Momentum Library

Minified

Competitive Programming Library

of

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

September 11th, 2019

Contents

[Data Structures 1](#_Toc19122871)

[Sparse Table 1](#_Toc19122872)

[Monotonic Queue 1](#_Toc19122873)

[Disjoint-Set Union (DSU) 2](#_Toc19122874)

[Fenwick Tree (Binary Indexed Tree) 3](#_Toc19122875)

[Segment Tree as Multiset 4](#_Toc19122876)

[Strings 8](#_Toc19122877)

[KMP 8](#_Toc19122878)

[Z-Algorithm 8](#_Toc19122879)

[Trie 9](#_Toc19122880)

[Suffix Array 10](#_Toc19122881)

[Graphs 12](#_Toc19122882)

[Shortest Path (Floyd Warshal’s Algorithm) 12](#_Toc19122883)

[Shortest Path (Bellman Ford’s Algorithm) 12](#_Toc19122884)

[Shortest Path (Dijkstra’s Algorithm) 13](#_Toc19122885)

[Minimum Spanning Tree (Kruskal’s Algorithm) 14](#_Toc19122886)

[SCC (Kosaraju’s Algorithm) 15](#_Toc19122887)

[Topological Sort (Khan’s Algorithm) 15](#_Toc19122888)

[Tree Diameter 16](#_Toc19122889)

[Bipartite Graph Check 17](#_Toc19122890)

[Bridge Tree 17](#_Toc19122891)

[LCA (Euler Walk + RMQ) 18](#_Toc19122892)

[LCA (Parent Sparse Table) 20](#_Toc19122893)

[Max Flow (Edmonds Karp’s Algorithm) 21](#_Toc19122894)

[Math 24](#_Toc19122895)

[GCD 24](#_Toc19122896)

[LCM 24](#_Toc19122897)

[Extended Euclid 24](#_Toc19122898)

[Fast Power 24](#_Toc19122899)

[Modular Inverse 25](#_Toc19122900)

[Combinations (nCr) 25](#_Toc19122901)

[Pascal’s Triangle 25](#_Toc19122902)

[Prime Check 25](#_Toc19122903)

[Prime Check (Miller Rabin’s Algorithm) 26](#_Toc19122904)

[Generate Primes 26](#_Toc19122905)

[Generate Prime Divisors 27](#_Toc19122906)

[Generate Divisors 27](#_Toc19122907)

[Others 28](#_Toc19122908)

[Longest Increasing Sub-sequence 28](#_Toc19122909)

# Data Structures

## Sparse Table

*// The array to compute its sparse table and its size.***int** n, a[N];

*// Sparse table related variables. Don't access them directly.***int** ST[LOG\_N][N], LOG[N];  
 *// Builds the sparse table for computing min/max/gcd/lcm/...etc  
// for any contiguous sub-segment of the array in O(n.log(n)).*

*//  
// This is an example of computing the index of the minimum value.***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);  
 }  
 }  
}

*// Queries the sparse table for the computed value of the interval [l, r] in O(1).***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

*/\*\*  
 \* Monotonic queue to keep track of the minimum and the maximum  
 \* elements so far in the queue in amortized time of O(1).  
 \*/***template**<**class** T>  
**class** monotonic\_queue {  
 queue<T> qu;  
 deque<T> mx, mn;  
  
**public**:

**void** push(T v) {  
 qu.push(v);  
 **while** (mx.size() && mx.back() < v) mx.pop\_back();  
 mx.push\_back(v);  
 **while** (mn.size() && mn.back() > v) mn.pop\_back();  
 mn.push\_back(v);  
 }  
   
 **void** pop() {  
 **if** (mx.front() == qu.front()) mx.pop\_front();  
 **if** (mn.front() == qu.front()) mn.pop\_front();  
 qu.pop();  
 }  
   
 T front() **const** {  
 **return** qu.front();  
 }  
   
 T max() **const** {  
 **return** mx.front();  
 }  
   
 T min() **const** {  
 **return** mn.front();  
 }  
   
 size\_t size() **const** {  
 **return** qu.size();  
 }  
};

## Disjoint-Set Union (DSU)

*/\*\*  
 \* Disjoint-set data structure to tracks a set of elements partitioned  
 \* into a number of disjoint subsets.  
 \*/***class** DSU {  
 **int** setsCount;  
 vector<**int**> siz;  
 **mutable** vector<**int**> par;  
  
**public**:  
DSU(**int** n) {  
 setsCount = n;  
 siz.resize(n, 1);  
 par.resize(n);  
 iota(par.begin(), par.end(), 0);  
 }  
**int** findSetId(**int** u) **const** {  
 **return** u == par[u] ? u : par[u] = findSetId(par[u]);  
 }  
**bool** areInSameSet(**int** u, **int** v) **const** {  
 **return** findSetId(u) == findSetId(v);  
 }  
**bool** unionSets(**int** u, **int** v) {  
 u = findSetId(u);  
 v = findSetId(v);  
  
 **if** (u == v) {  
 **return false**;  
 }  
  
 setsCount--;  
 siz[v] += siz[u];  
 par[u] = v;  
 **return true**;  
 }  
**int** getSetSize(**int** u) **const** {  
 **return** siz[findSetId(u)];  
 }  
**int** getSetsCount() **const** {  
 **return** setsCount;  
 }  
};

## Fenwick Tree (Binary Indexed Tree)

*/\*\*  
 \* Regular Fenwick tree class to compute and update prefix sum in O(log(N)).  
 \*  
 \* Note that the tree is is 1-indexed.  
 \*/***template**<**class** T>  
**class** fenwick\_tree {  
 T BIT[N];  
  
**public**:

fenwick\_tree() {  
 memset(BIT, 0, **sizeof**(BIT));  
 }  
**void** update(**int** idx, T val) {  
 **while** (idx < N) {  
 BIT[idx] += val;  
 idx += idx & -idx;  
 }  
 }  
T **operator**[](**int** idx) {  
 T res = 0;  
 **while** (idx > 0) {  
 res += BIT[idx];  
 idx -= idx & -idx;  
 }  
 **return** res;  
 }  
};  
  
*/\*\*  
 \* Fenwick tree class to compute and update range sum in O(log(N)).  
 \*  
 \* Note that the tree is is 1-indexed.  
 \*/***template**<**class** T>  
**class** range\_fenwick\_tree {  
 fenwick\_tree<T> M, C;  
  
**public**:**void** update(**int** l, **int** r, T val) {  
 M.update(l, val);  
 M.update(r + 1, -val);  
 C.update(l, -val \* (l - 1));  
 C.update(r + 1, val \* r);  
 }  
T **operator**[](**int** idx) {  
 **return** idx \* M[idx] + C[idx];  
 }  
};

## Segment Tree as Multiset

*/\*\*  
 \* Segment tree node struct.  
 \*/***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;  
 }  
};  
  
*/\*\*  
 \* Multiset that store integers in the range of [-N, N].  
 \* The multiset is implemented using segment tree.  
 \*  
 \* Note that the multiset is 0-indexed.  
 \* The most complex function in this class is done in time complexity of O(log(N)).  
 \*/***class** segment\_multiset {  
 **const int** N;  
 node\* nil, \* root;  
  
**public**:

segment\_multiset(**int** N) : N(N) {  
 root = nil = **new** node();  
 }

~segment\_multiset() {  
 destroy(root);  
 **delete** nil;  
 }  
**void** clear() {  
 destroy(root);  
 root = nil;  
 }  
**int** size() {  
 **return** root->size;  
 }  
**void** insert(**int** val, **int** cnt = 1) {  
 insert(root, val, cnt, -N, N);  
 }  
**int** erase(**int** val, **int** cnt = 1) {  
 **return** erase(root, val, cnt, -N, N);  
 }  
**int** count(**int** val) {  
 node\* cur = root;  
 **int** l = -N, r = N;  
  
 **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;  
 }  
**int operator**[](**int** idx) {  
 node\* cur = root;  
 **int** l = -N, r = N;  
  
 **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) {  
 node\* cur = root;  
 **int** l = -N, r = N, ret = 0;  
  
 **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

*// KMP longest match array.***int** F[N];  
 *// KMP failure function.***int** failure(**const char**\* pat, **char** cur, **int** len) {  
 **while** (len > 0 && cur != pat[len]) {  
 len = F[len - 1];  
 }  
 **return** len + (cur == pat[len]);  
}  
 *// Computes the length of the longest suffix ending at the i-th character  
// that match a prefix of the string, and fills the results in the global "F" array.***void** KMP(**const char**\* pat) {  
 **for** (**int** i = 1; pat[i]; ++i) {  
 F[i] = failure(pat, pat[i], F[i - 1]);  
 }  
}

## Z-Algorithm

*// Z-Algorithm longest match array.***int** Z[N];  
 *// Computes the length of the longest prefix starting at the i-th character  
// that match a prefix of the string, and fills the results in the global "Z" array.***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

*// The total length of all the string, and the size of the alphabet.***const int** N = 100100, ALPA\_SIZE = 255;  
  
**int** trie[N][ALPA\_SIZE]; *// The trie.***int** nodesCount; *// The number of nodes in the trie.***int** distinctWordsCount; *// The number of distinct word in the trie.***int** wordsCount[N]; *// Number of words sharing node "i".***int** wordsEndCount[N]; *// Number of words ending at node "i".  
  
// Initializes the trie. This must be called before each test case.***void** init() {  
 nodesCount = 0;  
 memset(trie, -1, **sizeof**(trie));  
}  
 *// Outs a new edge with character “c” from the given node if not exists .***int** addEdge(**int** id, **char** c) {  
 **int**& nxt = trie[id][c];  
 **if** (nxt == -1) {  
 nxt = ++nodesCount;  
 }  
 **return** nxt;  
}  
 *// Inserts a new word in the trie.***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);  
}  
 *// Removes a word from the trie assuming that it was added before.***void** erase(**const char**\* str) {  
 **int** cur = 0;  
  
 **for** (**int** i = 0; str[i]; ++i) {  
 wordsCount[cur]--;  
  
 **int** nxt = trie[cur][str[i]];  
  
 **if** (wordsCount[nxt] == 1) {  
 trie[cur][str[i]] = -1;  
 }  
  
 cur = nxt;  
 }  
  
 wordsCount[cur]--;  
 distinctWordsCount -= (--wordsEndCount[cur] == 0);  
}  
 *// Searches for a word in the trie and returns its number of occurrences.***int** 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 : the length of the string not the number of suffixes.   
// str : the string itself.*

*// Note that “str[n+1]” must be smaller than any value of “str”  
// SA : the suffix array, holding all the suffixes in lexicographical order.  
// suffixRank : array holding the order of the i-th suffix after sorting.  
// LCP : array holding the length of the longest common prefix between “SA[i]”*

*// and “SA[i - 1]”.***int** n, SA[N], suffixRank[N], LCP[N];  
**char** str[N];  
  
*// Temporary arrays needed while computing the suffix array.***int** sortedSA[N], sortedRanks[N], rankStart[N];  
 *// Comparator struct to be used internally from “buildSuffixArray” function.***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];  
 }  
};  
  
*// To be called internally from “buildSuffixArray” function.***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];  
 }  
}  
 *// Builds the suffix array of the given string in time complexity of O(n.log(n)).***void** buildSuffixArray() {**for** (**int** i = 0; i <= n; ++i) {  
 sortedSA[i] = i;  
 suffixRank[i] = str[i];  
 }  
sort(sortedSA, sortedSA + n + 1, comparator(0));  
 computeSuffixRanks(0);  
**for** (**int** h = 1; sortedRanks[n] != n; h <<= 1) {**for** (**int** i = 0; i <= n; ++i) {  
 **int** k = SA[i] - h;  
  
 **if** (k >= 0) {  
 sortedSA[rankStart[suffixRank[k]]++] = k;  
 }  
 }  
computeSuffixRanks(h);  
 }  
}  
 *// Computes the longest common prefix (LCP) for every two consecutive suffixes in the  
// suffix array in time complexity of O(n).***void** buildLCP() {  
 **int** cnt = 0;  
 **for** (**int** i = 0; i < n; ++i) {  
 **int** j = SA[suffixRank[i] - 1];  
 **while** (str[i + cnt] == str[j + cnt]) ++cnt;  
 LCP[suffixRank[i]] = cnt;  
 **if** (cnt > 0) --cnt;  
 }  
}

# Graphs

## Shortest Path (Floyd Warshal’s Algorithm)

**int** n; *// The number of nodes.***int** adj[N][N]; *// The graph adjacency matrix.***int** par[N][N]; *// par[u][v] : holds the parent node of "v" in the shortest*

*// path from "u" to "v".  
  
// Initializes the graph. Must be called before each test case.***void** init() {  
 **for** (**int** i = 0; i < n; ++i)  
 **for** (**int** j = 0; j < n; ++j)  
 adj[i][j] = (i == j ? 0 : oo), par[i][j] = i;  
}  
 *// Computes all-pair shortest paths using Floyd Warshall’s algorithm in O(n^3).***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];  
}  
 *// Checks whether the graph has negative cycles or not.***bool** checkNegativeCycle() {  
 **bool** ret = **false**;  
 **for** (**int** i = 0; i < n; ++i) {  
 ret = ret || (adj[i][i] < 0);  
 }  
 **return** ret;  
}  
 *// Prints the shortest path from node "u" to node "v".***void** printPath(**int** u, **int** v) {  
 **if** (u != v) {  
 printPath(u, par[u][v]);  
 }  
  
 printf(**"%d "**, v + 1);  
}

## Shortest Path (Bellman Ford’s Algorithm)

**int** n; *// The number of nodes.***int** dis[N]; *// dis[v] : holds the shortest distance between*

*// the source and node "v".*vector<pair<**int**, **int**>> edges[N]; *// The graph adjacency list.  
  
// Computes signle-source shortest paths using Bellman Ford’s algorithm in O(n^2).  
// And returns whether the graph contains negative cycles or not.***bool** bellmanFord(**int** src) {  
 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;   
 updated = 1;  
 }  
 }  
 }  
 }  
  
 **return** updated;  
}

## Shortest Path (Dijkstra’s Algorithm)

*/\*\*  
 \* Edge structs to holds the needed information about an edge.  
 \*/***struct** edge {  
 **int** to, weight;  
  
 edge() {}  
 edge(**int** t, **int** w) : to(t), weight(w) {}  
  
 **bool operator**<(**const** edge& rhs) **const** {  
 **return** weight > rhs.weight;  
 }  
};  
  
**int** n; *// The number of nodes.***int** dis[N]; *// dis[v] : holds the shortest distance between the source*

*// and node "v".*vector<edge> edges[N]; *// The graph adjacency list.  
  
// Computes signle-source shortest paths using Dijkstra’s algorithm in O(n.log(n)).***void** dijkstra(**int** src) {  
 priority\_queue<edge> q;  
 q.push(edge(src, 0));  
  
 memset(dis, 0x3F, **sizeof**(dis));  
  
 **while** (!q.empty()) {  
 **int** u = q.top().to;  
 **int** w = q.top().weight;  
 q.pop();  
  
 **if** (dis[u] <= w) {  
 **continue**;  
 }  
  
 dis[u] = w;  
  
 **for** (edge& e : edges[u]) {  
 **if** (w + e.weight < dis[e.to]) {  
 q.push(edge(e.to, w + e.weight));  
 }  
 }  
 }  
}

## Minimum Spanning Tree (Kruskal’s Algorithm)

*/\*\*  
 \* Edge structs to holds the needed information about an edge.  
 \*/***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);  
 }  
};  
  
**int** n; *// The number of nodes.***int** par[N]; *// The DSU parent array.*vector<edge> edges; *// The edges of the graph.  
  
// Finds and returns the set id of an element using the DSU data structure.***int** findSetId(**int** u) {  
 **return** u == par[u] ? u : par[u] = findSetId(par[u]);  
}  
 *// Computes and returns the minimum spanning tree of a weighted graph.***int** kruskalMST() {  
 **int** MST = 0;  
  
 sort(edges.begin(), edges.end());  
  
 **for** (**int** i = 1; i <= n; ++i) {  
 par[i] = i;  
 }  
  
 **for** (**auto**& e : edges) {  
 **int** x = findSetId(e.from);  
 **int** y = findSetId(e.to);  
  
 **if** (x != y) {  
 par[x] = y;  
 MST += e.weight;  
 }  
 }  
  
 **return** MST;  
}

## SCC (Kosaraju’s Algorithm)

**int** n; *// Number of nodes.***bool** vis[N]; *// Nodes visited array.*vector<**int**> edges[N]; *// Graph adjacency list.*vector<**int**> edgesT[N]; *// Transposed graph adjacency list (i.e. reversed edges).*vector<vector<**int**>> scc; *// Strongly connected components.  
  
// Sorts the nodes in a topological order.***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);  
}  
 *// Extracts the strongly connected components (SCC) of the given directed graph  
// using Kosaraju's algorithm, and fills them in the global "scc" vector".***void** kosaraju() {  
 vector<**int**> sortedNodes;  
  
 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]) {  
 vector<**int**> tmp;  
 topoSortDFS(u, edgesT, tmp);  
 scc.push\_back(tmp);  
 }  
 }  
}

## Topological Sort (Khan’s Algorithm)

**int** n; *// Number of nodes.***bool** vis[N]; *// Nodes visited array.*vector<**int**> edges[N]; *// Graph adjacency list.*vector<**int**> sortedNodes; *// List of topologically sorted nodes.  
  
// Sorts the graph in a topological order using Khan BFS algorithm,  
// and fills the result in the global "sortedNodes" vector***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);  
 }  
 }  
 }  
}

## Tree Diameter

**int** dis[N]; *// dis[v] : holds the shortest distance between the source*

*// and node "v".*vector<**int**> edges[N]; *// The graph adjacency list.  
  
// Returns the farthest node from the source node.***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;  
}  
  
*// Computes and returns the length of the diameter of the tree.***int** calcTreeDiameter(**int** root) {  
 **int** u = bfs(root);  
 **int** v = bfs(u);  
 **return** dis[v];  
}

## Bipartite Graph Check

**int** color[N]; *// The set each node belongs to.*vector<**int**> edges[N]; *// The graph adjacency list.  
  
// Do not call this directly.***bool** dfs(**int** u = 1) {  
 **for** (**int** v : edges[u]) {  
 **if** (color[v] == color[u]) {  
 **return false**;  
 }  
  
 **if** (color[v] == -1) {  
 color[v] = color[u] ^ 1;  
  
 **if** (!dfs(v)) {  
 **return false**;  
 }  
 }  
 }  
  
 **return true**;  
}  
  
*// Checks whether the given graph is bipartite or not.***bool** isBipartiteGraph() {  
 memset(color, -1, **sizeof**(color));  
 color[1] = 0;  
 **return** dfs();  
}

## Bridge Tree

**int** n; *// The number of nodes.*vector<**int**> edges[N]; *// The graph adjacency list.  
  
//  
// Bridge tree related variables  
//***int** T**int** root**int** par[N];**int** tin[N;**int** low[N];vector<**int**> tree[N];vector<pair<**int**, **int**>> bridges; *// Do not call this directly.***int** findSetId(**int** u) {  
 **return** (par[u] == u ? u : par[u] = findSetId(par[u]));  
}  
  
*// Do not call this directly.***void** findBridges(**int** u = 1, **int** p = -1) {  
 tin[u] = low[u] = ++T;  
  
 **for** (**int** 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** {  
 par[findSetId(u)] = findSetId(v);  
 }  
 }  
  
 low[u] = min(low[u], low[v]);  
 }  
}  
  
*// Builds the bridge tree of a graph in O(n+m).***void** buildBridgeTree() {  
 **for** (**int** i = 1; i <= n; ++i) {  
 par[i] = i;  
 }  
  
 findBridges();  
  
 **for** (**auto**& b : bridges) {  
 **int** u = findSetId(b.first);  
 **int** v = findSetId(b.second);  
  
 tree[u].push\_back(v);  
 tree[v].push\_back(u);  
  
 root = u;  
 }  
}

## LCA (Euler Walk + RMQ)

**int** n; *// The number of nodes.*vector<**int**> edges[N]; *// The graph adjacency list.  
  
//  
// LCA related variables.  
//***int** dep[N];  
**int** ST[LOG\_N][N << 1];  
**int** LOG[N << 1];  
**int** F[N];  
vector<**int**> E;  
  
*// Do not call this directly.***void** dfs(**int** u = 1, **int** p = -1, **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);  
 }  
 }  
}  
  
*// Do not call this directly.***void** buildRMQ() {  
 LOG[0] = -1;  
  
 **for** (**int** i = 0; i < E.size(); ++i) {  
 ST[0][i] = i;  
 LOG[i + 1] = LOG[i] + !(i & (i + 1));  
 }  
  
 **for** (**int** j = 1; (1 << j) <= E.size(); ++j) {  
 **for** (**int** i = 0; (i + (1 << j)) <= E.size(); ++i) {  
 **int** x = ST[j - 1][i];  
 **int** y = ST[j - 1][i + (1 << (j - 1))];  
 ST[j][i] = (dep[E[x]] < dep[E[y]]) ? x : y;  
 }  
 }  
}  
  
*// Builds the LCA data structure once per test case in O(n.log(n)).***void** buildLCA() {  
 dfs();  
 buildRMQ();  
}  
  
*// Do not call this directly.***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;  
}  
  
*// Returns the LCA of node "u" and node "v" in O(1).***int** getLCA(**int** u, **int** v) {  
 **return** E[query(F[u], F[v])];  
}  
  
*// Returns the distance between node "u" and node "v" in O(1).***int** getDistance(**int** u, **int** v) {  
 **return** dep[u] + dep[v] - 2 \* dep[getLCA(u, v)];  
}

## LCA (Parent Sparse Table)

**int** n; *// The number of nodes.*vector<**int**> edges[N]; *// The graph adjacency list.  
  
//  
// LCA related variables.  
//***int** dep[N];  
**int** par[LOG\_N][N];  
**int** LOG[N];  
  
*// Do not call this directly.***void** dfs(**int** u = 1, **int** p = -1, **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);  
 }  
 }  
}  
  
*// Do not call this directly.***void** computeLog() {  
 LOG[0] = -1;  
 **for** (**int** i = 1; i <= n; ++i) {  
 LOG[i] = LOG[i - 1] + !(i & (i - 1));  
 }  
}  
  
*// Builds the LCA data structure once per test case in O(n.log(n)).***void** buildLCA() {  
 dfs();  
 computeLog();  
}  
  
*// Returns the k-th ancestor of a node "u".***int** getAncestor(**int** u, **int** k) {  
 **while** (k > 0) {  
 **int** x = k & -k;  
 k -= x;  
 u = par[LOG[x]][u];  
 }  
 **return** u;  
}  
  
*// Returns the LCA of node "u" and node "v" in O(log(n)).***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];  
}  
  
*// Returns the distance between node "u" and node "v" in O(log(n)).***int** getDistance(**int** u, **int** v) {  
 **return** dep[u] + dep[v] - 2 \* dep[getLCA(u, v)];  
}

## Max Flow (Edmonds Karp’s Algorithm)

**int** n, m; *// The number of nodes and number of edges***int** edgeId; *// The next edge id to be inserted.***int** head[N]; *// head[u] : the id of the last edge added from node "u".***int** nxt[M]; *// nxt[e] : the next edge id pointed from the same node  
 // as "e".***int** to[M]; *// to[e] : the id of the node pointed by edge "e".***int** capacity[M]; *// capacity[e] : the maximum capacity of edge "e".***int** flow[M]; *// flow[u] : the current flow of edge "e".***int** src, snk; *// The id of source and sink nodes.***int** dist[N]; *// dist[u] : the shortest distance between the source and  
 // node "u".***int** from[N]; *// from[u] : the id of the edge that leads to node "u" in the  
 // path from source to sink.   
  
// Initializes the graph. Must be called before each test case.***void** init() {  
 edgeId = 0;  
 memset(head, -1, **sizeof**(head));  
}  
  
*// Adds a new directed edge in the graph from node "f" to node "t"  
// with maximum capacity "c".***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;  
}  
  
*// Adds a new augmented edge in the graph between node "f" and node "t"  
// with maximum capacity "w".***void** addAugEdge(**int** f, **int** t, **int** c) {  
 addEdge(f, t, c);  
 addEdge(t, f, 0);  
}  
  
*// Do not call this directly.***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** (f >= c) {  
 **continue**;  
 }  
  
 **if** (dist[v] == -1) {  
 dist[v] = dist[u] + 1;  
 from[v] = e;  
 q.push(v);  
 }  
  
 **if** (v == snk) {  
 **return true**;  
 }  
 }  
 }  
  
 **return false**;  
}  
  
*// Do not call this directly.***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;  
}  
  
*// Returns the the maximum flow/minimum cut of the graph.***int** maxFlow() {  
 **int** f = 0;  
  
 **while** (findPath()) {  
 f += augmentPath();  
 }  
  
 **return** f;  
}

# Math

## GCD

*// Computes the greatest common divisors GCD(a, b).***template**<**class** T>  
T gcd(T a, T b) {  
 **while** (b) {  
 **int** tmp = a % b;  
 a = b;  
 b = tmp;  
 }  
 **return** a;  
}

## LCM

*// Computes the least common multiple LCM(a, b).***template**<**class** T>  
T lcm(T a, T b) {  
 **return** a / gcd(a, b) \* b;  
}

## Extended Euclid

*// Computes the coeffs. of the smallest positive linear combination of "a" and "b".  
// (i.e. GCD(a, b) = s.a + t.b).***template**<**class** T>  
pair<T, T> extendedEuclid(T a, T b) {  
 **if** (b == 0) {  
 **return** {1, 0};  
 }  
  
 pair<T, T> p = extendedEuclid(b, a % b);  
  
 T s = p.first;  
 T t = p.second;  
  
 **return** {t, s - t \* (a / b)};  
}

## Fast Power

*// Computes ((base^exp) % mod).***template**<**class** T>  
T power(T base, T exp, T mod) {  
 T ans = 1;  
 base %= mod;  
  
 **while** (exp > 0) {  
 **if** (exp & 1) ans = (ans \* base) % mod;  
 exp >>= 1;  
 base = (base \* base) % mod;  
 }  
  
 **return** ans;  
}

## Modular Inverse

*// Computes the modular inverse of "a" modulo "m".***template**<**class** T>  
T modInverse(T a, T m) {  
 **return** power(a, m - 2, m);  
}

## Combinations (nCr)

*// Computes "n" choose "r".***int** nCr(**int** n, **int** r) {  
 **if** (n < r)  
 **return** 0;  
  
 **if** (r == 0)  
 **return** 1;  
  
 **return** n \* nCr(n - 1, r - 1) / r;  
}

## Pascal’s Triangle

*// comb[n][r] : holds the value of "n" choose "r" modulo "mod".***int** comb[N][N];  
 *// Builds Pascal's triangle for computing combinations (i.e. "nCr").***void** buildPT(**int** n, **int** mod) {  
 **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;  
}

## Prime Check

*// Checks whether an integer is prime or not.***template**<**class** T>  
**bool** isPrime(T 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;  
}

## Prime Check (Miller Rabin’s Algorithm)

*// Do not call it directly.***template**<**class** T>  
**bool** millerRabin(T k, T q) {  
 T n = (1LL << k) \* q + 1;  
 T a = 2 + rand() % (n - 2);  
 T x = power(a, q, n);  
  
 **if** (x == 1) {  
 **return true**;  
 }  
  
 **while** (k--) {  
 **if** (x == n - 1) {  
 **return true**;  
 }  
  
 x = (x \* x) % n;  
 }  
  
 **return false**;  
}  
 *// Checks whether an integer is prime or not using the probabilistic method of  
// Miller Rabin algorithm in O(t.log(n)).***template**<**class** T>  
**bool** isPrimeMillerRabin(T n, **int** t = 10) {  
 **if** (n == 2) {  
 **return** 1;  
 }  
  
 **if** (n < 2 || n % 2 == 0) {  
 **return** 0;  
 }  
T k = 0;  
 T q = n - 1;  
 **while** ((q & 1) == 0) {  
 k++;  
 q >>= 1;  
 }  
**while** (t--) {  
 **if** (!millerRabin(k, q)) {  
 **return false**;  
 }  
 }  
  
 **return true**;  
}

## Generate Primes

*// prime[i] : true if integer "i" is prime; false otherwise.***bool** prime[N];  
 *// Generates all the prime numbers of the integers from 1 to "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**;  
 }  
 }  
}

## Generate Prime Divisors

*// divs[i] : holds a list of all the prime divisors of integer "i".*vector<**int**> primeDivs[N];  
 *// Generates all the prime divisors of the integers from 1 to "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);  
 }  
 }  
}

## Generate Divisors

*// Computes all the divisors of a positive integer.***template**<**class** T>  
vector<T> getDivisors(T n) {  
 vector<T> divs;  
 **for** (T i = 1; i \* i <= n; ++i) {  
 **if** (n % i != 0) **continue**;  
 divs.push\_back(i);  
 **if** (i \* i == n) **continue**;  
 divs.push\_back(n / i);  
 }  
 sort(divs.begin(), divs.end());  
 **return** divs;  
}  
 *// divs[i] : holds a list of all the divisors of integer "i".*vector<**int**> divs[N];  
 *// Generates all the divisors of the integers from 1 to "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

*// The array to compute its LIS and its length.***int** n, a[N];  
 *// Computes and returns the length of the longest increasing subsequence (LIS) of  
// the global array "a" in time complexity of O(n.log(n)).***int** getLIS() {  
 **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;  
}