Momentum Library

Minified

Competitive Programming Library

Of

Omar Bazaraa

https://github.com/OmarBazaraa/Competitive-Programming

October 16th, 2018

Table of Contents

Data Structures

Sparse Table

```
int n, a[N], ST[LOG_N][N], LOG[N];

void buildST() {
    LOG[0] = -1;

    for (int i = 0; i < n; ++i) {
        ST[0][i] = i;
        LOG[i + 1] = LOG[i] + !(i & (i + 1));
    }

    for (int j = 1; (1 << j) <= n; ++j) {
        for (int i = 0; (i + (1 << j)) <= n; ++i) {
            int x = ST[j - 1][i];
            int y = ST[j - 1][i + (1 << (j - 1))];

            ST[j][i] = (a[x] <= a[y] ? x : y);
        }
    }
}

int query(int l, int r) {
    int g = LOG[r - l + 1];
    int x = ST[g][l];
    int y = ST[g][r - (1 << g) + 1];
    return (a[x] <= a[y] ? x : y);
}
```

Monotonic Queue

```
template<class T>
class monotonic_queue {
    queue<T> qu;
    deque<T> mx;

public:
    void push(T v) {
        qu.push(v);
        while (mx.size() && mx.back() < v) mx.pop_back();
        mx.push_back(v);
    }

    void pop() {
        if (mx.front() == qu.front()) mx.pop_front();
        qu.pop();
    }

    T front() {
        return qu.front();
    }

    T max() {
        return mx.front();
    }

    T size() {
        return qu.size();
    }
};
```

Disjoint-Sets Union (DSU)

```
int setsCount;
int par[N], siz[N];

void init(int n) {
	setsCount = n;

	for (int i = 0; i < n; ++i) {
		par[i] = i;
		siz[i] = 1;
	}
}

int findSet(int u) {
	return u == par[u] ? u : par[u] = findSet(par[u]);
}

bool sameSet(int u, int v) {
	return findSet(u) == findSet(v);
}

bool unionSets(int u, int v) {
	u = findSet(u);
	v = findSet(v);

	if (u == v) {
		return false;
	}

	setsCount--;
	siz[v] += siz[u];
	par[u] = v;
	return true;
}

int getSetSize(int u) {
	return siz[findSet(u)];
}
```

Fenwick Tree

```
class fenwick_tree {
    int BIT[N];

public:
    fenwick_tree() {
        memset(BIT, 0, sizeof(BIT));
    }

    void update(int idx, int val) {
        while (idx < N) {
            BIT[idx] += val;
            idx += idx & -idx;
        }
    }

    int operator[](int idx) {
        int res = 0;
        while (idx > 0) {
            res += BIT[idx];
            idx -= idx & -idx;
        }
        return res;
    }
};

class range_fenwick_tree {
    fenwick_tree M, C;

public:
    void update(int l, int r, int val) {
        M.update(l, val);
        M.update(r + 1, -val);
        C.update(l, -val * (l - 1));
        C.update(r + 1, val * r);
    }

    int operator[](int idx) {
        return idx * M[idx] + C[idx];
    }
};
```

Segment Tree as Multiset

```
const int MAX_RANGE = 1e9;

struct node {
    int size;
    node *childL, *childR;

    node() {
        size = 0;
        childL = childR = this;
    }

    node(int s, node* l, node* r) {
        size = s;
        childL = l;
        childR = r;
    }

    void update() {
        size = childL->size + childR->size;
    }
};

class segment_multiset {
    node *nil, *root;

public:
    segment_multiset() {
        root = nil = new node();
    }

    ~segment_multiset() {
        clear();
        delete nil;
    }

    void clear() {
        destroy(root);
        root = nil;
    }

    int size() {
        return root->size;
    }

    int count(int val) {
        node* cur = root;
        int l = - MAX_RANGE, r = MAX_RANGE;

        while (l < r) {
            int mid = l + (r - l) / 2;
```

```
            if (val <= mid) {
                cur = cur->childL;
                r = mid;
            } else {
                cur = cur->childR;
                l = mid + 1;
            }
        }

        return cur->size;
    }

    void insert(int val, int cnt = 1) {
        assert(cnt > 0);
        insert(root, val, cnt, -MAX_RANGE, MAX_RANGE);
    }

    int erase(int val, int cnt = 1) {
        assert(cnt > 0);
        return erase(root, val, cnt, -MAX_RANGE, MAX_RANGE);
    }

    // Returns integer from the multiset by its index (0-indexed).
    int operator[](int idx) {
        if (idx < 0 || idx >= root->size) {
            throw out_of_range("ERROR :: trying to access an out of range
element");
        }

        node* cur = root;
        int l = -MAX_RANGE, r = MAX_RANGE;

        while (l < r) {
            int mid = l + (r - l) / 2;

            if (idx < cur->childL->size) {
                cur = cur->childL;
                r = mid;
            } else {
                idx -= cur->childL->size;
                cur = cur->childR;
                l = mid + 1;
            }
        }

        return r;
    }

    int lower_bound(int val) {
        int ret = 0;

        node* cur = root;
        int l = -MAX_RANGE, r = MAX_RANGE;
```

```
        while (l < val) {
            int mid = l + (r - l) / 2;

            if (val <= mid) {
                cur = cur->childL;
                r = mid;
            } else {
                ret += cur->childL->size;
                cur = cur->childR;
                l = mid + 1;
            }
        }

        return ret;
    }

    int upper_bound(int val) {
        return lower_bound(val + 1);
    }

private:

    void insert(node*& root, int val, int cnt, int l, int r) {
        if (val < l || val > r) {
            return;
        }

        if (root == nil) {
            root = new node(0, nil, nil);
        }

        root->size += cnt;

        if (l == r) {
            return;
        }

        int mid = l + (r - l) / 2;

        insert(root->childL, val, cnt, l, mid);
        insert(root->childR, val, cnt, mid + 1, r);
    }

    int erase(node*& root, int val, int cnt, int l, int r) {
        if (val < l || val > r) {
            return 0;
        }

        if (root == nil) {
            return 0;
        }
```

```
        if (l == r) {
            return remove(root, cnt);
        }

        int mid = l + (r - l) / 2;

        int ret = 0;

        ret += erase(root->childL, val, cnt, l, mid);
        ret += erase(root->childR, val, cnt, mid + 1, r);

        return remove(root, ret);
    }

    int remove(node*& root, int cnt) {
        int ret = min(cnt, root->size);

        root->size -= cnt;

        if (root->size <= 0) {
            destroy(root);
            root = nil;
        }

        return ret;
    }

    void destroy(node* root) {
        if (root == nil) {
            return;
        }

        destroy(root->childL);
        destroy(root->childR);
        delete root;
    }
};
```

Strings

KMP

```
int F[N];

int failure(const char* pat, char c, int len) {
    while (len > 0 && c != pat[l]) {
        len = F[l - 1];
    }
    return len + (c == pat[l]);
}

void KMP(const char* str) {
    F[0] = 0;
    for (int i = 1; str[i]; ++i) {
        F[i] = failure(str, str[i], F[i - 1]);
    }
}
```

Z-Algorithm

```
int z[N];

void z_function(const char* str) {
    for (int i = 1, l = 0, r = 0; str[i]; ++i) {
        if (i <= r)
            z[i] = min(r - i + 1, z[i - l]);

        while (str[i + z[i]] && str[z[i]] == str[i + z[i]])
            z[i]++;

        if (i + z[i] - 1 > r)
            l = i,
            r = i + z[i] - 1;
    }
}
```

Trie

```
const int N = 100100, ALPA = 255;   // N: total length of all strings

int nodesCount;
int distinctWordsCount;
int trie[N][A];
int wordsCount[N];                  // Number of words sharing node "i"
int wordsEndCount[N];               // Number of words ending at node "i"

void init() {
    nodesCount = 0;
    memset(trie, -1, sizeof(trie));
}

int addEdge(int id, char c) {
    int& nxt = trie[id][c];
    if (nxt == -1) {
        nxt = ++nodesCount;
    }
    return nxt;
}

void insert(const char* str) {
    int cur = 0;

    for (int i = 0; str[i]; ++i) {
        wordsCount[cur]++;
        cur = addEdge(cur, str[i]);
    }

    wordsCount[cur]++;
    distinctWordsCount += (++wordsEndCount[cur] == 1);
}

void erase(const char* str) {
    int cur = 0;

    for (int i = 0; str[i]; ++i) {
        int nxt = trie[cur][str[i]];

        if (wordsCount[cur]-- == 1) {
            trie[cur][str[i]] = -1;
        }

        cur = nxt;
    }

    wordsCount[cur]--;
    distinctWordsCount -= (--wordsEndCount[cur] == 0);
}
```

```
bool search(const char* str) {
    int cur = 0;

    for (int i = 0; str[i]; ++i) {
        int nxt = trie[cur][str[i]];

        if (nxt == -1) {
            return 0;
        }

        cur = nxt;
    }

    return wordsEndCount[cur];
}
```

Suffix Array (N.log(N))

```
const int N = 1e5 + 5;

// n:           the number of suffixes (length of the string + 1)
// SA:          the suffix array, holding all the suffixes in lexicographical
order.
// suffixRank:  the order of the i-th suffix after sorting.
// LCP:         the length of the longest common prefix between SA[i] and
SA[i - 1].

int n, SA[N], suffixRank[N], LCP[N];

// Temporary arrays needed while computing the suffix array
int sortedSA[N], sortedRanks[N], rankStart[N];

struct comparator {
    int h;

    comparator(int h) : h(h) {}

    bool operator()(int i, int j) const {
        if (suffixRank[i] != suffixRank[j])
            return suffixRank[i] < suffixRank[j];
        return suffixRank[i + h] < suffixRank[j + h];
    }
};

void computeSuffixRanks(int h) {
    comparator comp(h);

    for (int i = 1; i < n; ++i) {
        int& r = sortedRanks[i] = sortedRanks[i - 1];

        if (comp(sortedSA[i - 1], sortedSA[i])) {
            rankStart[++r] = i;
        }
    }

    for (int i = 0; i < n; ++i) {
        SA[i] = sortedSA[i];
        suffixRank[SA[i]] = sortedRanks[i];
    }
}
```

```
void buildSuffixArray(const string& str) {
    n = str.size() + 1;

    for (int i = 0; i < n; ++i) {
        sortedSA[i] = i;
        suffixRank[i] = str[i];
    }

    sort(sortedSA, sortedSA + n, comparator(0));
    computeSuffixRanks(0);

    for (int h = 1; sortedRanks[n - 1] != n - 1; h <<= 1) {
        for (int i = 0; i < n; ++i) {
            int k = SA[i] - h;

            if (k >= 0) {
                sortedSA[rankStart[suffixRank[k]]++] = k;
            }
        }

        computeSuffixRanks(h);
    }
}

void buildLCP(const string& str) {
    int cnt = 0;
    for (int i = 0, k = 0; i < str.size(); ++i) {
        int j = SA[suffixRank[i] - 1];
        while (str[i + cnt] == str[j + cnt]) ++cnt;
        LCP[suffixRank[i]] = cnt;
        if (cnt > 0) --cnt;
    }
}
```

Graphs

Topological Sort

```
bool vis[N];
vector<int> edges[N];       // Graph adjacency list
vector<int> sortedNodes;    // List of topologically sorted nodes

void topoSortDFS(int u) {
    vis[u] = true;

    for (int v : edges[u]) {
        if (!vis[v]) {
            topoSortDFS(v);
        }
    }

    sortedNodes.push_back(u);
}

void topoSortBFS() {
    queue<int> q;
    vector<int> inDeg(n + 1, 0);

    for (int i = 1; i <= n; ++i) {
        for (int v : edges[i]) {
            ++inDeg[v];
        }
    }

    for (int i = 1; i <= n; ++i) {
        if (inDeg[i] == 0) {
            q.push(i);
        }
    }

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        sortedNodes.push_back(u);

        for (int v : edges[u]) {
            if (--inDeg[v] == 0) {
                q.push(v);
            }
        }
    }
}
```

Bellman Ford

```
vector<pair<int, int>> edges[N];

bool bellmanFord(int src) {
    memset(par, -1, sizeof(par));
    memset(dis, 0x3F, sizeof(dis));
    dis[src] = 0;

    bool updated = 1;

    for (int k = 0; k < n && updated; ++k) {
        updated = 0;

        for (int u = 1; u <= n; ++u) {
            for (auto& e : edges[u]) {
                int v = e.first;
                int w = e.second;

                if (dis[v] > dis[u] + w) {
                    dis[v] = dis[u] + w;
                    par[v] = u;
                    updated = 1;
                }
            }
        }
    }

    return updated;   // Whether a negative cycle exist or not
}
```

Floyd Warshal

```
// Note that:
// ------------
// adj[u][u] = 0
// adj[u][v] = cost(u, v)   when direct edge exists from u to v with cost (u, v)
// adj[u][v] = INF          otherwise
// ---------------------------------------------------
// par[u][v] = u

int n, adj[N][N], par[N][N];

void init() {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            adj[i][j] = (i == j ? 0 : 1e9),
            par[i][j] = i;
}

void floyd() {
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (adj[i][j] > adj[i][k] + adj[k][j])
                    adj[i][j] = adj[i][k] + adj[k][j],
                    par[i][j] = par[k][j];
}

bool checkNegativeCycle() {
    bool ret = false;
    for (int i = 0; i < n; ++i) {
        ret = ret || (adj[i][i] < 0);
    }
    return ret;
}
```

BFS on Complement Graph

```
void bfs(int u) {
    queue<int> q;
    q.push(u);

    memset(dis, -1, sizeof(dis));
    dis[u] = 0;

    int id = 0;
    list<int> l1, l2;
    memset(vis,  0, sizeof(vis));

    for (int i = 1; i <= n; ++i) {
        if (i != u) {
            l1.push_back(i);
        }
    }

    while (!q.empty()) {
        u = q.front();
        q.pop();
        id++;

        for (int v : edges[u]) {
            if (dis[v] == -1) {
                vis[v] = id;
            }
        }

        for (int v : l1) {
            if (vis[v] == id) {
                l2.push_back(v);
                continue;
            }

            dis[v] = dis[s] + 1;
            q.push(v);
        }

        l1.clear();
        l1.swap(l2);
    }
}
```

Dijkstra

```
struct edge {
    int from, to, weight;

    edge() {}
    edge(int f, int t, int w) : from(f), to(t), weight(w) {}

    bool operator<(const edge& rhs) const {
        return weight > rhs.weight;
    }
};

void dijkstra(int src) {
    priority_queue<edge> q;
    q.push(edge(-1, src, 0));

    memset(par, -1, sizeof(par));
    memset(dis, 0x3F, sizeof(dis));

    while (!q.empty()) {
        int u = q.top().from;
        int v = q.top().to;
        int w = q.top().weight;
        q.pop();

        if (dis[v] <= w) {
            continue;
        }

        dis[v] = w;
        par[v] = u;

        for (edge& e : edges[v]) {
            if (w + e.weight < dis[e.to]) {
                q.push(edge(v, e.to, w + e.weight));
            }
        }
    }
}
```

Kosaraju's SCC Algorithm

```
int n, m;                   // Number of nodes and edges
bool vis[N];                // Whether node u has been visited before or not
vector<int> edges[N];       // Graph adjacency list
vector<int> edgesT[N];      // Transposed graph adjacency list (i.e. with
reversed edges)
vector<int> sortedNodes;    // List of topologically sorted nodes
vector<vector<int>> scc;    // Strongly connected components

void topoSortDFS(int u, vector<int>* edges, vector<int>& nodes) {
        vis[u] = 1;

        for (int v : edges[u]) {
                if (!vis[v]) {
                        topoSortDFS(v, edges, nodes);
                }
        }

        nodes.push_back(u);
}

void kosaraju() {
        memset(vis, 0, sizeof(vis));
        for (int i = 1; i <= n; ++i) {
                if (!vis[i]) {
                        topoSortDFS(i, edges, sortedNodes);
                }
        }

        memset(vis, 0, sizeof(vis));
        for (int i = sortedNodes.size() - 1; i >= 0; --i) {
                int u = sortedNodes[i];

                if (!vis[u]) {
                        scc.push_back(vector<int>());
                        topoSortDFS(u, edgesT, scc.back());
                }
        }
}
```

LCA (Eular walk + RMQ)

```
int n, m, u, v, dep[N];
int F[N], ST[LOG_N][N << 1], LOG[N << 1];
vector<int> E, edges[N];

void dfs(int u = 1, int p = 0, int d = 0) {
    dep[u] = d;
    F[u] = E.size();
    E.push_back(u);

    for (int v : edges[u]) {
        if (v != p) {
            dfs(v, u, d + 1);
            E.push_back(u);
        }
    }
}

void buildRMQ() {
    int i, j, x, y;
    for (i = 0, LOG[0] = -1; i < E.size(); ++i) {
        ST[0][i] = i;
        LOG[i + 1] = LOG[i] + !(i & (i + 1));
    }
    for (j = 1; (1 << j) <= E.size(); ++j) {
        for (i = 0; (i + (1 << j)) <= E.size(); ++i) {
            x = ST[j - 1][i];
            y = ST[j - 1][i + (1 << (j - 1))];
            ST[j][i] = (dep[E[x]] < dep[E[y]]) ? x : y;
        }
    }
}

int query(int l, int r) {
    if (l > r) swap(l, r);
    int g = LOG[r - l + 1];
    int x = ST[g][l];
    int y = ST[g][r - (1 << g) + 1];
    return (dep[E[x]] < dep[E[y]]) ? x : y;
}

int getLCA(int u, int v) {
    return E[query(F[u], F[v])];
}

int getDistance(int u, int v) {
    return dep[u] + dep[v] - 2 * dep[getLCA(u, v)];
}
```

LCA (Parent Sparse Table)

```
int n, m, u, v, dep[N];
int LOG[N], par[LOG_N][N];    // par[j][i] : the (2^j)-th ancestor of node number i.
vector<int> edges[N];

void dfs(int u = 1, int p = 0, int d = 0) {
    dep[u] = d;
    par[0][u] = p;

    for (int i = 1; (1 << i) <= d; ++i) {
        par[i][u] = par[i - 1][par[i - 1][u]];
    }

    for (int v : edges[u]) {
        if (v != p) {
            dfs(v, u, d + 1);
        }
    }
}

int getAncestor(int u, int k) {
    while (k > 0) {
        int x = k & -k;
        k -= x;
        u = par[LOG[x]][u];
    }

    return u;
}

int getLCA(int u, int v) {
    if (dep[u] > dep[v]) {
        swap(u, v);
    }

    v = getAncestor(v, dep[v] - dep[u]);

    if (u == v) {
        return u;
    }

    for (int i = LOG[dep[u]]; i >= 0; --i) {
        if (par[i][u] != par[i][v]) {
            u = par[i][u];
            v = par[i][v];
        }
    }

    return par[0][u];
}
```

```
int getDistance(int u, int v) {
    return dep[u] + dep[v] - 2 * dep[getLCA(u, v)];
}

void computeLog() {
    LOG[0] = -1;
    for (int i = 1; i <= n; ++i) {
        LOG[i] = LOG[i - 1] + !(i & (i - 1));
    }
}
```

Max Flow (Edmonds-Karp's Algorithm)

```
const int N = 105;  // Max number of nodes
const int M = 105;  // Max number of edges

int n, m;        // Number of nodes and number of edges

int edgeId;      // The next edge id to be inserted
int head[N];     // head[u]: id of the last edge added from node u
int nxt[M];      // nxt[e] : next edge id pointed from the same node as e
int to[M];       // to[e]  : id of the node pointed by edge e
int capacity[M]; // capacity[e] : maximum capacity of edge e
int flow[M];     // flow[u]: current flow of edge e

int src, snk;    // id of source and sink nodes
int dist[N];     // dist[u]: shortest distance between the source and node u
int from[N];     // from[u]: id of the edge that leads to node u in the
                 //          path from source to sink nodes

void init() {
    edgeId = 0;
    memset(head, -1, sizeof(head));
}

void addEdge(int f, int t, int c) {
    int e = edgeId++;

    to[e] = t;
    capacity[e] = c;
    flow[e] = 0;

    nxt[e] = head[f];
    head[f] = e;
}

void addAugEdge(int f, int t, int c) {
    addEdge(f, t, c);
    addEdge(t, f, 0);
}

bool findPath() {
    queue<int> q;
    q.push(src);

    memset(dist, -1, sizeof(dist));
    dist[src] = 0;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int e = head[u]; ~e; e = nxt[e]) {
            int v = to[e];
```

```
            int c = capacity[e];
            int f = flow[e];

            if (c <= f) {
                continue;
            }

            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                from[v] = e;
                q.push(v);
            }

            if (v == snk) {
                return 1;
            }
        }
    }

    return 0;
}

int augmentPath() {
    int f = INT_MAX;

    for (int u = snk, e, r; u != src; u = to[r]) {
        e = from[u];    // x ---e--> u
        r = e ^ 1;      // x <--r--- u

        f = min(f, capacity[e] - flow[e]);
    }

    for (int u = snk, e, r; u != src; u = to[r]) {
        e = from[u];    // x ---e--> u
        r = e ^ 1;      // x <--r--- u

        flow[e] += f;
        flow[r] -= f;   // Reversed edge for flow cancelation
    }

    return f;
}

int maxFlow() {
    int f = 0;

    while (findPath()) {
        f += augmentPath();
    }

    return f;
}
```

Tree Diameter

```
int bfs(int u) {
    queue<int> q;
    q.push(u);

    memset(dis, -1, sizeof(dis));
    dis[u] = 0;

    while (!q.empty()) {
        u = q.front();
        q.pop();

        for (auto v : edges[u]) {
            if (dis[v] == -1) {
                dis[v] = dis[u] + 1;
                q.push(v);
            }
        }
    }

    return u;
}

int calcTreeDiameter(int root) {
    int u = bfs(root);
    int v = bfs(u);
    return dis[v];
}
```

Bipartite Graph Check

```
int color[N];
vector<int> edges[N];

bool dfs(int u = 0) {
    for (int v : edges[u]) {
        if (color[v] == -1) {
            color[v] = color[u] ^ 1;

            if (!dfs(v)) {
                return false;
            }
        }
        else if (color[v] == color[u]) {
            return false;
        }
    }

    return true;
}

bool isBipartiteGraph() {
    memset(color, -1, sizeof(color));
    color[1] = 0;
    return dfs();
}
```

Bridge Tree

```
// n:           total number of vertices in the graph.
// T:           counter represents time.
// root:        node id in the built bridge tree after calling
//              buildBridgeTree().
// par[u]:      the parent array of the DSU data structure.
// tin[u]:      visiting (discovery) time of node u.
// low[u]:      earliest visiting time of a vertex that node u is reachable
//              from.
// edges[u]:    list of out edges of node u in the graph.
// tree[u]:     list of out edges of node u in the built bridge tree after
//              calling
//              buildBridgeTree().
// bridges:     list contains all bridge edges of the graph after calling
//              findBridges().

int n;
int T, root, par[N], tin[N], low[N];
vector<int> edges[N], tree[N];
vector<pair<int, int>> bridges;

int findSet(int u) {
    return (par[u] == u ? u : par[u] = findSet(par[u]));
}

void unionSets(int u, int v) {
    par[findSet(u)] = findSet(v);
}

void findBridges(int u = 1, int p = -1) {
    tin[u] = low[u] = ++T;

    for (auto v : edges[u]) {
        if (v == p) {
            continue;
        }

        if (tin[v] == 0) {
            findBridges(v, u);

            if (low[v] > tin[u]) {
                bridges.push_back({ u, v });
            } else {
                unionSets(u, v);
            }
        }

        low[u] = min(low[u], low[v]);
    }
}
```

```
void buildBridgeTree() {
    for (int i = 1; i <= n; ++i) {
        par[i] = i;
    }

    findBridges();

    for (auto& b : bridges) {
        int u = findSet(b.first);
        int v = findSet(b.second);

        tree[u].push_back(v);
        tree[v].push_back(u);

        root = u;
    }
}
```

Math

```
int gcd (int a, int b) {
    return b == 0 ? a : gcd (b, a % b);
}

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

pair<int, int> extendedEuclid(int a, int b) {
    if (b == 0) {
        return { 1, 0 };
    }

    pair<int, int> p = extendedEuclid(b, a % b);

    int s = p.first;
    int t = p.second;

    return { t, s - t * (a / b) };
}

int power(int base, int exp, int mod) {
    int ans = 1;
    base %= mod;

    while (exp > 0) {
        if (exp & 1) ans = (ans * base) % mod;
        exp >>= 1;
        base = (base * base) % mod;
    }

    return ans;
}

int modInverse(int a, int m) {
    return power(a, m - 2, m);
}

int nCr(int n, int r) {
    if (n < r)
        return 0;

    if (r == 0)
        return 1;

    return n * nCr(n - 1, r - 1) / r;
}
```

```
int comb[N][N];
void buildPT(int n) {
    for (int i = comb[0][0] = 1; i <= n; ++i)
        for (int j = comb[i][0] = 1; j <= i; ++j)
            comb[i][j] = (comb[i - 1][j] + comb[i - 1][j - 1]) % MOD;
}

bool isPrime(int n) {
    if (n < 2)
        return 0;
    if (n % 2 == 0)
        return (n == 2);
    for (int i = 3; i * i <= n; i += 2)
        if (n % i == 0)
            return 0;
    return 1;
}

bool prime[N];
void generatePrimes(int n) {
    memset(prime, true, sizeof(prime));
    prime[0] = prime[1] = false;

    for (int i = 2; i * i <= n; ++i) {
        if (!prime[i]) continue;

        for (int j = i * i; j <= n; j += i) {
            prime[j] = false;
        }
    }
}

vector<int> primeDivs[N];
void generatePrimeDivisors(int n) {
    for (int i = 2; i <= n; ++i) {
        if (primeDivs[i].size()) continue;

        for (int j = i; j <= n; j += i) {
            primeDivs[j].push_back(i);
        }
    }
}
```

```
vector<int> getDivisors(int n) {
    vector<int> divs;

    for (int i = 1; i * i <= n; ++i) {
        if (n % i == 0) {
            divs.push_back(i);

            if (i * i != n) {
                divs.push_back(n / i);
            }
        }
    }

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

    return divs;
}

vector<int> divs[N];
void generateDivisors(int n) {
    for (int i = 1; i <= n; ++i)
        for (int j = i; j <= n; j += i)
            divs[j].push_back(i);
}
```

Others

Longest Increasing Sub-sequence

```
int n, a[N];

int getLIS() {
    if (n < 1) return 0;

    int len = 0;
    vector<int> LIS(n, INT_MAX);

    for (int i = 0; i < n; ++i) {
        // To get the length of the longest non decreasing subsequence
        // replace function "lower_bound" with "upper_bound"
        int idx = lower_bound(LIS.begin(), LIS.end(), a[i]) - LIS.begin();
        LIS[idx] = a[i];
        len = max(len, idx);
    }

    return len + 1;
}
```