
  // negated sum of the artificial variables. If the maximum value is 0, take the
  // resulting matrix and remove the artificial variable columns and replace function
  // to maximise with original and run simplex again. If maximum value of simplex with
  // artificial variables is non-zero there is no solution. First column of m is the constants
  // on the RHS of all constraints. First row is the expression to maximise with all
  // coefficients negated. M[i][j] is the coefficient of the (j-1)th term in the
  // (i-1)th constraint (0 based).
  public static double simplex(double[][] m) {
    while (true) {
      double min = -EPS;
      int c = -1;
      for (int j = 1; j < m[0].length; j++) {
        if (m[0][j] < min) {
          min = m[0][j];
          c = j;
```

```
        }
      }
      if (c < 0) break;
      min = Double.MAX_VALUE;
      int r = -1;
      for (int i = 1; i < m.length; i++) {
        if (m[i][c] > EPS) {
          double v = m[i][0] / m[i][c];
          if (v < min) {
            min = v;
            r = i;
          }
        }
      }
      double v = m[r][c];
      for (int j = 0; j < m[r].length; j++) m[r][j] /= v;
      for (int i = 0; i < m.length; i++) {
        if (i != r) {
          v = m[i][c];
          for (int j = 0; j < m[i].length; j++) m[i][j] -= m[r][j] * v;
        }
      }
    }
    return m[0][0];
  }
}
```

4.8 Polynomials

```
/* Description: Given $n$ points (x[i], y[i]), computes an n-1-degree polynomial $p$ that
 *  passes through them: $p(x) = a[0]*x^0 + ... + a[n-1]*x^{n-1}$.
 *  For numerical precision, pick $x[k] = c*\cos(k/(n-1)*\pi), k=0 \dots n-1$.
 * Time: O(n^2)
 */

typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
  vd res(n), temp(n);
  rep(k,0,n-1) rep(i,k+1,n)
    y[i] = (y[i] - y[k]) / (x[i] - x[k]);
  double last = 0; temp[0] = 1;
  rep(k,0,n) rep(i,0,n) {
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
}

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

/* Description: Finds the real roots to a polynomial.
 * Usage: poly_roots({{2,-3,1}},-1e9,1e9) // solve x^2-3x+2 = 0
 * Time: O(n^2 \log(1/\epsilon))
 */

vector<double> poly_roots(Poly p, double xmin, double xmax) {
  if (sz(p.a) == 2) { return {-p.a[0]/p.a[1]}; }
  vector<double> ret;
  Poly der = p;
  der.diff();
  auto dr = poly_roots(der, xmin, xmax);
  dr.push_back(xmin-1);
  dr.push_back(xmax+1);
  sort(all(dr));
  rep(i,0,sz(dr)-1) {
    double l = dr[i], h = dr[i+1];
    bool sign = p(l) > 0;
    if (sign ^ (p(h) > 0)) {
      rep(it,0,60) { // while (h - l > 1e-8)
        double m = (l + h) / 2, f = p(m);
        if ((f <= 0) ^ sign) l = m;
        else h = m;
      }
      ret.push_back((l + h) / 2);
    }
  }
  return ret;
}
struct Poly {
  vector<double> a;
  double operator()(double x) const {
    double val = 0;
    for(int i = sz(a); i--;) (val *= x) += a[i];
    return val;
  }
  void diff() {
    rep(i,1,sz(a)) a[i-1] = i*a[i];
    a.pop_back();
  }
  void divroot(double x0) {
    double b = a.back(), c; a.back() = 0;
    for(int i=sz(a)-1; i--;) c = a[i], a[i] = a[i+1]*x0+b, b=c;
    a.pop_back();
  }
};
```

4.9 Linear recurrences

```
/* Description: Generates the $k$'th term of an $n$-order
 * linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$,
 * given $S[0 \dots n-1]$ and $tr[0 \dots n-1]$.
 * Faster than matrix multiplication.
 * Useful together with Berlekamp--Massey.
 * Usage: linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number
 * Time: O(n^2 \log k)
 */

const ll mod = 1000000007;
typedef vector<ll> Poly;
ll linearRec(Poly S, Poly tr, ll k) {
  int n = sz(S);

  auto combine = [&](Poly a, Poly b) {
    Poly res(n * 2 + 1);
    rep(i,0,n+1) rep(j,0,n+1)
      res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
    for (int i = 2 * n; i > n; --i) rep(j,0,n)
      res[i - 1 - j] = (res[i - 1 - j] + res[i] * tr[j]) % mod;
    res.resize(n + 1);
    return res;
  };
  Poly pol(n + 1), e(pol);
  pol[0] = e[1] = 1;
  for (++k; k; k /= 2) {
    if (k % 2) pol = combine(pol, e);
    e = combine(e, e);
  }
  ll res = 0;
  rep(i,0,n) res = (res + pol[i + 1] * S[i]) % mod;
  return res;
}


/* Description: Recovers any $n$-order linear recurrence relation from the first
 * $2n$ terms of the recurrence.
 * Useful for guessing linear recurrences after brute-forcing the first terms.
 * Should work on any field, but numerical stability for floats is not guaranteed.
 * Output will have size $\le n$.
 * Usage: BerlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}
 */

vector<ll> BerlekampMassey(vector<ll> s) {
  int n = sz(s), L = 0, m = 0;
  vector<ll> C(n), B(n), T;
  C[0] = B[0] = 1;
  ll b = 1;
  rep(i,0,n) { ++m;
    ll d = s[i] % mod;
    rep(j,1,L+1) d = (d + C[j] * s[i - j]) % mod;
    if (!d) continue;
    T = C; ll coef = d * modpow(b, mod-2) % mod;
    rep(j,m,n) C[j] = (C[j] - coef * B[j - m]) % mod;
    if (2 * L > i) continue;
    L = i + 1 - L; B = T; b = d; m = 0;
  }

  C.resize(L + 1); C.erase(C.begin());
  trav(x, C) x = (mod - x) % mod;
  return C;
}

ll modpow(ll a, ll e) {
  if (e == 0) return 1;
  ll x = modpow(a * a % mod, e >> 1);
  return e & 1 ? x * a % mod : x;
}
```

4.10 Game of Nim

```
/*
```

```
We have a game which fulfills the following requirements:
    - There are two players who move alternately.
    - The game consists of states, and the possible moves in a state do not depend on whose turn it is.
    - The game ends when a player cannot make a move.
    - The game surely ends sooner or later.
    - The players have complete information about the states and allowed moves, and there is no randomness in the game.
The idea is to calculate Grundy numbers for each game state. It is calculated like so: mex({g_1, g_2, ..., g_n}),
where g_1, g_2, ..., g_n are the Grundy numbers of the states which are reachable from the current state. $mex$ gives the smallest nonnegative number that
is not in the set (mex(\{0, 1, 3\}) = 2, mex(\emptyset) = 0). If the Grundy number of a state is 0, then this state is a losing state. Otherwise it's a winning
state.
Sometimes a move in a game divides the game into subgames that are independent of each other. In this case, the Grundy number of a game state is
$mex({g_1, g_2, ..., g_n}), g_k = a_{k,1} xor a_{k,2} xor ... xor a_{k,m}$ meaning that move $k$ divides the game into $m$ subgames whose Grundy numbers are $a_{i,j}$.

Example: We have a heap with $n$ sticks. On each turn, the player chooses a heap and divides it into two nonempty heaps such that the heaps are of different size. The player
who makes the last move wins the game. Let $g(n)$ denote the Grundy number of a heap of size $n$. The Grundy number can be calculated by going though all possible ways to divide the heap into
two parts. E.g. $g(8) = mex(\{g(1) \oplus g(7), g(2) \oplus g(6), g(3) \oplus g(5)\}).$ Base case: $g(1) = g(2) = 0$, because these are losing states
*/


map<set<int>,int> grundy;
map<ll,set<int>> mp;

int getGrundy(set<int> x){
    // base case
    if( sz(x) == 0 ) return 0;
    if( grundy.find(x) != grundy.end() ) return grundy[x];

    set<int> S;
    int res = 0;

    auto iter = x.end(); iter--;
    int mx = *iter;

    // transition : which k to select
    for(int i=1;i<=mx;i++){
        set<int> nxt;
        for(auto e : x){
            if( e < i ) nxt.insert(e);
            else if( e == i ) continue;
            else nxt.insert(e-i);
        }
        S.insert(get_grundy(nxt));
    }

    // find mex and return
    while( S.find(res) != S.end() ) res++;
    grundy[x] = res;
    return res;
}
```

5 String

5.1 String alignment

```
#define MAXN 1001
#define MAXM 1001
int V[MAXN][MAXM];
int path[MAXN][MAXM];
int stringAlignment(string A, string B) //Needleman-Wunsch
{   // change scores accordingly
  int matchScore = 2, misScore = -1, spaceAScore = -1, spaceBScore = -1;
  memset(V, 0, sizeof(V));
  memset(path, -1, sizeof(path));
  for(int i=1; i<=A.size(); i++){
    V[i][0] = i * spaceBScore;
    path[i][0] = 1;
  }
  for(int i=1; i<=B.size(); i++){
    V[0][i] = i * spaceAScore;
    path[0][i] = 2;
  }
  for(int i=1; i<=A.size(); i++){
    for(int j=1; j<=B.size(); j++){
      int tmp[3];
      tmp[0] = V[i-1][j-1] + (A[i-1]==B[j-1] ? matchScore : misScore);
      tmp[1] = V[i-1][j] + spaceBScore;
      tmp[2] = V[i][j-1] + spaceAScore;
      int max = 0;
      if(tmp[1]>tmp[max]) max = 1;
      if(tmp[2]>tmp[max]) max = 2;
      path[i][j] = max;
      V[i][j] = tmp[max];
    }
  }
  return V[A.size()][B.size()];
}
```

5.2 Extended KMP

```
/* S[i] stores the maximum common prefix between s[i:] and t;
 * T[i] stores the maximum common prefix between t[i:] and t for i>0;
 */
int S[N], T[N];

void extKMP(const string&s, const string &t) {
  int m = t.size();
  T[0] = 0;
  int maT = 0;
  for (int i = 1; i < m; i++) {
    if (maT + T[maT] >= i) {
      T[i] = min(T[i - maT], maT + T[maT] - i);
    }else {
      T[i] = 0;
    }
    while (T[i] + i < m && t[T[i]] == t[T[i] + i])
      T[i]++;
    if (i + T[i] > maT + T[maT])
      maT = i;
  }
  int maS = 0;
  int n = s.size();
  for (int i = 0; i < n; i++) {
    if (maS + S[maS] >= i) {
      S[i] = min(T[i - maS], maS + S[maS] - i);
    }else {
      S[i] = 0;
    }
    while (S[i] < m && i + S[i] < n && t[S[i]] == s[S[i] + i])
      S[i]++;
    if (i + S[i] > maS + S[maS])
      maS = i;

  }
}
```

5.3 Suffix Array

```
struct SuffixArray{
  string s;
  int n, m;
  vector<int> ra, tra, phi, plcp, lcp;
  vector<int> sarr, tsa;
  // LCP[i] stores the LCP between previous suffix "s + SuffArr[i-1]" and current suffix "s + SuffArr[i]"

  void csort(int k){
    int i, sum, maxi = max(300, n);   // up to 255 ASCII chars or length of n
    vector<int> c(maxi,0);
    for (i = 0; i < n; i++)
      c[i + k < n ? ra[i + k] : 0]++;
    for (i = sum = 0; i < maxi; i++){
      int t = c[i]; c[i] = sum; sum += t;
    }
    for (i = 0; i < n; i++)
      tsa[c[sarr[i] + k < n ? ra[sarr[i] + k] : 0]++] = sarr[i];
    for (i = 0; i < n; i++)
      sarr[i] = tsa[i];
  }
  void constructSA() {
    int i, k, r;
    for (i = 0; i < n; i++) ra[i] = s[i]; //initial rank
    for (i = 0; i < n; i++) sarr[i] = i;
    for (k = 1; k < n; k <<= 1) {
      csort(k);
      csort(0);
      tra[sarr[0]] = r = 0;
      for (i = 1; i < n; i++)
        tra[sarr[i]] = (ra[sarr[i]] == ra[sarr[i-1]] && ra[sarr[i]+k] == ra[sarr[i-1]+k]) ? r : ++r;
      for (i = 0; i < n; i++) // update the rank array RA
        ra[i] = tra[i];
```

```
      }
    }
    void computeLCP() {
      int i, l;
      phi[sarr[0]] = -1;
      for (i = 1; i < n; i++)
        phi[sarr[i]] = sarr[i-1];         // remember which suffix is behind this suffix
      for (i = l = 0; i < n; i++) {                    // compute Permuted LCP in O(n)
        if (phi[i] == -1) { plcp[i] = 0; continue; }                // special case
        while (i + l < (int)s.size() && phi[i] + l < (int)s.size() && s[i + l] == s[phi[i] + l]) l++;
        plcp[i] = l;
        l = max(l-1, 0);
      }
      for (i = 1; i < n; i++)
        lcp[i] = plcp[sarr[i]];   // put the permuted LCP back to the correct position
    }
    int strncmp(string a, int i, string b, int j, int n){
      for (int k=0; i+k < (int)a.size() && j+k < (int)b.size(); k++)
        if (a[i+k] != b[j+k]) return a[i+k] - b[j+k];
      return 0;
    }
    vector<int> stringMatching(string str){ // string matching in O(m log n)
      //char P[] = str.toCharArray();
      m = str.size();
      int lo = 0, hi = n-1, mid = lo;
      while (lo < hi) {                                   // find lower bound
        mid = (lo + hi) / 2;
        int res = strncmp(s, sarr[mid], str, 0, m);
        if (res >= 0) hi = mid;
        else          lo = mid + 1;
      }
      if (strncmp(s,sarr[lo], str,0, m) != 0) return {-1, -1};         // if not found
      vector<int> ans = { lo, 0} ;
      lo = 0; hi = n - 1; mid = lo;
      while (lo < hi) {                 // if lower bound is found, find upper bound
        mid = (lo + hi) / 2;
        int res = strncmp(s, sarr[mid], str,0, m);
        if (res > 0) hi = mid;
        else         lo = mid + 1;
      }
      if (strncmp(s, sarr[hi], str,0, m) != 0) hi--;// special case
      ans[1] = hi;
      return ans;// return lower/upper bound of the range where the pattern is found
    }
      string LongestRpeatedSubstring(){ //longest repeated substring O(n)
      int i, idx = 0, mam = 0;
      for (i = 1; i < n; i++)
      if (lcp[i] > mam) {
        mam = lcp[i];
        idx = i;
      }
      return s.substr(sarr[idx], mam); // maxLCP is the length
    }
    int owner(int idx) { return (idx < n-m-1) ? 1 : 2; }
    string LongestCommonSubstring(string str0, string str1) {
      int i, mam = 0, idx = 0;
      m = str1.length();
      s = str0 + str1 + "#";   // append P and '#'
      n = s.size(); // update n
      ra = vector<int>(n,0);
      tra = vector<int>(n,0);
      sarr = vector<int>(n,0);
      tsa = vector<int>(n,0);
      phi = vector<int>(n,0);
      plcp = vector<int>(n,0);
      lcp = vector<int>(n,0);
      constructSA();
      computeLCP();
      for (i = 1, mam = -1; i < n; i++){
        if (lcp[i] > mam && owner(sarr[i]) != owner(sarr[i-1])) {  // different owner
          mam = lcp[i];
          idx = i;
        }
      }
      return s.substr(sarr[idx], mam);
    }
    void suffixarray(string str){
      s = str+"$";
      n = s.size();
      ra = vector<int>(n,0);
      tra = vector<int>(n,0);
      sarr = vector<int>(n,0);
      tsa = vector<int>(n,0);
      phi = vector<int>(n,0);
      plcp = vector<int>(n,0);
      lcp = vector<int>(n,0);
      constructSA();
      computeLCP();
    }
  };
```

5.4 Booths

```
public class BoothsAlgorithm {
  // Performs Booths algorithm returning the earliest index of the
  // lexicographically smallest string rotation. Note that comparisons
  // are done using ASCII values, so mixing lowercase and uppercase
  // letters may give you unexpected results, O(n)
  public static int leastCyclicRotation(String s) {
    s += s;
    int[] f = new int[s.length()];
    java.util.Arrays.fill(f, -1);
    int k = 0;
    for (int j = 1; j < s.length(); j++) {
      char sj = s.charAt(j);
      int i = f[j - k - 1];
      while (i != -1 && sj != s.charAt(k + i + 1)) {
        if (sj < s.charAt(k + i + 1)) k = j - i - 1;
        i = f[i];
      }
      if (sj != s.charAt(k + i + 1)) {
        if (sj < s.charAt(k)) k = j;
        f[j - k] = -1;
      } else f[j - k] = i + 1;
    }
    return k;
  }
}
```

5.5 Manachers

```
public class ManachersAlgorithm {

  // Manacher's algorithm finds the length of the longest palindrome O(n)
  // centered at a specific index. Since even length palindromes have
  // a center in between two characters we expand the string to insert
  // those centers, for example "abba" becomes "^#a#b#b#a#$" where the
  // '#' sign represents the center of an even length string and '^' & '$'
  // are the front and the back of the string respectively. The output
  // of this function gives the diameter of each palindrome centered
  // at each character in this expanded string, for instance:
  // manachers("abba") -> [0, 0, 1, 0, 1, 4, 1, 0, 1, 0, 0]
  public static int[] manachers(char[] str) {
    char[] arr = preProcess(str);
    int n = arr.length, c = 0, r = 0;
    int[] p = new int[n];
    for (int i = 1; i < n - 1; i++) {
      int invI = 2 * c - i;
      p[i] = r > i ? Math.min(r - i, p[invI]) : 0;
      while (arr[i + 1 + p[i]] == arr[i - 1 - p[i]]) p[i]++;
      if (i + p[i] > r) {
        c = i;
        r = i + p[i];
      }
    }
    return p;
  }

  // Pre-process the string by injecting separator characters.
  // We do this to account for even length palindromes, so we can
  // assign them a unique center, for example: "abba" -> "^#a#b#b#a#$"
  private static char[] preProcess(char[] str) {
    char[] arr = new char[str.length * 2 + 3];
    arr[0] = '^';
    for (int i = 0; i < str.length; i++) {
      arr[i * 2 + 1] = '#';
      arr[i * 2 + 2] = str[i];
    }
    arr[arr.length - 2] = '#';
    arr[arr.length - 1] = '$';
    return arr;
  }

  // This method finds all the palindrome substrings found inside
  // a string it uses Manacher's algorithm to find the diameter
  // of each palindrome centered at each position.
  public static java.util.TreeSet<String> findPalindromeSubstrings(String str) {
    char[] S = str.toCharArray();
    int N = S.length;
    int[] centers = manachers(S);
    java.util.TreeSet<String> palindromes = new java.util.TreeSet<>();

    for (int i = 0; i < centers.length; i++) {
```

```
      int diameter = centers[i];
      if (diameter >= 1) {

        // Even palindrome substring
        if (i % 2 == 1) {
          while (diameter > 1) {
            int index = (i - 1) / 2 - diameter / 2;
            palindromes.add(new String(S, index, diameter));
            diameter -= 2;
          }
          // Odd palindrome substring
        } else {
          while (diameter >= 1) {
            int index = (i - 2) / 2 - (diameter - 1) / 2;
            palindromes.add(new String(S, index, diameter));
            diameter -= 2;
          }
        }
      }
    }
    return palindromes;
  }

  public static void main(String[] args) {
    String s = "abbaabba";

    // Outputs: [a, aa, abba, abbaabba, b, baab, bb, bbaabb]
    System.out.println(findPalindromeSubstrings(s));
  }
}
```

6 Geometry

6.1 Lines and Points

```
#define dbl double
#define MAX_SIZE 1000

const dbl PI = 2.0*acos(0.0);
const dbl EPS = 1e-9; //too small/big???
dbl DEG_to_RAD(dbl d) { return d * PI / 180.0; }
dbl RAD_to_DEG(dbl r) { return r * 180.0 / PI; }

struct pt {
  dbl x,y;
  dbl length() { return sqrt(x*x+y*y); }
  dbl dist2() { return x*x + y*y; }

  int normalize(){
    dbl l = length();
    if(fabs(l)<EPS) return -1;
    x/=l; y/=l;
    return 0;
  }
  pt operator-(pt a){
    return {x-a.x , y-a.y};
  }
  pt operator+(pt a){
    return {x+a.x , y+a.y};
  }
  pt operator*(dbl sc){
    return {x*sc , y*sc};
  }
  pt operator/(dbl sc){
    return {x/sc , y/sc};
  }
  dbl cross(pt p) { return x*p.y - y*p.x; }
  dbl cross(pt a, pt b) { return (a-*this).cross(b-*this); }
  dbl dot(pt p) { return x*p.x + y*p.y; }

  pt perp() { return {-y, x}; }

  bool operator == (pt a){
    return x==a.x && y==a.y;
  }
};
bool operator<(const pt& a,const pt& b){
  if(fabs(a.x-b.x)<EPS) return a.y<b.y;
  return a.x<b.x;
}
dbl dist(pt& a, pt& b){
  return sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}
inline dbl dot(pt a, pt b){
  return(a.x*b.x+a.y*b.y);
}

void show(pt& p){
  cout << "(" << p.x << ", " << p.y << ")" << endl;
}
void show(vector<pt>& p){
  int i,n=p.size();
  for(i=0;i<n;i++) show(p[i]);
  cout << ":)" << endl;
}

double trap(pt a, pt b){
  return (0.5*(b.x - a.x)*(b.y + a.y));
}

double triarea(pt a, pt b, pt c){
  return fabs(trap(a,b)+trap(b,c)+trap(c,a));
}

 // POINTS AND LINES
int intersection(pt p1, pt p2, pt p3, pt p4, pt &r ){
  // two lines given by $p_1\rightarrow p_2$, $p_3 \rightarrow p_4$, r is the intersection point
  // returns: $-1$ if two lines are parallel

  dbl d = (p4.y - p3.y)*(p2.x-p1.x) - (p4.x - p3.x)*(p2.y - p1.y);
  if( fabs( d ) < EPS ) return -1;
  // might need to do something special!!!

  dbl ua /*, ub*/;
  ua = (p4.x - p3.x)*(p1.y-p3.y) - (p4.y-p3.y)*(p1.x-p3.x);
  ua /= d;
  // ub = (p2.x - p1.x)*(p1.y-p3.y) - (p2.y-p1.y)*(p1.x-p3.x);
  //ub /= d;
  r = p1 + (p2-p1)*ua;
  return 0;
}

void closestpt( pt p1, pt p2, pt p3, pt &r ){
  // the closest point on the line $p_1 \rightarrow p_2$ to $p_3$
  if( fabs( triarea( p1, p2, p3 ) ) < EPS ){
    r = p3; return;
  }
  pt v = p2-p1;
  v.normalize();
  dbl pr; //inner product
  pr = (p3.y-p1.y)*v.y + (p3.x-p1.x)*v.x;
  r = p1+v*pr;
}

//segments
dbl segDist(pt& s, pt& e, pt& p) {
  if (s==e) return (p-s).length();
  auto d = (e-s).length() * (e-s).length(), t = min(d,max(.0,dot((p-s),(e-s))));
  return ((p-s)*d-(e-s)*t).length()/d;
}
int segmentIntersection(pt& s1, pt& e1, pt& s2, pt& e2, pt& r1, pt& r2) {
  if (e1==s1) {
    if (e2==s2) {
      if (e1==e2) { r1 = e1; return 1; } //all equal
      else return 0; //different point segments
    } else return segmentIntersection(s2,e2,s1,e1,r1,r2);//swap
  }
  //segment directions and separation
  pt v1 = e1-s1, v2 = e2-s2, d = s2-s1;
  auto a = v1.cross(v2), a1 = v1.cross(d), a2 = v2.cross(d);
  if (a == 0) { //if parallel
    auto b1=s1.dot(v1), c1=e1.dot(v1),
         b2=s2.dot(v1), c2=e2.dot(v1);
    if (a1 || a2 || max(b1,min(b2,c2))>min(c1,max(b2,c2)))
      return 0;
    r1 = min(b2,c2)<b1 ? s1 : (b2<c2 ? s2 : e2);
    r2 = max(b2,c2)>c1 ? e1 : (b2>c2 ? s2 : e2);
    return 2-(r1==r2);
  }
  if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
  if (0<a1 || a<-a1 || 0<a2 || a<-a2)
    return 0;
  r1 = s1-v1*a2/a;
  return 1;
}

int hcenter( pt p1, pt p2, pt p3, pt& r ){
  // point generated by altitudes
  if( triarea( p1, p2, p3 ) < EPS ) return -1;
  pt a1, a2;
  closestpt( p2, p3, p1, a1 );
  closestpt( p1, p3, p2, a2 );
  intersection( p1, a1, p2, a2, r );
```

```
  return 0;
}

int center( pt p1, pt p2, pt p3, pt& r ){
  // point generated by circumscribed circle
  if( triarea( p1, p2, p3 ) < EPS ) return -1;
  pt a1, a2, b1, b2;
  a1 = (p2+p3)*0.5;
  a2 = (p1+p3)*0.5;
  b1.x = a1.x - (p3.y-p2.y);
  b1.y = a1.y + (p3.x-p2.x);
  b2.x = a2.x - (p3.y-p1.y);
  b2.y = a2.y + (p3.x-p1.x);
  intersection( a1, b1, a2, b2, r );
  return 0;
}

int bcenter( pt p1, pt p2, pt p3, pt& r ){
  // angle bisection
  if( triarea( p1, p2, p3 ) < EPS ) return -1;
  dbl s1, s2, s3;
  s1 = dist( p2, p3 );
  s2 = dist( p1, p3 );
  s3 = dist( p1, p2 );


  dbl rt = s2/(s2+s3);
  pt a1,a2;
  a1 = p2*rt+p3*(1.0-rt);
  rt = s1/(s1+s3);
  a2 = p1*rt+p3*(1.0-rt);
  intersection( a1,p1, a2,p2, r );
  return 0;
}
// PONTS, LINES and CIRCLES
int pAndSeg(pt& p1, pt& p2, pt& p){
  // the relation of the point $p$ and the segment $p_1 \rightarrow p_2$.
  // returns: $1$ if point is on the segment, $0$ if not on the line,
  //         $-1$ if on the line but not on the segment
  dbl s=triarea(p, p1, p2);
  if(s>EPS) return(0);
  dbl sg=(p.x-p1.x)*(p.x-p2.x);
  if(sg>EPS) return(-1);
  sg=(p.y-p1.y)*(p.y-p2.y);
  if(sg>EPS) return(-1);
  return(1);
}

int lineAndCircle(pt& oo, dbl r, pt& p1, pt& p2, pt& r1, pt& r2){
  // returns: $-1$ if there is no intersection
  //           $1$ if there is only one intersection
  pt m;
  closestpt(p1,p2,oo,m);
  pt v = p2-p1;
  v.normalize();

  dbl r0=dist(oo, m);
  if(r0>r+EPS) return -1;
  if(fabs(r0-r)<EPS){
    r1=r2=m;
    return 1;
  }
  dbl dd = sqrt(r*r-r0*r0);
  r1 = m-v*dd; r2 = m+v*dd;
  return 0;
}


// ANGLES
dbl angle(pt& p1, pt& p2, pt& p3){
  // angle from $p_1 \rightarrow p_2$ to $p_1 \rightarrow p_3$, returns $-\pi$ to %\pi%
  pt va = p2-p1;
  va.normalize();
  pt vb; vb.x=-va.y; vb.y=va.x;
  pt v = p3-p1;
  dbl x,y;
  x=dot(v, va);
  y=dot(v, vb);
  return(atan2(y,x));
}

dbl angle(dbl a, dbl b, dbl c){ // $a^2 = b^2 + c^2 - 2bc\cos{\gamma}$
  // in a triangle with sides $a$,$b$,$c$, the angle between $b$ and $c$
  // we do not check if $a$,$b$,$c$ is a triangle here
  dbl cs=(b*b+c*c-a*a)/(2.0*b*c);
  return(acos(cs));
}

void rotate(pt p0, pt p1, dbl a, pt& r){
  // rotate p1 around $p_0$ clockwise, by angle $a$
  // d o n t pass by reference for p1, so r and p1 can be the same
  p1 = p1-p0;
  r.x = cos(a)*p1.x-sin(a)*p1.y;
  r.y = sin(a)*p1.x+cos(a)*p1.y;
  r = r+p0;
}

int CAndC(pt o1, dbl r1, pt o2, dbl r2, pt& q1, pt& q2){
  // returns: $-1$ if no intersection or infinite intersection
  //           $1$ if only one point

  dbl r=dist(o1,o2);
  if(r1<r2) { swap(o1,o2); swap(r1,r2); }
  if(r<EPS) return(-1);
  if(r>r1+r2+EPS) return(-1);
  if(r<r1-r2-EPS) return(-1);
  pt v = o2-o1; v.normalize();
  q1 = o1+v*r1;

  if(fabs(r-r1-r2)<EPS || fabs(r+r2-r1)<EPS){
    q2=q1;
    return(1);
  }
  dbl a=angle(r2, r, r1);
  q2=q1;
  rotate(o1, q1, a, q1);
  rotate(o1, q2, -a, q2);
  return 0;
}

int pAndPoly(vector<pt> pv, pt p){
  // returns: $1$ if $p$ is in pv, $0$ outside; $-1$ on the polygon
  int i, j;
  int n=pv.size();
  pv.push_back(pv[0]);
  for(i=0;i<n;i++)
    if(pAndSeg(pv[i], pv[i+1], p)==1) return(-1);
  for(i=0;i<n;i++)pv[i] = pv[i]-p;

  p.x=p.y=0.0;
  dbl a, y;

  while(1){
    a=(dbl)rand()/10000.00;
    j=0;
    for(i=0;i<n;i++){
      rotate(p, pv[i], a, pv[i]);
      if(fabs(pv[i].x)<EPS) j=1;
    }
    if(j==0){
      pv[n]=pv[0];
      j=0;
      for(i=0;i<n;i++) if(pv[i].x*pv[i+1].x < -EPS){
        y=pv[i+1].y-pv[i+1].x*(pv[i].y-pv[i+1].y)/(pv[i].x-pv[i+1].x);
        if (y>0) j++;
      }
      return(j%2);
    }
  }
  return 1;
}




void reflect(pt& p1, pt& p2, pt p3, pt& r){
  // $p_1 \rightarrow p_2$ line, reflect $p_3$ to get $r$.
  if(dist(p1, p3)<EPS) {r=p3; return;}
  dbl a=angle(p1, p2, p3);
  r=p3;
  rotate(p1, r, -2.0*a, r);
}

//points where the tangents touch the circle
template<class pt>
pair<pt,pt> circleTangents(const pt &p, const pt &c, dbl r) {
  pt a = p-c;
  dbl x = r*r/a.dist2(), y = sqrt(x-x*x);
  return make_pair(c+a*x+a.perp()*y, c+a*x-a.perp()*y);
//P perp() const { return P(-y, x); } // rotates +90 degrees
}

//circle intersection
bool circleIntersection(pt a, pt b, dbl r1, dbl r2,
    pair<pt, pt>* out) {
  pt delta = b - a;
  assert(delta.x || delta.y || r1 != r2);
  if (!delta.x && !delta.y) return false;
```

```
  dbl r = r1 + r2, d2 = delta.dist2();
  dbl p = (d2 + r1*r1 - r2*r2) / (2.0 * d2);
  dbl h2 = r1*r1 - p*p*d2;
  if (d2 > r*r || h2 < 0) return false;
  pt mid = a + delta*p, per = delta.perp() * sqrt(h2 / d2);
  *out = {mid + per, mid - per};
  return true;
}
```

6.2 Polygons

```

dbl area(vector<pt> &vin){ //not necessary convex
  int n = vin.size();
  dbl ret = 0.0;
  for(int i = 0; i < n; i++)
  ret += trap(vin[i], vin[(i+1)%n]);
  return fabs(ret);
}

dbl peri(vector<pt> &vin){ //not necessary convex
  int n = vin.size();
  dbl ret = 0.0;

  for(int i = 0; i < n; i++)
    ret += dist(vin[i], vin[(i+1)%n]);
  return ret;
}

dbl heronArea(dbl ab, dbl bc, dbl ca) {
  dbl s = 0.5 * ab+bc+ca;
  return sqrt(s) * sqrt(s - ab) * sqrt(s - bc) * sqrt(s - ca);
}

dbl height(pt a, pt b, pt c){
  // height from $a$ to the line $b\rightarrow c$
  dbl s3 = dist(c, b);
  dbl ar=triarea(a,b,c);
  return(2.0*ar/s3);
}

 // CONVEX HULL
int sideSign(pt& p1, pt& p2, pt& p3){
  // which side is $p_3$ to the line $p_1\rightarrow p_2$? return: $1$ left, $0$ on, $-1$ right
  dbl sg = (p1.x-p3.x)*(p2.y-p3.y)-(p1.y - p3.y)*(p2.x-p3.x);
  if(fabs(sg)<EPS) return 0;
  if(sg>0) return 1;
  return -1;
}

bool better(pt& p1,pt& p2,pt& p3){
  // used by convec hull: from $p_3$, if $p_1$ is better than $p_2$
  dbl sg = (p1.y - p3.y)*(p2.x-p3.x)-(p1.x-p3.x)*(p2.y-p3.y);
  //watch range of the numbers
  if(fabs(sg)<EPS){
    if(dist(p3,p1) > dist(p3,p2))return true;
    else return false;
  }
  if(sg<0) return true;
  return false;
}

void convHull(vector<pt> vin, vector<pt>& vout){
  //vin is not pass by reference, since we will rotate it
  vout.clear();
  int n=vin.size();
  sort(vin.begin(),vin.end());
  pt stk[MAX_SIZE];
  int pstk, i;
  //hopefully more than 2 points
  stk[0] = vin[0]; stk[1] = vin[1];
  pstk = 2;
  for(i=2; i<n; i++){
    if(dist(vin[i], vin[i-1])<EPS) continue;
    while(pstk > 1 && better(vin[i], stk[pstk-1], stk[pstk-2]))
    pstk--;
    stk[pstk] = vin[i];
    pstk++;
  }

  for(i=0; i<pstk; i++) vout.push_back(stk[i]);
  for(i=0; i<n; i++){ //turn 180 degree
    vin[i].y = -vin[i].y;
    vin[i].x = -vin[i].x;
  }

  sort(vin.begin(), vin.end());

  stk[0] = vin[0];
  stk[1] = vin[1];
  pstk = 2;
  for(i=2; i<n; i++){
    if(dist(vin[i], vin[i-1])<EPS) continue;
    while(pstk > 1 && better(vin[i], stk[pstk-1], stk[pstk-2]))
      pstk--;
    stk[pstk] = vin[i];
    pstk++;
  }
  for(i=1; i<pstk-1; i++){
    stk[i].x= -stk[i].x; //d o n t  forget rotate 180 d back.
    stk[i].y= -stk[i].y;
    vout.push_back(stk[i]);
  }
}

int isConvex(vector<pt>& v){
  // return: $0$ !convex, $1$ convex: $2$ convex with unnecesary pts,
  // does not work if the poly is self intersecting, compute
  //  convex hull of v, and see if both have the same area
  int i,j,k;
  int c1=0; int c2=0; int c0=0;
  int n=v.size();
  for(i=0;i<n;i++){
    j=(i+1)%n;
    k=(j+1)%n;
    int s=sideSign(v[i], v[j], v[k]);
    if(s==0) c0++;
    if(s>0) c1++;
    if(s<0) c2++;
  }
  if(c1 && c2) return 0;
  if(c0) return 2;
  return 1;
}

// polygon cut
vector<pt> polygonCut(const vector<pt>& poly, pt s, pt e) {
  vector<pt> res;
  for(int i=0;i<(int)poly.size();i++){
    pt cur = poly[i], prev = i ? poly[i-1] : poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0)) {
      res.emplace_back();
      intersection(s, e, cur, prev, res.back());
    }
    if (side)
      res.push_back(cur);
  }
  return res;
}
```

```
























//
```