Contents

Data Structures

Sparse Table

```
int a[N], st[LG + 1][N];
void preprocess() {
    for (int i = 1; i <= n; ++i) st[0][i] = a[i];
    for (int j = 1; j <= LG; ++j)
        for (int i = 1; i + (1 << j) - 1 <= n; ++i)
            st[j][i] = min(st[j - 1][i], st[j - 1][i +
                 (1 << (j - 1))]);
}

int query(int l, int r) {
    int k = __lg(r - l + 1);
    return min(st[k][l], st[k][r - (1 << k) + 1]);
}

//query sum:
int querySum(int l, int r) {
    int len = r - l + 1;
    int sum = 0;
    for (int j = 0; (1 << j) <= len; ++j)
        if (len >> j & 1) {
            sum = sum + st[j][l];
            l = l + (1 << j);
        }
    return sum;
}
```

Fenwick Tree

```
void update(int i, int val){
    for (; i <= n; i += i & -i) bit[i] += val;
}

int get(int i){
    int res = 0;
    for (; i; i -= i & -i) res += bit[i];
    return res;
}
```

Segment Tree

```
struct Segment_tree{
  int st[4 * N], lazy[4 * N];

    void apply(int id, int c){
        update(st[id], c);
        update(lazy[id], c);
    }

  void down(int id, int l, int r){
    int c = lazy[id]; lazy[id] = 0;
        apply(id << 1, c); apply (id << 1 | 1, c);
  }

  void build(int id, int l, int r){
    if (l == r){
      st[id] = a[l];
      return;
    }

    int mid = (l + r) >> 1;
    build(id << 1, l, mid);
    build(id << 1 | 1, mid + 1, r);

    st[id] = merge(st[id << 1], st[id << 1 | 1]);
  }

  void update(int id, int l, int r, int u, int v, int
       x){
    if (r < u || v < l) return;
    if (u <= l && r <= v){
      apply(id, x);
      return;
    }

    down(id, l, r);

    int mid = (l + r) >> 1;
    update(id << 1, l, mid, u, v, x);
    update(id << 1 | 1, mid + 1, r, u, v, x);

    st[id] = merge(st[id << 1], st[id << 1 | 1]);
  }

  int get(int id, int l, int r, int u, int v){
    if (r < u || v < l) return -INF;
    if (u <= l && r <= v) return st[id];

    down(id, l, r);

    int mid = (l + r) >> 1;
    return merge(get(id << 1, l, mid, u, v), get(id
        << 1 | 1, mid + 1, r, u, v));
  }
} ST;
```

Sigma Tree

```
struct Sigma_Tree{
    int st[2 * N];

    void init(){
        For(i, 1, n) st[i + n - 1] = a[i];
        ForD(i, n - 1, 1) st[i] = merge(st[i << 1],
            st[i << 1 | 1]);
    }

  void update(int p, int val){
    p += n - 1;
    st[p] = val;
    for (; p > 1; p >>= 1) st[p >> 1] = merge(st[p],
        st[p ^ 1]);
  }

    int get(int l, int r){
        int res = 0;
        for (l += n - 1, r += n - 1; l <= r; l >>= 1,
             r >>= 1){
            if (l & 1) res = merge(res, st[l++]);
            if (!(r & 1)) res = merge(res, st[r--]);
        }
        return res;
    }
} ST;
```

Persistent Segment Tree

```
struct Node {
    int left, right; // ID of left child & right
        child
    long long ln; // Max value of node
    Node() {}
    Node(long long ln, int left, int right) : ln(ln),
         left(left), right(right) {}
```

```
} it[N]; // Each node has a position in this array,
     called ID
int nNode;

int ver[N]; // ID of root in each version

// Update max value of a node
inline void refine(int cur) {
    it[cur].ln = max(it[it[cur].left].ln, it[it[cur].
        right].ln);
}

// Update a range, and return new ID of node
int update(int l, int r, int u, int x, int oldId) {
    if (l == r) {
        ++nNode;
        it[nNode] = Node(x, 0, 0);
        return nNode;
    }

    int mid = (l + r) >> 1;
    int cur = ++nNode;

    if (u <= mid) {
        it[cur].left = update(l, mid, u, x, it[oldId
            ].left);
        it[cur].right = it[oldId].right;
        refine(cur);
    }
    else {
        it[cur].left = it[oldId].left;
        it[cur].right = update(mid+1, r, u, x, it[
            oldId].right);
        refine(cur);
    }

    return cur;
}

// Get max of range. Same as usual IT
int get(int nodeId, int l, int r, int u, int v) {
    if (v < l || r < u) return -1;
    if (u <= l && r <= v) return it[nodeId].ln;

    int mid = (l + r) >> 1;
    return max(get(it[nodeId].left, l, mid, u, v),
        get(it[nodeId].right, mid+1, r, u, v));
}


// When update:
    ++nVer;
    ver[nVer] = update(1, n, u, x, ver[nVer-1]);

// When query:
    res = get(ver[t], 1, n, u, v);
```

Hash Map

```
//faster than unordered_map
struct hash_map {
    const static int SZ = 2e4 + 9;
    int nxt[SZ >> 3], val[SZ >> 3];
    int key[SZ >> 3];
    int h[SZ + 5], cnt;
    vector<int>vec;

    void clear(){
        for (int i : vec) h[i] = 0;
        for (int i = 1; i <= cnt; i++)
            val[i] = nxt[i] = 0, key[i] = 0;

        vec.clear();
        cnt = 0;
    }

    int hash(int u) {
        return u % SZ;
    }

    int &operator[](int u) {
        int x = hash(u);

        for (int i = h[x]; i; i = nxt[i])
            if (key[i] == u) return val[i];

        if (!h[x]) vec.push_back(x);

        ++cnt;
        key[cnt] = u;
        val[cnt] = 0;
        nxt[cnt] = h[x];
        h[x] = cnt;
        return val[cnt];
    }

    int qry(int u) {
        int x = hash(u);
        for (int i = h[x]; i; i = nxt[i])
            if (key[i] == u) return val[i];

        return 0;
    }
} hs;
```

String

Trie 1

```
struct node{
  node *g[26];
  node(){
    rep(i, 26) g[i] = NULL;
  }
} *root = new node();

void Insert(string s){
  node *p = root;
  for (char t: s){
    if (p->g[t - 'a'] == NULL)
      p->g[t - 'a'] = new node();

    p = p->g[t - 'a'];
  }
}
```

Trie 2

```
int nNode = 0;
int g[N][26];

void Insert(string s){
    int p = 0;
    for (char t: s){
        if (!g[p][t - 'a']) g[p][t - 'a'] = ++nNode;
        p = g[p][t - 'a'];
    }
}
```

Hash

```
ll getHashT(int i, int j) {
    return (hashT[j] - mul(hashT[i - 1], POW[j - i +
        1]) + MOD) % MOD;
}

// Precalculate base^i
for (int i = 1; i <= lenT; i++)
    POW[i] = (POW[i - 1] * base) % MOD;

// Calculate hash value of T[1..i]
for (int i = 1; i <= lenT; i++)
    hashT[i] = (hashT[i - 1] * base + (T[i] - 'a' +
        1)) % MOD;
```

KMP

```
//prefix function: length of the longest prefix of
    the substring s[1..i] that is also a suffix of
    this same substring
int k = 0;
For(i, 2, n){ //1-indexed
    while (k && s[k + 1] != s[i]) k = kmp[k];
    kmp[i] = (s[k + 1] == s[i]) ? ++k : 0;
}
```

Manacher

```
vector<int> manacher_odd(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    int l = 0, r = 1;
    for(int i = 1; i <= n; i++) {
        p[i] = min(r - i, p[l + (r - i)]);
        while(s[i - p[i]] == s[i + p[i]]) {
            p[i]++;
        }
        if(i + p[i] > r) {
            l = i - p[i], r = i + p[i];
        }
    }
    return vector<int>(begin(p) + 1, end(p) - 1);
}

vector<int> manacher(string s) {
    string t;
    for(auto c: s) {
        t += string("#") + c;
    }
    auto res = manacher_odd(t + "#");
    return vector<int>(begin(res) + 1, end(res) - 1);
}
```

Aho - Corasick

```
namespace Trie{
  struct Node{
    int child[26], p = -1, cnt = 0;
    char pch;
    int link = -1, go[26];
    Node(int p = -1, char ch = '#'): p(p), pch(ch){
      fill(begin(child), end(child), -1);
        fill(begin(go), end(go), -1);
    }
  };
  vector<Node> g(1);

  void add(string s){
    int v = 0;
    for (char t: s){
      int c = t - 'a';
      if (g[v].child[c] == -1){
        g[v].child[c] = g.size();
        g.emplace_back(v, t);
      }
      v = g[v].child[c];
    }
    g[v].cnt++;
  }

  int go(int v, char c);

  int get_link(int v){
    if (g[v].link == -1){
      if (!v || !g[v].p) g[v].link = 0;
      else g[v].link = go(get_link(g[v].p), g[v].pch)
          ;
    }
    return g[v].link;
  }

  int go(int v, char t){
    int c = t - 'a';
    if (g[v].go[c] == -1){
      if (g[v].child[c] != -1) g[v].go[c] = g[v].
          child[c];
      else g[v].go[c] = (v == 0) ? 0 : go(get_link(v)
          , t);
    }
    return g[v].go[c];
  }
}
```

Aho - Corasick (BFS)

```
struct trie{
  struct Node{
    Node *child[26], *link;
    int cnt = 0;
    Node(){
      cnt = 0;
      rep(i, 26) child[i] = NULL;
      link = NULL;
    }
  } *root = new Node();


  void add(string &s){
    Node* p = root;
    for (char &t: s){
      int c = t - 'a';
      if (p->child[c] == NULL) p->child[c] = new Node
          ();;
      p = p->child[c];
    }
    p->cnt++;
  }

  void AhoCorasick(){
    root->link = root;
    queue<Node*> q; q.push(root);
    while (!q.empty()){
      Node* p = q.front(); q.pop();
      rep(i, 26) if (p->child[i]){
```

```
        Node* k = p->link;
        while (k != root && k->child[i] == NULL) k =
            k->link;

        if (k->child[i] && k != p) p->child[i]->link
            = k->child[i];
        else p->child[i]->link = root;

        p->child[i]->cnt += p->child[i]->link->cnt;
        q.push(p->child[i]);
      }
    }
  }
};
```

Graph

Joint and Bridge

```
void dfs(int u, int pre) {
    int child = 0;
    num[u] = low[u] = ++timer;
    for (int v: g[u]) {
        if (v == pre) continue;
        if (!num[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] == num[v]) bridge++;
            child++;
            if (u == pre){
                if (child > 1) joint[u] = true;
            }
            else if (low[v] >= num[u]) joint[u] = true
                ;
        }
        else low[u] = min(low[u], num[v]);
    }
}
```

SCC

```
void dfs(int u) {
    num[u] = low[u] = ++timer;
    st.push(u);
    for (int v : g[u]) {
        if (!num[v]){
            dfs(v);
            low[u] = min(low[u], low[v]);
        }
        else low[u] = min(low[u], num[v]);
    }
    if (low[u] == num[u]) {
        scc++;
        int v;
        do {
            v = st.top();
            st.pop();
            num[v] = INF;
        }
        while (v != u);
    }
}
```

Topology Sort 1

```
//u -> v
//++deg[v]
for (int u = 1; u <= n; ++u)
    if (!deg[u]) q.push(u);

while (!q.empty()) {
    int u = q.front();
    q.pop();
    topo.push_back(u);
    for (auto v : g[u]) {
        deg[v]--;
        if (!deg[v]) q.push(v);
    }
}
```

Topology Sort 2

```
void dfs(int u) {
    visit[u] = 1;
    for (auto v : g[u]) {
        assert(visit[v] != 1);
        //graph contains a cycle
        if (!visit[v]) dfs(v);
    }
    topo.push(u);
    visit[u] = 2;
}
```

Max Flow

```
struct edge{
  int to, rev, flow, cap;
};

void add_edge(int u, int v, int cap){
  edge e1 = {v, sz(g[v]), 0, cap};
    edge e2 = {u, sz(g[u]), 0 , 0};
    g[u].pb(e1); g[v].pb(e2);
}

bool bfs(){
  memset(dist, 0x3f, sizeof dist);
  queue<int> q;
  q.push(source); dist[source] = 0;
  while (!q.empty()){
    int u = q.front(); q.pop();
    for (edge e: g[u]){
      int v = e.to, flow = e.flow, cap = e.cap;
      if (flow < cap && minimize(dist[v], dist[u] +
          1))
        q.push(v);
    }
  }
  return dist[sink] < INF;
}

int dfs(int u, int mn){
  if (u == sink) return mn;
  for (int &i = lazy[u]; i < sz(g[u]); ++i){
    auto &[v, rev, flow, cap] = g[u][i];
    if (dist[v] == dist[u] + 1 && flow < cap){
      int cur = dfs(v, min(mn, cap - flow));
      if (cur > 0){
        flow += cur;
        g[v][rev].flow -= cur;
        return cur;
      }
    }
  }
  return 0;
}
```

```
int main(){
    //...
    int res = 0;
    while (bfs()){
        memset(lazy, 0, sizeof lazy);
        while (int del = dfs(source, INF))
            res += del;
    }

    cout << res;
    return 0;
}
```

Bipartite Matching

```
bool dfs(int u){
  if (seen[u]) return 0;
  seen[u] = 1;

  for (int v: g[u])
    if (!mt[v] || dfs(mt[v]))
      return mt[v] = u, 1;

  return 0;
}

//memset(mt, 0, sizeof mt);
//For(i, 1, n){
    //memset(seen, 0, sizeof seen);
    //dfs(i);
//}
```

HLD

```
void dfs(int u){
  sz[u] = 1;
  for (int v: g[u]) if (v != par[u]){
    par[v] = u;
    dfs(v);
    sz[u] += sz[v];
  }
}

void hld(int u){
  if (!Head[nChain]) Head[nChain] = u;
  idChain[u] = nChain;

  pos[u] = ++timer;
  node[timer] = u;

  int bigC = 0;
  for (int v: g[u]) if (v != par[u])
    if (!bigC || sz[v] > sz[bigC])
      bigC = v;

  if (bigC) hld(bigC);
  for (int v: g[u]) if (v != par[u] && v != bigC){
    ++nChain;
    hld(v);;
  }
}

//LCA
int LCA(int u, int v){
  while (idChain[u] != idChain[v]){
    if (idChain[u] > idChain[v])
      u = par[Head[idChain[u]]];
    else
      v = par[Head[idChain[v]]];
  }

  if (h[u] < h[v]) return u;
  return v;
}

int get(int u, int v){
  int res = 0;
  while (idChain[u] != idChain[v]){
    if (idChain[u] > idChain[v]){
        maximize(res, ST.get(pos[Head[idChain[u]]],
            pos[u]));
        u = par[Head[idChain[u]]];
    }
    else{
        maximize(res, ST.get(pos[Head[idChain[v]]],
            pos[v]));
        v = par[Head[idChain[v]]];
    }
  }

  if (pos[u] < pos[v])
    maximize(res, ST.get(pos[u], pos[v]));
  else
    maximize(res, ST.get(pos[v], pos[u]));

  return res;
}
```

DSU on tree

```
void dfs(int u, int prev = -1){
  in[u] = ++timer; node[timer] = u;
  for (int v: g[u]) if (v != prev)
    dfs(v, u);
  out[u] = timer;
}

#define sz(u) out[u] - in[u]

void calc(int u, int prev = -1){
  int bigC = 0;
  for (int v: g[u]) if (v != prev)
    if (sz(v) > sz(bigC))
      bigC = v;

  for (int v: g[u]) if (v != prev && v != bigC){
      calc(v, u);
      //reset(v)...
    }

  if (bigC) calc(bigC, u);

  for (int v: g[u]) if (v != prev && v != bigC){
    For(t, in[v], out[v]){
      int x = node[t];
      //...
    }
  }
}
```

Centroid Decomposition

```
int size(int u, int prev){
  sz[u] = 1;
  for (int v: g[u]) if (!del[v] && v != prev)
```

```
    sz[u] += size(v, u);
  return sz[u];
}

int centroid(int u, int prev){
  for (int v: g[u]) if (!del[v] && v != prev)
    if (sz[v] > n/2)
      return centroid(v, u);
  return u;
}

void dfs(int u, int prev){
    in[u] = ++timer; node[timer] = u;
    for (int v: g[u]) if (!del[v] && v != prev){
        dfs(v, u);
        //...
    }
    out[u] = timer;
}

void calc(int u){
  n = size(u, 0);
  u = centroid(u, 0);

  timer = 0;
  dfs(u, 0);

    for (int v: g[u]) if (!del[v]){
        //subtree v...
    }
    //reset
  del[u] = 1;
  for (auto [v, c]: g[u]) if (!del[v])
    calc(v);
}
```

Centroid Tree (CT)

Centroid Tree properties:

- Centroid tree height $\leq \log(n)$
- $LCA(u, v)$ in CT lies on the path from u to v in the original tree

```
int size(int u, int prev){
  sz[u] = 1;
  for (int v: g[u]) if (v != prev && !del[v]){
    sz[u] += size(v, u);
  }
    return sz[u];
}

int centroid(int u, int prev, int m){
  for (int v: g[u]) if (v != prev && !del[v])
    if (sz[v] > m/2)
      return centroid(v, u, m);
  return u;
}

int cd(int u){
  int m = size(u);
  u = centroid(u, 0, m);
  del[u] = 1;
  for (int v: g[u]) if (!del[v]){
    v = cd(v);
    par[v] = u;
  }
  return u;
}

//example problems:
void solve(){
    dfs(1, 0); init(); //to calculate the dist(u, v)
        from the original tree
  cd(1);

  memset(d, 0x3f, sizeof d);

  c[1] = 1; //color
  int pp = 1;
  while (pp){
    minimize(d[pp], dist(pp, 1));
    pp = par[pp];
  }

  while (q--){
    int t; cin >> t;
    if (t == 1){
      int u; cin >> u;
      c[u] = 1;

      int p = u;
      while (p){
        minimize(d[p], dist(p, u));
        p = par[p];
      }
    }
    else{
      int u; cin >> u;
      if (c[u]) {
                cout << 0 << endl; continue;
            }

      int p = u, res = INF;
      while(p){
        minimize(res, dist(u, p) + d[p]);
        p = par[p];
      }

      cout << res << endl;
    }
  }
}
```

Virtual Tree

```
void dfs(int u){
  in[u] = ++timer;
  for (int v: g[u]) if (v != up[u][0]){
    up[v][0] = u;
    For(j, 1, 17) up[v][j] = up[up[v][j - 1]][j - 1];
    dfs(v);
  }
  out[u] = timer;
}

bool is_anc(int u, int v){
  if (!u) return 1;
  return in[u] <= in[v] && in[v] <= out[u];
}

//short LCA
int lca(int u, int v){
  if (is_anc(u, v)) return u;
  ForD(j, 17, 0){
      if (!is_anc(up[u][j], v)){
```

```
            u = up[u][j];
        }
    }
  return up[u][0];
}

bool cmp(int u, int v){
  return in[u] < in[v];
}

void query(){
  cin >> k;
  For(i, 1, k) cin >> a[i], sz[a[i]] = 1;

  sort(a + 1, a + k + 1, cmp);
  For(i, 1, k - 1) a[i + k] = lca(a[i], a[i + 1]);

  sort(a + 1, a + k + k, cmp);
  k = unique(a + 1, a + k + k) - a - 1;

  stack<int> st; st.push(a[1]);
  For(i, 2, k){
    while (!is_anc(st.top(), a[i])) st.pop();
    g[st.top()].pb(a[i]);
    st.push(a[i]);
  }

  res = 0; calc(a[1]);
  cout << res << endl;

  For(i, 1, k) sz[a[i]] = 0, g[a[i]].clear();
}

void solve(){
  //...
  dfs(1);
  For(i, 1, n) g[i].clear();
  while (q--) query();
}
```

Math

Euler's totient function

```
int phi(int n) {
    int res = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0) n /= i;
            res -= res / i;
        }
    }
    if (n > 1) res -= res / n;
    return res;
}
```

Euler's totient function from 1 to N

```
void preCompute(int n) {
    iota(phi, phi + N, 0); //phi[i] = i
    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
```

Modular Inverse

```
//if MOD is a prime number then phi(MOD)= MOD - 1
int inv(int x, int MOD){
    return Pow(x, phi(MOD) - 1);
}
```

Extended Euclidean Algorithm

```
//computing gcd(a, b) and finding (x, y) that
//ax + by = gcd(a, b)

//recursive version
int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1; y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}

//iterative version
int gcd(int a, int b, int& x, int& y) {
    x = 1, y = 0;
    int x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        int q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a1, b1) = make_tuple(b1, a1 - q * b1);
    }
    return a1;
}
```

Diophantine

```
bool find_any_solution(int a, int b, int c, int &x0,
     int &y0, int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) return false;

    x0 *= c / g; y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}

//all the solutions have the form:
//x = x0 + k * b/g
//y = y0 - k * b/g


//IN A GIVEN INTERVAL:
void shift(int & x, int & y, int a, int b, int cnt) {
    x += cnt * b;
    y -= cnt * a;
}

int find_all_solutions(int a, int b, int c, int minx,
      int maxx, int miny, int maxy) {
    int x, y, g;
    if (!find_any_solution(a, b, c, x, y, g)) return
         0;
    a /= g; b /= g;
```

```
    int sign_a = a > 0 ? +1 : -1;
    int sign_b = b > 0 ? +1 : -1;

    shift(x, y, a, b, (minx - x) / b);
    if (x < minx) shift(x, y, a, b, sign_b);
    if (x > maxx) return 0;
    int lx1 = x;

    shift(x, y, a, b, (maxx - x) / b);
    if (x > maxx) shift(x, y, a, b, -sign_b);
    int rx1 = x;

    shift(x, y, a, b, -(miny - y) / a);
    if (y < miny) shift(x, y, a, b, -sign_a);
    if (y > maxy) return 0;
    int lx2 = x;

    shift(x, y, a, b, -(maxy - y) / a);
    if (y > maxy) shift(x, y, a, b, sign_a);
    int rx2 = x;

    if (lx2 > rx2) swap(lx2, rx2);
    int lx = max(lx1, lx2);
    int rx = min(rx1, rx2);

    if (lx > rx) return 0;
    return (rx - lx) / abs(b) + 1;
}
```

Chinese Remainder Theorem

```
// Combine two congruences:
// x = a1 (mod m1), x = a2 (mod m2)
// Returns (x, lcm) or (-1,-1) if no solution
pair<ll, ll> crt2(ll a1, ll m1, ll a2, ll m2) {
    int x, y;
    ll g = gcd(m1, m2, x, y);

    if ((a2 - a1) % g != 0) {
        return {-1, -1}; // no solution
    }

    ll lcm = m1 / g * m2;

    ll k = (a2 - a1) / g;
    ll mult = (1LL * x * k) % (m2 / g);

    ll ans = (a1 + m1 * mult) % lcm;
    if (ans < 0) ans += lcm;

    return {ans, lcm};
}

//solve a system of congruences:
//x = a1 (mod m1)
//x = a2 (mod m2)
//...
//x = ak (mod mk)
pair<ll, ll> crt(vector<ll> a, vector<ll> m) {
    pair<ll,ll> res = {a[0], m[0]};
    for (int i = 1; i < sz(a); i++) {
        res = crt2(res.first, res.second, a[i], m[i])
            ;
        if (res.first == -1) return {-1,-1};
    }
    return res;
}
//x = sol.first (mod sol.second)
```

Rabin-Miller primality test

```
bool test(ll a, ll n, ll k, ll m){
    ll mod = Pow(a, m, n);
    if (mod == 1 || mod == n - 1) return 1;
    for (int l = 1; l < k; ++l){
        mod = (mod * mod) % n;
        if (mod == n - 1) return 1;
    }
    return 0;
}

//check if n is a prime number
bool RabinMiller(ll n){
    if (n == 2 || n == 3 || n == 5 || n == 7) return
         1;
    if (n < 11) return 0;

    ll k = 0, m = n - 1;
    while (!(m & 1)){
        m >>= 1;
        k++;
    }

    const static int repeatTime = 3;
    for (int i = 0; i < repeatTime; ++i){
        ll a = rand() % (n - 3) + 2;
        if (!test(a, n, k, m)) return 0;
    }
    return 1;
}
```

Combination

```
//recursive version:
void preCompute(){
    for (int i = 0; i <= n; i++){
        C[i][0] = 1;
        for (int k = 1; k <= i; k++){
            C[i][k] = C[i - 1][k - 1] + C[i - 1][k];
        }
    }
}

//"you know what it is" version:
int C(int n, int k){
    if (n < k || k < 0) return 0;
    return mul(fact[n], mul(ifact[n - k], ifact[k]))
}
```

Geometry