Team reference - UTP - 0x7DD

Universidad Tecnológica de Pereira

September 14, 2014

Contents

1 Data structures

1.1 Fenwick Tree

```
public static class FenwickTree {
    long[] fenwickTree; int tam;
    public FenwickTree(int t) {
        fenwickTree = new long[t]; tam = t;
    }
    long query(int a, int b) {
        if (a == 0) {
            long sum = 0;
            for (; b >= 0; b = (b & (b + 1)) - 1)
              sum += fenwickTree[b];
            return sum;
        }
        else { return (query(0, b) - query(0, a - 1));}
    }
    void increase(int k, long inc)
    { for (; k < tam; k |= k + 1) fenwickTree[k] += inc;}
    void increaseRange(int a, int b, long val)
    {increase(a, val); increase(b + 1, -val);}
}
```

1.2 SegmentTree RMQ

```
static class SegmentTree {
    long[] M;
    public SegmentTree(int size) {
        M = new long[size * 4 + 4];
        Arrays.fill(M, Long.MAX_VALUE);
    }
    //it's initially called update(1, 0, size - 1, pos, value)
    void update(int node, int b, int e, int pos, long value) {
        //if the current interval doesn't intersect
        //the updated position return -1
        if (pos > e || pos < b)
            return;
        //if the current interval is the updated position
        //then update it
        if (b == e)
        {   M[node] = value; return; }
        update(2 * node, b, (b + e) / 2, pos, value);
        update(2 * node + 1, (b + e) / 2 + 1, e, pos, value);
        //update current value after updating childs
        M[node] = Math.min(M[2 * node], M[2 * node + 1]);
    }
    //it's initially called query(1, 0, size - 1, i, j)
    long query(int node, int b, int e, int i, int j) {
        long p1, p2;
        //if the current interval doesn't intersect
        //the query interval return Long.MAX_VALUE
        if (i > e || j < b)
            return Long.MAX_VALUE;
        //if the current interval is completely included in
        //the query interval return the value of this node
        if (b >= i && e <= j)
            return M[node];
        //compute the value from
        //left and right part of the interval
        p1 = query(2 * node, b, (b + e) / 2, i, j);
        p2 = query(2 * node + 1, (b + e) / 2 + 1, e, i, j);
        //join them to generate result
        long tmp = Math.min(p1, p2);
        return tmp;
    }
}
```

1.3 RMQ Sparse table

```
// Preprocess
for (int i = 0; i < n; ++i) M[i][0] = i;
for (int j = 1, p = 2, q = 1; p <= n; ++j, p <<= 1, q <<= 1)
  for (int i = 0; i + p - 1 < n; ++i) {
    long long a = M[i][j - 1], b = M[i + q][j - 1];
    M[i][j] = nums[a] <= nums[b] ? a : b;
  }

// query in interval [b, e] (inclusive)
int k = log2(e - b + 1);
```

```
long long a = M[b][k], a2 = M[e + 1 - (1<<k)][k];
int idx = nums[a] <= nums[a2] ? a : a2;
```

1.4 Splay Tree + lazy propagation

```
struct node{
  node *left, *right, *parent;
  int cur, vset, size, c[27]; // current character, value to set, size,
      ans for i-th character at c[i].
  int rev, set; // Values for propagation, reverse and set operations.
  node (int k) : cur(k), left(0), right(0), parent(0) , rev(0), set(0) {
    c[k] = 1;
    size = 1;
  }
  void set_val(int a) {
    cur = vset = a;
    set = 1;
    memset(c, 0, sizeof c);
    c[a] = size;
  }
  void reverse() {
    rev ^= 1;
    swap(left, right);
  }
  void update() {
    size = 1;
    memset(c, 0, sizeof c);
    c[cur] = 1;
    if (left) {
      size += left->size;
      for (int i = 0; i < 26; ++i) c[i] += left->c[i];
    }
    if (right) {
      size += right->size;
      for (int i = 0; i < 26; ++i) c[i] += right->c[i];
    }
  }
  void propagate() {
    if (rev) {
      rev = 0;
      if (left) left->reverse();
      if (right) right->reverse();
    }
    if (set) {
      set = 0;
      if (left) left->set_val(vset);
      if (right) right->set_val(vset);
    }
  }
};
struct splay_tree{
  node *root;
  void right_rot(node *x) {
    node *p = x->parent;
    if (x->parent = p->parent) {
      if (x->parent->left == p) x->parent->left = x;
      if (x->parent->right == p) x->parent->right = x;
    }
    if (p->left = x->right) p->left->parent = p;
    x->right = p;
    p->parent = x;
    p->update();
  }
  void left_rot(node *x) {
    node *p = x->parent;
    if (x->parent = p->parent) {
      if (x->parent->left == p) x->parent->left = x;
      if (x->parent->right == p) x->parent->right = x;
    }
    if (p->right = x->left) p->right->parent = p;
    x->left = p;
    p->parent = x;
    p->update();
  }
  void splay(node *x, node *fa = 0) {
    while( x->parent != fa and x->parent != 0) {
      node *p = x->parent;
      if (p->parent == fa)
        if (p->right == x)
          left_rot(x);
        else
          right_rot(x);
      else {
        node *gp = p->parent;
        if (gp->left == p)
          if (p->left == x)
            right_rot(x),right_rot(x);
```

```
        else
          left_rot(x),right_rot(x);
      else
        if (p->left == x)
          right_rot(x), left_rot(x);
        else
          left_rot(x), left_rot(x);
    }
  }
  x->update();
  if (fa == 0) root = x;
  else fa->update();
}
splay_tree(){ root = 0;};
node *build(int a, int b) {
  if (b < a) return 0;
  int mid = (a + b)>>1;
  node *x = new node(line[mid] - 'a');
  x->left = build(a, mid - 1);
  if (x->left) x->left->parent = x;
  x->right = build(mid + 1, b);
  if (x->right) x->right->parent = x;
  x->update();
  return x;
}
node *find(int k) {
  node *cur = root;
  while (true) {
    cur->propagate();
    if (cur->left) {
      if (cur->left->size >= k) {
        cur = cur->left;
        continue;
      }
      k -= cur->left->size;
    }
    if ( k == 1) break;
    k--;
    cur = cur->right;
  }
  return cur;
}
void set_val(int a, int b, int c) {
  node *end = find(b + 1);
  node *begin = find(a - 1);
    splay(end); splay(begin, end);
    begin->right->set_val(c);
    begin->update();
    end->update();
  }
  void reverse(int a, int b) {
    node *end = find(b + 1);
    node *begin = find(a - 1);
    splay(end); splay(begin, end);
    begin->right->reverse();
  }
  int query(int a, int b, int pos) {
    node *end = find(b + 1);
    node *begin = find(a - 1);
    splay(end); splay(begin, end);
    return begin->right->c[pos];
  }
  void good_bye(node *x) {
    if (x == 0) return;
    good_bye(x->left);
    good_bye(x->right);
    delete (x);
    x = 0;
  }
  void print(node *a) {
    if (!a) return;
    print(a->left);
    cout<<(char)(a->cur + 'a')<<" - ";
    print(a->right);
  }
};
```

2 Graphs

2.1 LCA

```
// T[i] : Parent of node i in the tree
void process3(int N, int T[MAXN], int P[MAXN][LOGMAXN]) {
  int i, j;
  for (i = 0; i < N; i++)
      for (j = 0; 1 << j < N; j++)
          P[i][j] = -1;
```

```
//the first ancestor of every node i is T[i]
  for (i = 0; i < N; i++)
      P[i][0] = T[i];
//bottom up dynamic programing
  for (j = 1; 1 << j < N; j++)
     for (i = 0; i < N; i++)
         if (P[i][j - 1] != -1)
             P[i][j] = P[P[i][j - 1]][j - 1];
}
// L[i] : level of node i (dist to root)
int query(int N, int P[MAXN][LOGMAXN], int T[MAXN], int L[MAXN], int p,
    int q) {
  int tmp, log, i;
  //if p is situated on a higher level than q then we swap them
  if (L[p] < L[q])
      tmp = p, p = q, q = tmp;
  //we compute the value of [log(L[p)]
  for (log = 1; 1 << log <= L[p]; log++);
  log--;
  //we find the ancestor of node p situated on the same level
  //with q using the values in P
  for (i = log; i >= 0; i--)
      if (L[p] - (1 << i) >= L[q])
          p = P[p][i];
  if (p == q)
      return p;
  //we compute LCA(p, q) using the values in P
  for (i = log; i >= 0; i--)
      if (P[p][i] != -1 && P[p][i] != P[q][i])
          p = P[p][i], q = P[q][i];
  return T[p];
}
```

2.2 Heavy Light Decomposition

```
// Heavy-Light Decomposition
struct TreeDecomposition {
  vector<int> g[MAXN], c[MAXN];
  int s[MAXN]; // subtree size
  int p[MAXN]; // parent id
  int r[MAXN]; // chain root id
  int t[MAXN]; // index used in segtree/bit/...
  int d[MAXN]; // depht
  int ts;     // time stamp
  void dfs(int v, int f) {
    p[v] = f;
    s[v] = 1;
    if (f != -1) d[v] = d[f] + 1;
    else d[v] = 0;
    for (int i = 0; i < g[v].size(); ++i) {
      int w = g[v][i];
      if (w != f) {
        dfs(w, v);
        s[v] += s[w];
      }
    }
  }
  void hld(int v, int f, int k) {
    t[v] = ts++;
    c[k].push_back(v);
    r[v] = k;
    int x = 0, y = -1;
    for (int i = 0; i < g[v].size(); ++i) {
      int w = g[v][i];
      if (w != f) {
        if (s[w] > x) {
          x = s[w];
          y = w;
        }
      }
    }
    if (y != -1) {
      hld(y, v, k);
    }
    for (int i = 0; i < g[v].size(); ++i) {
      int w = g[v][i];
      if (w != f && w != y) {
        hld(w, v, w);
      }
    }
  }
  void init(int n) {
    for (int i = 0; i < n; ++i) {
      g[i].clear();
    }
  }
  void add(int a, int b) {
```

```
    g[a].push_back(b);
    g[b].push_back(a);
  }
  void build() {
    ts = 0;
    dfs(0, -1);
    hld(0, 0, 0);
  }
};
```

2.3 Articulation Points

```
typedef string node;
typedef map<node, vector<node> > graph;
typedef char color;
const color WHITE = 0, GRAY = 1, BLACK = 2;
graph g;
map<node, color> colors;
map<node, int> d, low;
set<node> cameras; //articulation points
int timeCount;
// Uso: Para cada nodo u:
// colors[u] = WHITE, g[u] = Aristas salientes de u.
// Funciona para grafos no dirigidos.
void dfs(node v, bool isRoot = true){
  colors[v] = GRAY;
  d[v] = low[v] = ++timeCount;
  const vector<node> &neighbors = g[v];
  int count = 0;
  for (int i=0; i<neighbors.size(); ++i){
    if (colors[neighbors[i]] == WHITE){
      //(v, neighbors[i]) is a tree edge
      dfs(neighbors[i], false);
      if (!isRoot && low[neighbors[i]] >= d[v]){
        //current node is an articulation point
        cameras.insert(v);
      }
      low[v] = min(low[v], low[neighbors[i]]);
      ++count;
    }else{ // (v, neighbors[i]) is a back edge
      low[v] = min(low[v], d[neighbors[i]]);
    }
  }
  if (isRoot && count > 1){
    //Is root and has two neighbors in the DFS-tree
    cameras.insert(v);
  }
  colors[v] = BLACK;
}
```

2.4 Bridges

```
int visited[MP];
int prev[MP], low[MP], d[MP];
vector< vector<int> > g;
vector< pair<int,int> > bridges;
int n, ticks;
void dfs(int u){
  visited[u] = true;
  d[u] = low[u] = ticks++;
  for (int i=0; i<g[u].size(); ++i){
    int v = g[u][i];
    if (prev[u] != v){
      if(!visited[v]){
        prev[v] = u;
        dfs(v);
        if (d[u] < low[v]){
          bridges.push_back(make_pair(min(u,v),max(u,v)));
        }
        low[u] = min(low[u], low[v]);
      }else{
        low[u] = min(low[u], d[v]);
      }
    }
  }
}
/**Example of use **/
memset(visited,false,sizeof(visited));
memset(prev,-1,sizeof(prev));
g.assign(n, vector<int>());
bridges.clear();
if (n == 0){ printf("0 critical links\n"); continue; }
for (int i=0; i<n; ++i){
  int node, deg;
  scanf("%d (%d)", &node, &deg);
  g[node].resize(deg);
```

```
  for (int k=0, x; k<deg; ++k){
    scanf("%d", &x);
    g[node][k] = x;
  }
}
ticks = 0;
for (int i=0; i<n; ++i){
  if (!visited[i]){
    dfs(i);
  }
}
sort(bridges.begin(), bridges.end());
printf("%d critical links\n", bridges.size());
foreach(p, bridges)
  printf("%d - %d\n", p->first, p->second);
```

2.5 Stable Marriage

```
int N,pref_men[MAX_N][MAX_N],pref_women[MAX_N][MAX_N];
int inv[MAX_N][MAX_N],cont[MAX_N],wife[MAX_N],husband[MAX_N];
void stable_marriage(){
    for(int i = 0 ; i < N ; i++)
        for(int j = 0;j < N; j++)
            inv[i][pref_women[i][j]] = j;

    fill(cont,cont+N,0);
    fill(wife,wife+N,-1);
    fill(husband,husband+N,-1);
    queue<int> Q;
    for(int i = 0; i < N; i++) Q.push(i);
    int m,w;
    while(!Q.empty()){
        m = Q.front();
        w = pref_men[m][cont[m]];
        if(husband[w] == -1){
            wife[m] = w;
            husband[w] = m;
            Q.pop();
        }else{
            if( inv[w][m] < inv[w][husband[w]] ){
                wife[m] = w;
                wife[husband[w]] = -1;
                Q.pop();
                Q.push(husband[w]);
                husband[w] = m;
            }
        }
        cont[m]++;
    }
}
```

2.6 SCC - Tarjan

```
vector<int> g[MAXN];
int d[MAXN], low[MAXN], scc[MAXN];
bool stacked[MAXN];
stack<int> s;
int ticks, current_scc;
void tarjan(int u){
  d[u] = low[u] = ticks++;
  s.push(u);
  stacked[u] = true;
  const vector<int> &out = g[u];
  for (int k=0, m=out.size(); k<m; ++k){
    const int &v = out[k];
    if (d[v] == -1){
      tarjan(v);
      low[u] = min(low[u], low[v]);
    }else if (stacked[v]){
      low[u] = min(low[u], low[v]);
    }
  }
  if (d[u] == low[u]){
    int v;
    do{
      v = s.top();
      s.pop();
      stacked[v] = false;
      scc[v] = current_scc;
    }while (u != v);
    current_scc++;
  }
}
```

2.7 Dinnic

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).
const int INF = 2000000000;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct Dinic {
  int N;
  vector<vector<Edge> > G;
  vector<Edge *> dad;
  vector<int> Q;

  Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }

  long long BlockingFlow(int s, int t) {
    fill(dad.begin(), dad.end(), (Edge *) NULL);
    dad[s] = &G[0][0] - 1;

    int head = 0, tail = 0;
    Q[tail++] = s;
    while (head < tail) {
      int x = Q[head++];
      for (int i = 0; i < G[x].size(); i++) {
        Edge &e = G[x][i];
        if (!dad[e.to] && e.cap - e.flow > 0) {
          dad[e.to] = &G[x][i];
          Q[tail++] = e.to;
        }
      }
    }
    if (!dad[t]) return 0;

    long long totflow = 0;
    for (int i = 0; i < G[t].size(); i++) {
      Edge *start = &G[G[t][i].to][G[t][i].index];
      int amt = INF;
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
        if (!e) { amt = 0; break; }
        amt = min(amt, e->cap - e->flow);
      }
      if (amt == 0) continue;
      for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
        e->flow += amt;
        G[e->to][e->index].flow -= amt;
      }
      totflow += amt;
    }
    return totflow;
  }

  long long GetMaxFlow(int s, int t) {
    long long totflow = 0;
    while (long long flow = BlockingFlow(s, t))
      totflow += flow;
    return totflow;
  }
};
```

2.8 Max Flow - Push Relabel

```
static class Edge {
    int from, to, index;
    long cap, flow;

    Edge(int fromi, int toi, long capi, long flowi, int indexi) {
```

```
        from = fromi; to = toi; cap = capi; flow = flowi;
        index = indexi;
    }
}

static class PushRelabel {
    int N; int ans; boolean[] active;
    ArrayList <Edge> [] G; long[] excess; int[] dist, count;
    ArrayDeque <Integer> Q = new ArrayDeque <Integer> ();

    PushRelabel(int N1) {
        N = N1; G = new ArrayList[N]; active = new boolean[N];
        for(int i = 0; i < N; i++)
            G[i] = new ArrayList <Edge> ();
        excess = new long[N]; dist = new int[N];
        count = new int[2 * N];
    }

    void AddEdge(int from, int to, int cap) {
        int cambio = from == to ? 1 : 0;
        G[from].add(new Edge(from, to, cap, 0, G[to].size() + cambio));
        G[to].add(new Edge(to, from, 0, 0, G[from].size() - 1));
    }

    void Enqueue(int v) {
        if (!active[v] && excess[v] > 0){active[v] = true; Q.add(v);}
    }

    void Push(Edge e) {
        long amt = Math.min(excess[e.from], e.cap - e.flow);
        if(dist[e.from] <= dist[e.to] || amt == 0) return;
        e.flow += amt; G[e.to].get(e.index).flow -= amt;
        excess[e.to] += amt; excess[e.from] -= amt; Enqueue(e.to);
    }

    void Gap(int k) {
        for(int v = 0; v < N; v++) {
            if(dist[v] < k) continue;
            count[dist[v]]--; dist[v] = Math.max(dist[v], N + 1);
            count[dist[v]]++; Enqueue(v);
        }
    }

    void Relabel(int v) {
        count[dist[v]]--; dist[v] = 2 * N;
        for (Edge e : G[v])
            if (e.cap - e.flow > 0)
                dist[v] = Math.min(dist[v], dist[e.to] + 1);
        count[dist[v]]++; Enqueue(v);
    }

    void Discharge(int v) {
        for(Edge e : G[v]) { if(excess[v] <= 0) break; Push(e);}
        if(excess[v] > 0)
            {if(count[dist[v]] == 1) Gap(dist[v]); else Relabel(v);}
    }

    long GetMaxFlow(int s, int t) {
        count[0] = N - 1; count[N] = 1; dist[s] = N;
        active[s] = active[t] = true;
        for (Edge e : G[s]) { excess[s] += e.cap; Push(e); }
        while (!Q.isEmpty()) { int v = Q.poll(); active[v] = false;
                               Discharge(v);}
        long totflow = 0;
        for (Edge e : G[s]) totflow += e.flow; return totflow;
    }
}
```

2.9 Mincost matching

```
double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
  int n = int(cost.size());

  // construct dual feasible solution
  VD u(n);
  VD v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
  }

  // construct primal solution satisfying complementary slackness
  Lmate = VI(n, -1);
  Rmate = VI(n, -1);
```

```
int mated = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    if (Rmate[j] != -1) continue;
    if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
      Lmate[i] = j;
      Rmate[j] = i;
      mated++;
      break;
    }
  }
}

VD dist(n);
VI dad(n);
VI seen(n);

// repeat until primal solution is feasible
while (mated < n) {

  // find an unmatched left node
  int s = 0;
  while (Lmate[s] != -1) s++;

  // initialize Dijkstra
  fill(dad.begin(), dad.end(), -1);
  fill(seen.begin(), seen.end(), 0);
  for (int k = 0; k < n; k++)
    dist[k] = cost[s][k] - u[s] - v[k];

  int j = 0;
  while (true) {

    // find closest
    j = -1;
    for (int k = 0; k < n; k++) {
      if (seen[k]) continue;
      if (j == -1 || dist[k] < dist[j]) j = k;
    }
    seen[j] = 1;

    // termination condition
    if (Rmate[j] == -1) break;

    // relax neighbors
    const int i = Rmate[j];
    for (int k = 0; k < n; k++) {
      if (seen[k]) continue;
      const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
      if (dist[k] > new_dist) {
        dist[k] = new_dist;
        dad[k] = j;
      }
    }
  }

  // update dual variables
  for (int k = 0; k < n; k++) {
    if (k == j || !seen[k]) continue;
    const int i = Rmate[k];
    v[k] += dist[k] - dist[j];
    u[i] -= dist[k] - dist[j];
  }
  u[s] += dist[j];

  // augment along path
  while (dad[j] >= 0) {
    const int d = dad[j];
    Rmate[j] = Rmate[d];
    Lmate[Rmate[j]] = j;
    j = d;
  }
  Rmate[j] = s;
  Lmate[s] = j;

  mated++;
 }

 double value = 0;
 for (int i = 0; i < n; i++)
   value += cost[i][Lmate[i]];

 return value;
}
```

2.10 Min cost - Max flow

```
const L INF = numeric_limits<L>::max() / 4;
```

```
struct MinCostMaxFlow {
  int N;
  VVL cap, flow, cost;
  VI found;
  VL dist, pi, width;
  VPII dad;

  MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

  void AddEdge(int from, int to, L cap, L cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
  }

  void Relax(int s, int k, L cap, L cost, int dir) {
    L val = dist[s] + pi[s] - pi[k] + cost;
    if (cap && val < dist[k]) {
      dist[k] = val;
      dad[k] = make_pair(s, dir);
      width[k] = min(cap, width[s]);
    }
  }

  L Dijkstra(int s, int t) {
    fill(found.begin(), found.end(), false);
    fill(dist.begin(), dist.end(), INF);
    fill(width.begin(), width.end(), 0);
    dist[s] = 0;
    width[s] = INF;

    while (s != -1) {
      int best = -1;
      found[s] = true;
      for (int k = 0; k < N; k++) {
        if (found[k]) continue;
        Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
        Relax(s, k, flow[k][s], -cost[k][s], -1);
        if (best == -1 || dist[k] < dist[best]) best = k;
      }
      s = best;
    }
    for (int k = 0; k < N; k++)
      pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
  }

  pair<L, L> GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
    while (L amt = Dijkstra(s, t)) {
      totflow += amt;
      for (int x = t; x != s; x = dad[x].first) {
        if (dad[x].second == 1) {
          flow[dad[x].first][x] += amt;
          totcost += amt * cost[dad[x].first][x];
        } else {
          flow[x][dad[x].first] -= amt;
          totcost -= amt * cost[x][dad[x].first];
        }
      }
    }
    return make_pair(totflow, totcost);
  }
};
```

3 Math

3.1 Chinese remainder theorem

```
void extended_euclid(ll a, ll b, ll &x, ll &y, ll &g) {
    x = 0; y = 1; g = b;
    ll m, n, q, r;
    for (ll u=1, v=0; a != 0; g=a, a=r) {
        q = g / a; r = g % a;
        m = x-u*q; n = y-v*q;
        x=u; y=v; u=m; v=n;
    }
}
/*****************************
* Find z such that         *
* z % x[i] = a[i] for all i. *
*****************************/
ll chinese_remainder_theorem(vector<ll> ns, vector<ll> as){
    int k = ns.size();
```

```
    ll N = 1, x = 0, r, s, g;
    for (int i = 0; i < k; ++i) N *= ns[i];
    for (int i = 0; i < k; ++i) {
        extended_euclid(ns[i], N/ns[i], r, s, g);
        x += as[i]*s*(N/ns[i]);
        x %= N;
    }
    if (x < 0) x += N;
    return x;
}
```

3.2 utilities

```
// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}
// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
  return solutions;
}
// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}
// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
  int d = gcd(a,b);
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}
```

3.3 Euler Totient

```
for (int i = 0; i < MP; ++i)phi[i] = i;
for (int i = 0; primes[i] <= 5000000; ++i) {
    phi[primes[i]] = primes[i] - 1;
    for (i64 j = 2 * primes[i]; j <= 5000000; j += primes[i]) {
        phi[j] = phi[j] * (primes[i]-1);
        phi[j] = phi[j] / primes[i];
    }
}
```

4 Strings

4.1 Z-Algorithm

```
vector<int> compute_z(const string &s){
  int n = s.size();
  vector<int> z(n,0);
  int l,r;
  r = l = 0;
  for(int i = 1; i < n; ++i){
    if(i > r) {
      l = r = i;
      while(r < n and s[r - l] == s[r])r++;
      z[i] = r - l;r--;
    }else{
      int k = i-l;
      if(z[k] < r - i +1) z[i] = z[k];
```

```
      else {
        l = i;
        while(r < n and s[r - l] == s[r])r++;
        z[i] = r - l;r--;
      }
    }
  }
  return z;
}
```

4.2 KMP

```
vector<int> compute_prefix_function(string p){
    vector<int> pi(p.size());
    pi[0]=-1;
    int k=-1;
    for(int i=1;i<p.size();i++){
        while(k>=0 && p[k+1]!=p[i]) k=pi[k];
        if (p[k+1]==p[i])k++;
        pi[i]=k;
    }
    return pi;
}

int KMP_Matcher(string p,string t){
    vector<int> pi=compute_prefix_function(p);
    int q=-1;
    int last = q;
    for(int i=0;i<t.size();i++){
        while(q>=0 && p[q+1]!=t[i]) q=pi[q];
        if (p[q+1]==t[i]) q++;
        last = q;
        if (q==p.size() - 1){
            q=pi[q];
        }
    }
    return last;
}
```

4.3 Aho Corasik

```
// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 6 * 50 + 10;

// Number of characters in the alphabet.
const int MAXC = 26;

// Output for each state, as a bitwise mask.
// Bit i in this mask is on if the keyword with index i
// appears when the machine enters this state.
int out[MAXS];

// Used internally in the algorithm.
int f[MAXS]; // Failure function
int g[MAXS][MAXC]; // Goto function, or -1 if fail.

// Builds the string matching machine.
//
// words - Vector of keywords. The index of each keyword is
// important:
// "out[state] & (1 << i)" is > 0 if we just found
// word[i] in the text.
// lowestChar - The lowest char in the alphabet.
// Defaults to 'a'.
// highestChar - The highest char in the alphabet.
// Defaults to 'z'.
// "highestChar - lowestChar" must be <= MAXC,
// otherwise we will access the g matrix outside
// its bounds and things will go wrong.
//
// Returns the number of states that the new machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(const vector<string> &words,
                         char lowestChar = 'a',
                         char highestChar = 'z') {
    memset(out, 0, sizeof out);
    memset(f, -1, sizeof f);
    memset(g, -1, sizeof g);

    int states = 1; // Initially, we just have the 0 state

    for (int i = 0; i < words.size(); ++i) {
        const string &keyword = words[i];
        int currentState = 0;
        for (int j = 0; j < keyword.size(); ++j) {
```

```
        int c = keyword[j] - lowestChar;
        if (g[currentState][c] == -1) {
            // Allocate a new node
            g[currentState][c] = states++;
        }
        currentState = g[currentState][c];
    }
    // There's a match of keywords[i] at node currentState.
    out[currentState] |= (1 << i);
}

// State 0 should have an outgoing edge for all characters.
for (int c = 0; c < MAXC; ++c) {
    if (g[0][c] == -1) {
        g[0][c] = 0;
    }
}

// Now, let's build the failure function
queue<int> q;
// Iterate over every possible input
for (int c = 0; c <= highestChar - lowestChar; ++c) {
    // All nodes s of depth 1 have f[s] = 0
    if (g[0][c] != -1 and g[0][c] != 0) {
        f[g[0][c]] = 0;
        q.push(g[0][c]);
    }
}
while (q.size()) {
    int state = q.front();
    q.pop();
    for (int c = 0; c <= highestChar - lowestChar; ++c) {
        if (g[state][c] != -1) {
            int failure = f[state];
            while (g[failure][c] == -1) {
                failure = f[failure];
            }
            failure = g[failure][c];
            f[g[state][c]] = failure;

            // Merge out values
            out[g[state][c]] |= out[failure];
            q.push(g[state][c]);
        }
    }
    }

    return states;
}

int findNextState(int currentState, char nextInput,
                                    char lowestChar = 'a') {
    int answer = currentState;
    int c = nextInput - lowestChar;
    while (g[answer][c] == -1) answer = f[answer];
    return g[answer][c];
}


// How to use this algorithm:
//
// buildMatchingMachine(keywords, 'a', 'z');
// int currentState = 0;
// for (int i = 0; i < text.size(); ++i) {
//   currentState = findNextState(currentState, text[i], 'a');
//
//   Nothing new, let's move on to the next character.
//   if (out[currentState] == 0) continue;
//
//   for (int j = 0; j < keywords.size(); ++j) {
//     if (out[currentState] & (1 << j)) {
//       //Matched keywords[j]
//       cout << "Keyword " << keywords[j]
//       << " appears from "
//       << i - keywords[j].size() + 1
//       << " to " << i << endl;
//     }
//   }
// }
//
// The output of this program is:
//
// Keyword his appears from 1 to 3
// Keyword he appears from 4 to 5
// Keyword she appears from 3 to 5
// Keyword hers appears from 4 to 7
```

4.4 Suffix array

```
struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P;
  vector<pair<pair<int,int>,int> > M;

  SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L,
      0)), M(L) {
    for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
    for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
      P.push_back(vector<int>(L, 0));
      for (int i = 0; i < L; i++)
        M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ?
            P[level-1][i + skip] : -1000), i);
      sort(M.begin(), M.end());
      for (int i = 0; i < L; i++)
        P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ?
            P[level][M[i-1].second] : i;
    }
  }

  vector<int> GetSuffixArray() { return P.back(); }

  // returns the length of the longest common prefix of s[i...L-1] and
      s[j...L-1]
  int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
      if (P[k][i] == P[k][j]) {
        i += 1 << k;
        j += 1 << k;
        len += 1 << k;
      }
    }
    return len;
  }
};
```

5 Geometry

5.1 Picks theorem

$$Area = B/2 + I - 1$$

$Area$: Area of a polygon, B: Lattice points in the boundary, I: Lattice points inside.

5.2 Cross product

$$U \times W = (u_2w_3 - u_3w_2, u_3w_1 - u_1w_3, u_1w_2 - u_2w_1)$$

5.3 Rotation

CCW Rotation of (x, r) around the origin.

$$(x_r, y_r) = (x * \cos\theta - y * \sin\theta, x * \sin\theta + y * \cos\theta)$$

To rotate counterclockwise $v = (x, y, z)$ an angle θ around k (unitary vector that represents the rotation axis) use:$(Rodriguesrotationformula)$.

$$\mathbf{v}_{\text{rot}} = \mathbf{v}\cos\theta + (\mathbf{k} \times \mathbf{v})\sin\theta + \mathbf{k}(\mathbf{k} \cdot \mathbf{v})(1 - \cos\theta).$$

5.4 Cartesian coordinates from latitude and longitude

$$(x, y, z) = (r\cos\phi\cos\lambda, r\cos\phi\sin\lambda, r\sin\phi)$$

5.5 Triangles

Heron's Formula:

$$Area = \sqrt{s(s-a)(s-b)(s-c)}, s = 0.5 * perimeter$$

The radius r of the triangle's **inner circle** with area A and semiperimeter s is $r = A/s$. The radius R of the triangle's **Outer circle** with area A and sides a,b and c is $R = abc/4A$. Law of cosines:

$$c^2 = a^2 + b^2 - 2ab\cos\gamma$$

where γ denotes the angle contained between sides of lengths a and b and opposite the side of length c.

5.6 Volumes and Areas

Sphere Volume: $\frac{4}{3}\pi r^3$. Sphere Surface area: $4\pi r^2$. Lateral Surface area of a right circular cone: $LSA = \pi r\sqrt{r^2 + h^2}$.

5.7 Lines and Segments Intersection

```
point intersectionbtwlines(point p1,point p2,point p3,point p4){
    point p2_p1 = p2.sub(p1);
    point p4_p3 = p4.sub(p3);
    double den = p2_p1.cross(p4_p3);
    if (Math.abs(den)<eps) return null; //parallel or coincident
    point p1_p3 = p1.sub(p3);
    double num = p4_p3.cross(p1_p3);
    double ua = num/den;
    double num2 = p2_p1.cross(p1_p3)
    double ub = num2/den;
    return p2_p1.multbyscalar(ua).add(p1);
}
```

This function computes the intersection of the line passing through p1 and p2 with the line passing through p3 and p4. If `den` is equal to zero then the lines are parallel. If `den`, `num` and `num2` are equal to zero then the lines are coincident. If the **intersection of line segments** is required then it is only necessary to test if `ua` and `ub` lie between 0 and 1.

5.8 Circle Cirle Intersection

```
ArrayList<Point> CircleCircleIntersection(Point a, Point b,
double r, double R) {
    ArrayList<Point> ret = new ArrayList<Point>();
    double d = a.sub(b).norm();
    if ( (d > r+R+eps) || (d+Math.min(r, R) < Math.max(r, R)-eps))
        return ret;
    double x = (d*d-R*R+r*r)/(2*d);
    double y = Math.sqrt(Math.max(0.0,r*r-x*x));
    Point v = b.sub(a).multbyscalar(1 / d);
    Point xx = v.multbyscalar(x).add(a);
    Point yy = v.RotateCCW90().multbyscalar(y);
    ret.add(xx.add(yy));
    if (y > eps)
        ret.add(xx.sub(yy));
    return ret;
}
```

5.9 Half Plane Intersection

```
point[] sutherland_hodgman(point[] line, point[] poly,int flag){
    ArrayList<point> output=new ArrayList<point>();
    point S=poly[poly.length-1];
    for(int i=0;i<poly.length;i++){
        point E=poly[i];
        double ecross=line[1].sub(line[0]).cross(E.sub(line[0]));
        double scross=line[1].sub(line[0]).cross(S.sub(line[0]));
        if (Math.abs(ecross)<eps)
            output.add(E);
        else if (ecross*flag>eps){
            if (scross*flag<-eps)
                output.add(intersectionbtwlines(line[0]
                    ,line[1],S,E));
            output.add(E);
        }
        else if(scross*flag>eps)
            output.add(intersectionbtwlines(line[0],line[1],S,E));
        S=E;
    }
    point[] ret=new point[output.size()];
    return output.toArray(ret);
}
```

This function computes the intersection of a half plane with a polygon. The result is given in the same order of `poly`. The variable `flag` only can take 1.0 and -1.0 as values. If `flag` is equal to 1.0 then the halfplane considered is to the left of the directed vector `line`.

5.10 Polygon area

```
double polygonarea(point[] a, int n){
    double r=0;
    for(int i=0;i<n;i++)
        r+=a[i].cross(a[(i+1)%n]);
    return Math.abs(r/2.0);
}
```

5.11 Polygon's Centroid

```
static Point centroid(Point[] a, int n){
    Point ret = new Point(0.0, 0.0);
    double scale = 6.0 * signedpolygonarea(a, n);
    for(int i = 0; i < n; i++){
        int j = (i + 1) % n;
        Point t = a[i].add(a[j]);
        t = t.multbyscalar(a[i].cross(a[j]));
        ret = ret.add(t);
    }
    return ret.multbyscalar(1.0 / scale);
}
```

This function computes the centroid of a non-intersecting closed polygon. The function `signedpolygonarea` is equal to the `polygonarea` function but it doesn't take the absolute value of `r`.

5.12 Point in Polygon

```
boolean insideconvexpolygon(point[] p, int n, point t) {
    int mask=0;
    for(int i=0;i<n;i++){
        int z=ccw(p[i],p[(i+1)%n],t);
        if (z<0) mask |= 1;
        else if (z>0) mask |= 2;
        else if (z==0) return belongstoedge(p[i],p[(i+1)%n],t);
        if (mask==3) return false;
    }
    return true;
}

static double cross(Point2D a, Point2D b)
    {return (a.getX() * b.getY()) - (a.getY() * b.getX());}

static boolean contains(Point2D p) {
    int cnt = 0;
    for(Line2D line : lines) {
        Point2D curr = subtract(line.getP1(), p);
        Point2D next = subtract(line.getP2(), p);
        if(curr.getY() > next.getY())
        { Point2D temp = curr; curr = next; next = temp; }
        if(curr.getY() < 0 && 0 <= next.getY() &&
            cross(next, curr) >= 0) cnt++;
        if(is_point_online(line.getP1(), line.getP2(), p))
            return true;
      }
        return cnt % 2 == 1;
}
```

```
// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y -
          p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}
```

5.13 Min-distance between 3D segments

```
double distanciabtwsegments(Point[] r, Point[] s) {
    Point v = s[1].sub(s[0]);
    Point u = r[1].sub(r[0]);
    Point w = r[0].sub(s[0]);
    double a = u.dot(u), b = u.dot(v), c = v.dot(v),
```

```
    d = u.dot(w), e = v.dot(w);
    double D = a*c - b*b;
    double sc, sN, sD = D;
    double tc, tN, tD = D;
    if (D < epsilon) {
        sN = 0; sD = 1; tN = e; tD = c;
    }
    else {
        sN = (b*e - c*d);
        tN = (a*e - b*d);
        if (sN < 0) {
            sN = 0; tN = e; tD = c;
        } else if (sN > sD) {
            sN = sD; tN = e + b; tD = c;
        }
    }
    if (tN < 0) {
        tN = 0;
        if (-d < 0) {
            sN = 0;
        } else if (-d > a) {
            sN = sD;
        } else {
            sN = -d;
            sD = a;
        }
    }
    else if (tN > tD) {
        tN = tD;
        if ((-d + b) < 0) {
            sN = 0;
        } else if (-d + b > a) {
            sN = sD;
        } else {
            sN = -d + b;
            sD = a;
        }
    }
    sc = (Math.abs(sN) < epsilon ? 0 : sN / sD);
    tc = (Math.abs(tN) < epsilon ? 0 : tN / tD);
    Point dP = w.add(u.multbyscalar(sc)).sub(v.multbyscalar(tc));
    return dP.norm();
}
```

5.14 Min-distance between 3D lines

```
double distancebtwlines(Point[] r, Point[] s) {
    Point u = r[1].sub(r[0]); Point v = s[1].sub(s[0]);
    Point w = s[0].sub(r[0]);
    double a = u.dot(u), b = u.dot(v), c = v.dot(v),
    d = u.dot(w), e = v.dot(w);
    double D = a*c - b*b, sc, tc;
    if (D < epsilon) {
        sc = 0;
        tc = (b > c) ? d/b : e/c;
    } else {
        sc = (b*e - c*d) / D;
        tc = (a*e - b*d) / D;
    }
    Point dP = w.add(u.multbyscalar(sc)).sub(v.multbyscalar(tc));
    return dP.norm();
}
```

5.15 Convex Hull

```
static class cmp2 implements Comparator<point> {
    @Override
    public int compare(point f, point s) {
        double tmp=ccw(first_point,f,s);
        if (Math.abs(tmp)<eps) {
            if (f.sub(first_point).norm() < s.sub(first_point).norm())
                return -1;
            else
                return 1;
        }
        return (tmp>0)?1:-1;
    }
}


static point[] convexhull(point[] in,int n) {
    ArrayList<point> hull=new ArrayList<point>(n);
    int index = 0;
    for(int i = 1; i < n; i++)
        if (in[i].y < in[index].y ||
        (Math.abs(in[i].y - in[index].y)<eps && in[i].x < in[index].x))
```

```
            index = i;
    swap(in, 0, index);
    first_point=in[0];
    Arrays.sort(in,1,n,new cmp2());
    if (n<=3) {
        for(int i=0; i<n; i++)
            hull.add(in[i]);
        point[] a=new point[hull.size()];
        return hull.toArray(a);
    }
    // this is for a redundant CH
    int rev = n - 1;
    for(int r = n - 2; r >= 1; r--)
        if (ccw(first_point,in[r],in[rev]) == 0)
            rev = r;
        else
            break;
    if (rev != 1) reverse(in, rev, n);
    //
    hull.add(first_point);
    hull.add(in[1]);
    int top=1;
    int i=2;
    while(i<n) { // >= for a nonredundant CH
        if (top>0 && ccw(hull.get(top-1),hull.get(top),in[i]) > eps) {
            hull.remove(top);
            top=top-1;
        }
        else {
            top=top+1;
            hull.add(in[i]);
            i++;
        }
    }
    point[] a=new point[hull.size()];
    return hull.toArray(a);
}

static double ccw(point a,point b,point c) {
    return a.sub(b).cross(c.sub(b));
}
```

5.16 Convex Hull Trick

```
static class Line implements Comparable<Line>{
    long m,b;
    public Line(long mm, long bb){
        m = mm; b = bb;
    }
    long eval(long x){
        return m * x + b;
    }
    Frac getIntersection(Line o){
        if (o == null) return minf;
        return new Frac(o.b - b, m - o.m);
    }
    public int compareTo(Line o) {
        return Long.valueOf(m).compareTo(o.m);
    }
}

static class Hull{
    TreeMap<Line, Frac> lines;
    TreeMap<Frac, Line> intervals;

    public Hull(){
        lines =new TreeMap<Line, Frac>();
        intervals = new TreeMap<Frac, Line>();
    }

    long query(long x){
        Frac tmp = intervals.floorKey(new Frac(x, 1));
        Line maximal = intervals.get(tmp);
        return maximal.eval(x);
    }

    boolean checkbad(Line l1, Line l2, Line l3){
        Frac f13 = new Frac(l3.b - l1.b, l1.m - l3.m);
        Frac f12 = new Frac(l1.b - l2.b, l2.m - l1.m);
        return f13.compareTo(f12) <= 0;
    }

    void Add(long m, long b){
        Line l = new Line(m, b);
        Line floor = lines.floorKey(l);
        // is there a line in the hull with same slope?
        if (floor!= null && floor.m == m){
            // this is maximization problem so the higher the
```

```
    // y-intercept the better.
    if (floor.b < b){
        intervals.remove(lines.get(floor));
        lines.remove(floor);
    }
    else
        return;
}
Line lower = lines.lowerKey(l);
Line higher = lines.higherKey(l);
// is this line important for the hull?
if (lower != null && higher != null){
    if (checkbad(floor,l , higher))
        return;
}
// keep the invariant to the left
while(true){
    Line l2 = lines.lowerKey(l);
    if (l2 == null) break;
    Line l1 = lines.lowerKey(l2);
    if (l1 == null) break;
    if (!checkbad(l1,l2,l)) break;
    intervals.remove(lines.get(l2));
    lines.remove(l2);
}
//keep the invariant to the right
while(true){
    Line l4 = lines.higherKey(l);
    if (l4 == null) break;
    Line l5 = lines.higherKey(l4);
    if (l5 == null) break;
    if (!checkbad(l,l4,l5)) break;
    intervals.remove(lines.get(l4));
    lines.remove(l4);
}
Line forward = lines.higherKey(l);
if (forward != null){
    Frac intersection = l.getIntersection(forward);
    intervals.remove(lines.get(forward));
    lines.put(forward, intersection);
    intervals.put(intersection, forward);
}
lower = lines.lowerKey(l);
Frac intersection = l.getIntersection(lower);
lines.put(l, intersection);
        intervals.put(intersection, l);
    }
}
```

`Frac` stands for a simple fraction class, the comparison function between fractions should be defined carefully. `minf` represents minus infinity. This code represents an upper envelope (for max-queries). To build a lower envelope the comparison function between lines must be inverted and the `<` must be changed by `>` in the `Add` function.

5.17 Smallest enclosing disk

```
/*
 * Calculates the sed of a set of Points. Call initially with R = empty
     set.
 * P is the set of points in the plane. R is the set of points lying on
     the boundary of the current circle.
 */

function sed(P,R)
{
    if (P is empty or |R| = 3) then
         D := calcDiskDirectly(R)
    else
        choose a p from P randomly;
        D := sed(P - {p}, R);
        if (p lies NOT inside D) then
            D := sed(P - {p}, R u {p});
    return D;
}
```

5.18 Great circle distance

```
static double greatCircleDistance(double latitudeS, double longitudeS,
    double latitudeF, double longitudeF, double r) {
        latitudeS = Math.toRadians(latitudeS);
        latitudeF = Math.toRadians(latitudeF);
        longitudeS = Math.toRadians(longitudeS);
        longitudeF = Math.toRadians(longitudeF);
        double deltaLongitude = longitudeF - longitudeS;
        double a = Math.cos(latitudeF) * Math.sin(deltaLongitude);
        double b = Math.cos(latitudeS) * Math.sin(latitudeF);
```

```
        b -= Math.sin(latitudeS) * Math.cos(latitudeF)
         * Math.cos(deltaLongitude);
        double c = Math.sin(latitudeS) * Math.sin(latitudeF);
        c += Math.cos(latitudeS) * Math.cos(latitudeF)
         * Math.cos(deltaLongitude);
        return Math.atan(Math.sqrt(a * a + b * b) / c) * r;
}
```

6 Dp Optimizations

6.1 Divide and conquer optimization

```
static void solve(int k, int a, int b, int L, int R){
        if (b < a)
                return;
        int mid = (a + b)/2;
        int best = Integer.MAX_VALUE;
        int bestidx = -1;
        for(int i = Math.max(mid, L); i <= R; i++){
                if (dp[i + 1][k - 1] == Integer.MAX_VALUE)
                        continue;
                int value = dp[i + 1][k - 1] + some_function(mid, i);
                if (value < best){
                        best = value;
                        bestidx = i;
                }
        }
        dp[mid][k] = best;
        solve(k, a, mid - 1, L, bestidx);
        solve(k, mid + 1, b, bestidx, R);
}
```

6.2 Convex hull trick

```
struct line{
    long long a,b;

    line(){}
    line(long long _a, long long _b):
        a(_a),b(_b){}
};

long long a[MAXM],suma[MAXM];
long long dp[2][MAXM];
line H[MAXM];
int sz,pos;

bool check(line &l1, line &l2, line &l3){
    return (l3.b - l2.b) * (l1.a - l3.a) >= (l3.b - l1.b) * (l2.a - l3.a);
}

void insert(long long a, long long b){
    line l(a,b);
    while(sz >= 2 && !check(H[sz - 2],H[sz - 1],l)) --sz;
    H[sz] = l;
    ++sz;
}

long long eval(int ind, long long x){
    return H[ind].a * x + H[ind].b;
}

long long query(long long x){
    while(pos + 1 < sz && eval(pos,x) > eval(pos + 1,x)) ++pos;
    return eval(pos,x);
}
```

7 Misc

7.1 Longest Increasing subsecuence

```
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] = dad[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;
}
```

7.2 FFT

Call with $\theta = \frac{2\pi}{N}$, where N is a power of 2. For inverse transform use $\theta = -\frac{2\pi}{N}$ instead and divide every element by N. Tests:

- `[0.25,0.25,0.25,0.25] = [1,0,0,0]`
- `[-1,0,1,0] = [0,-2,0,-2]`

```
struct cpx {
  cpx(){}
  cpx(double aa):a(aa){}
  cpx(double aa, double bb):a(aa),b(bb){}
  double a;
  double b;
  double modsq(void) const {
    return a * a + b * b;
  }
  cpx bar(void) const {
    return cpx(a, -b);
  }
};
cpx operator +(cpx a, cpx b) {
  return cpx(a.a + b.a, a.b + b.b);
}
cpx operator *(cpx a, cpx b) {
  return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}
cpx operator /(cpx a, cpx b) {
  cpx r = a * b.bar();
  return cpx(r.a / b.modsq(), r.b / b.modsq());
}
cpx EXP(double theta) {
  return cpx(cos(theta),sin(theta));
}
void fft(int n, double theta, cpx* a) {
    cpx basew = EXP(theta);
    for (int m = n; m >= 2; m >>= 1) {
        int mh = m >> 1;
        cpx w = cpx(1,0);
        for (int i = 0; i < mh; i++) {
            for (int j = i; j < n; j += m) {
                int k = j + mh;
                cpx x = a[j] - a[k];
                a[j] = a[j] + a[k];
                a[k] = w * x;
            }
            w = w * basew;
        }
        theta *= 2; basew = basew*basew;
    }
    int i = 0;
    for (int j = 1; j < n - 1; j++) {
        for (int k = n >> 1; k > (i ^= k); k >>= 1);
        if (j < i) swap(a[i], a[j]);
    }
}
```

7.3 Integer sequences

7.3.1 Sums

$\sum_{k=0}^{n} k^2 = n(n+1)(2n+1)/6$
$\sum_{k=0}^{n} k^3 = n^2(n+1)^2/4$
$\sum_{k=0}^{n} k^4 = (6n^5 + 15n^4 + 10n^3 - n)/30$
$\sum_{k=0}^{n} k^5 = (2n^6 + 6n^5 + 5n^4 - n^2)/12$
$\sum_{k=0}^{n} x^k = (x^{n+1} - 1)/(x - 1)$
$\sum_{k=0}^{n} kx^k = (x - (n+1)x^{n+1} + nx^{n+2})/(x-1)^2$

7.3.2 Bell Numbers

Is the number of partitions of a set of n elements ($S = \{1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, ...\}$). They follow the recurrence $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$.

The algorithm to get the sequence produces a triangle as:

```
 1
 1  2
 2  3  5
 5  7 10 15
15 20 27 37 52
```

Where the first number of a row is the last of the previous one and each other number is produced by adding the previous number in the row by the number above it.

7.3.3 Catalan numbers

The sequence C={1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190} is called Catalan sequence. They obey the following formula:

$$C_n = \binom{2n}{n} - \binom{2n}{n+1}$$

And the following recurence relation:

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-1}$$

7.4 Combinations and permutations

Is order important?	Is repetition allowed?	Formula
Yes	Yes	n^r
No	Yes	$\frac{(n+r-1)!}{r!(n-1)!}$
Yes	No	$\frac{n!}{(n-r)!}$
No	No	$\frac{n!}{r!(n-r)!}$

7.4.1 Others

Derangements A derangement is a permutation of the elements of a set such that none of the elements appear in their original position. Number of permutations of $n = 0, 1, 2, \ldots$ elements without fixed points. Sequence: $1, 0, 1, 2, 9, 44, 265, 1854, 14833, \ldots$ Recurrence: $D_n = (n-1)(D_{n-1} + D_{n-2}) = nD_{n-1} + (-1)^n$. Corollary: number of permutations with exactly k fixed points is $\binom{n}{k} D_{n-k}$.

Double factorial Permutations of the multiset $\{1, 1, 2, 3, \ldots n, n\}$ such that for each k, all the numbers between two occurrences of k in the permutation are greater than k. $(2n-1)!! = \prod_{k=1}^{n} (2k-1)$.

Multinomial theorem $(a_1 + \cdots + a_k)^n = \sum \binom{n}{n_1, \ldots, n_k} a_1^{n_1} \ldots a_k^{n_k}$, where $n_i \geq 0$ and $\sum n_i = n$.
$\binom{n}{n_1, \ldots, n_k} = M(n_1, \ldots, n_k) = \frac{n!}{n_1! \ldots n_k!}$. $M(a, \ldots, b, c, \ldots) = M(a + \cdots + b, c, \ldots) M(a, \ldots, b)$

Luccas theorem Let $n, k, p \in naturals$ and p be a prime number. Then

$$\binom{n}{k} \equiv \prod_{j=0}^{t} \binom{n_j}{k_j} \mod p,$$

where n_j and k_j are the j-th digits of the numbers n and k in base p, respectively, and t is the length of $\max(n, m)$ in base p.

Pythagorean triples Integer solutions of $x^2 + y^2 = z^2$ All relatively prime triples are given by: $x = 2mn, y = m^2 - n^2, z = m^2 + n^2$ where $m > n, \gcd(m, n) = 1$ and $m \not\equiv n \pmod 2$. All other triples are multiples of these. Equation $x^2 + y^2 = 2z^2$ is equivalent to $(\frac{x+y}{2})^2 + (\frac{x-y}{2})^2 = z^2$.

Euler's phi function $\varphi(n) = |\{m \in N, m \le n, \gcd(m,n) = 1\}|$. $\varphi(n) = n \cdot \prod_{p|n} \left(1 - \frac{1}{p}\right)$ $\varphi(mn) = \frac{\varphi(m)\varphi(n)\gcd(m,n)}{\varphi(\gcd(m,n))}$. $\varphi(p^a) = p^{a-1}(p-1)$. $\sum_{d|n} \varphi(d) = \sum_{d|n} \varphi(\frac{n}{d}) = n$.

Euler's theorem $a^{\varphi(n)} \equiv 1 \pmod{n}$, if $\gcd(a,n) = 1$.

7.5 Simplex

```
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments. Then, call Solve(x).
struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] =
        A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D[i][n+1] =
        b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m+1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m+2; i++) if (i != r)
      for (int j = 0; j < n+2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m+2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m+1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] <
            N[s]) s = j;
      }
      if (D[x][s] >= -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] <= 0) continue;
        if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
            D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r =
                i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] <= -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m+1][n+1] < -EPS) return
          -numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] <
              N[s]) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
```

```
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
    return D[m][n+1];
  }
};
int main() {
  const int m = 4;
  const int n = 3;
  DOUBLE _A[m][n] = {
    { 6, -1, 0 },
    { -1, -5, 0 },
    { 1, 5, 1 },
    { -1, -5, -1 }
  };
  DOUBLE _b[m] = { 10, -4, 5, -5 };
  DOUBLE _c[n] = { 1, -1, 0 };

  VVD A(m);
  VD b(_b, _b + m);
  VD c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);
  LPSolver solver(A, b, c);
  VD x;
  DOUBLE value = solver.Solve(x);
  cerr << "VALUE: "<< value << endl;
  cerr << "SOLUTION:";
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
  return 0;
}
```

7.6 Primes

```
// Primes less than 1000:
//      2     3     5     7    11    13    17    19    23    29    31    37
//     41    43    47    53    59    61    67    71    73    79    83    89
//     97   101   103   107   109   113   127   131   137   139   149   151
//    157   163   167   173   179   181   191   193   197   199   211   223
//    227   229   233   239   241   251   257   263   269   271   277   281
//    283   293   307   311   313   317   331   337   347   349   353   359
//    367   373   379   383   389   397   401   409   419   421   431   433
//    439   443   449   457   461   463   467   479   487   491   499   503
//    509   521   523   541   547   557   563   569   571   577   587   593
//    599   601   607   613   617   619   631   641   643   647   653   659
//    661   673   677   683   691   701   709   719   727   733   739   743
//    751   757   761   769   773   787   797   809   811   821   823   827
//    829   839   853   857   859   863   877   881   883   887   907   911
//    919   929   937   941   947   953   967   971   977   983   991   997
// Other primes:
//    The largest prime smaller than 10 is 7.
//    The largest prime smaller than 100 is 97.
//    The largest prime smaller than 1000 is 997.
//    The largest prime smaller than 10000 is 9973.
//    The largest prime smaller than 100000 is 99991.
//    The largest prime smaller than 1000000 is 999983.
//    The largest prime smaller than 10000000 is 9999991.
//    The largest prime smaller than 100000000 is 99999989.
//    The largest prime smaller than 1000000000 is 999999937.
//    The largest prime smaller than 10000000000 is 9999999967.
//    The largest prime smaller than 100000000000 is 99999999977.
//    The largest prime smaller than 1000000000000 is 999999999989.
//    The largest prime smaller than 10000000000000 is 9999999999971.
//    The largest prime smaller than 100000000000000 is 99999999999973.
//    The largest prime smaller than 1000000000000000 is 999999999999989.
//    The largest prime smaller than 10000000000000000 is
    9999999999999937.
//    The largest prime smaller than 100000000000000000 is
    99999999999999997.
//    The largest prime smaller than 1000000000000000000 is
    999999999999999989.
```