Contents

[Strings 3](#_Toc172231244)

[KMP 3](#_Toc172231245)

[Z function 3](#_Toc172231246)

[Manacher 3](#_Toc172231247)

[Trie 3](#_Toc172231248)

[Hashing 4](#_Toc172231249)

[Suffix array (Geeks) 5](#_Toc172231250)

[Suffix array & LCP (BucketPotato) 6](#_Toc172231251)

[Aho-corasick algorithm 7](#_Toc172231252)

[Data Structures 9](#_Toc172231253)

[DSU 9](#_Toc172231254)

[DSU with rollback 9](#_Toc172231255)

[Fenwick tree 10](#_Toc172231256)

[2D Fenwick tree 11](#_Toc172231257)

[Segment tree 11](#_Toc172231258)

[Segment tree with lazy 12](#_Toc172231259)

[Sparse table 12](#_Toc172231260)

[SQRT decomposition 13](#_Toc172231261)

[Mo’s algorithm 14](#_Toc172231262)

[Mo’s algorithm (faster) 14](#_Toc172231263)

[Priority queue custom sort 16](#_Toc172231264)

[Set & multiset custom sort 16](#_Toc172231265)

[Ordered set & multiset 16](#_Toc172231266)

[Monotonic queue 17](#_Toc172231267)

[2D prefix sum 17](#_Toc172231268)

[2D partial sum 18](#_Toc172231269)

[Math 18](#_Toc172231270)

[Big int functions 18](#_Toc172231271)

[Rand 19](#_Toc172231272)

[Matrix power 19](#_Toc172231273)

[Calculate the determinant of matrix using gauss O(n3) 20](#_Toc172231274)

[Find matrix inverse O(n3) 21](#_Toc172231275)

[MillerRabin (Check if a number is prime) O(log n) 21](#_Toc172231276)

[Phi 22](#_Toc172231277)

[Phi (preprocessing) 22](#_Toc172231278)

[Sieve 22](#_Toc172231279)

[SPF 23](#_Toc172231280)

[Divisors (n) 23](#_Toc172231281)

[Divisors [2, n] 23](#_Toc172231282)

[Prime factors (n) 23](#_Toc172231283)

[Sum [l, r] 24](#_Toc172231284)

[Fast Power (log p) 24](#_Toc172231285)

[Factorial & nCr & nPr (preprocessing) 24](#_Toc172231286)

[nCr & nPr 25](#_Toc172231287)

[Fibonacci O(log n) 25](#_Toc172231288)

[Divisabilty rules 25](#_Toc172231289)

[Catalan number 26](#_Toc172231290)

[Trees 26](#_Toc172231291)

[LCA (Binary lifting) 26](#_Toc172231292)

[LCA with segment tree (Euler tour) 27](#_Toc172231293)

[HLD 28](#_Toc172231294)

[Centroid decomposition 30](#_Toc172231295)

[Small to large 31](#_Toc172231296)

[Graphs 32](#_Toc172231297)

[Floyd-Warshall Algorithm O(n3) 32](#_Toc172231298)

[Bellman Ford O(EV) 32](#_Toc172231299)

[SPFA 33](#_Toc172231300)

[SCC using Kosaraju's Algorithm 33](#_Toc172231301)

[2-sat 34](#_Toc172231302)

[Articulation points 35](#_Toc172231303)

[Bridges 35](#_Toc172231304)

[Minimum spanning arborescence (MSA) - Zhu-Liu algorithm O(EV) 36](#_Toc172231305)

[Ford-Fulkerson Max-flow O(E \* F) 37](#_Toc172231306)

[Dinic Max-flow (um\_nik) O(V2 E) 38](#_Toc172231307)

[Max-flow min-cost 39](#_Toc172231308)

# Strings

## KMP

vector<int> prefix\_function(string s) {

int n = (int)s.length();

vector<int> pi(n);

for (int i = 1; i < n; i++) {

int j = pi[i - 1];

while (j > 0 && s[i] != s[j])

j = pi[j - 1];

if (s[i] == s[j])

j++;

pi[i] = j;

}

return pi;

}

## Z function

vector<int> Z\_function(const string& s) {

int n = s.size();

vector<int> Z(n);

for (int i = 1, l = 0, r = 0; i < n; i++) {

if (i <= r)

Z[i] = min(Z[i - l], r - i + 1);

while (i + Z[i] < n && s[Z[i]] == s[i + Z[i]])

Z[i]++;

if (i + Z[i] - 1 > r) l = i, r = i + Z[i] - 1;

}

return Z;

}

## Manacher

pair<vector<int>, vector<int>> manacher(const string& s) {

int n = (int)s.size();

vector<int> d1(n);

for (int i = 0, l = 0, r = -1; i < n; i++) { // Find odd palindromes

int k = (i > r) ? 1 : min(d1[r - i + l], r - i + 1);

while (i - k >= 0 && i + k < n && s[i - k] == s[i + k]) k++;

d1[i] = k--;

if (i + k > r) l = i - k, r = i + k;

}

vector<int> d2(n);

for (int i = 0, l = 0, r = -1; i < n; i++) { // Find even palindromes

int k = (i > r) ? 0 : min(d2[r - i + l + 1], r - i + 1);

while (i - k - 1 >= 0 && i + k < n && s[i - k - 1] == s[i + k]) k++;

d2[i] = k--;

if (i + k > r) l = i - k - 1, r = i + k;

}

return {d1, d2};

}

## Trie

struct Trie {

Trie\* child[26];

int prevents, endCnt;

Trie() {

memset(child, 0, sizeof child);

prefCnt = endCnt = 0;

}

void insert(const string& s, int cnt = 1, int i = 0) {

prefCnt += cnt;

if (i == s.size()) {

endCnt += cnt;

return;

}

int cur = s[i] - 'a';

if (!child[cur]) {

child[cur] = new Trie();

}

child[cur]->insert(s, cnt, i + 1);

}

int erase(const string& s, int cnt = 1, int i = 0) {

if (i == s.size()) {

int minC = min(cnt, endCnt);

endCnt -= minC, prefCnt -= minC;

return minC;

}

int cur = s[i] - 'a';

int minC = child[cur]->erase(s, cnt, i + 1);

prefCnt -= minC;

if (child[cur]->prefCnt == 0) {

delete child[cur];

child[cur] = nullptr;

}

return minC;

}

pair<int, int> count(const string& s, int i = 0) {

if (i == s.size()) return { prefCnt, endCnt };

int cur = s[i] - 'a';

if (!child[cur]) return { 0, 0 };

return child[cur]->count(s, i + 1);

}

};

## Hashing

const int N = 300300;

const int m1 = 1e9 + 7, b1 = 17;

const int m2 = 1e9 + 9, b2 = 31;

int pw1[N], inv1[N];

int pw2[N], inv2[N];

void pre(int n = N) {

pw1[0] = inv1[0] = 1;

pw2[0] = inv2[0] = 1;

int iv1 = inv(b1, m1);

int iv2 = inv(b2, m2);

for (int i = 1; i < n; i++) {

pw1[i] = mul(pw1[i - 1], b1, m1);

inv1[i] = mul(inv1[i - 1], iv1, m1);

pw2[i] = mul(pw2[i - 1], b2, m2);

inv2[i] = mul(inv2[i - 1], iv2, m2);

}

}

struct Hashing {

int n;

string s;

vector<int> h1, h2;

Hashing(const string& s\_) {

s = s\_;

n = s.size();

h1.resize(n), h2.resize(n);

h1[0] = h2[0] = s[0] - '0' + 1;

for (int i = 1; i < n; i++) {

h1[i] = add(h1[i - 1], mul(s[i] - '0' + 1, pw1[i], m1), m1);

h2[i] = add(h2[i - 1], mul(s[i] - '0' + 1, pw2[i], m2), m2);

}

}

pair<int, int> get(int l, int r) {

assert(l >= 0 && r < n && l <= r);

int x = h1[r], y = h2[r];

if (l) {

x = mul(sub(x, h1[l - 1], m1), inv1[l], m1);

y = mul(sub(y, h2[l - 1], m2), inv2[l], m2);

}

return { x, y };

}

static pair<int, int> concat(pair<int, int> h1, pair<int, int> h2, int h1\_sz) {

h1.first = add(h1.first, mul(h2.first, pw1[h1\_sz]));

h1.second = add(h1.second, mul(h2.second, pw2[h1\_sz]));

return h1;

}

};

## Suffix array (Geeks)

struct suffix {

int index, rank[2];

};

int cmp(struct suffix a, struct suffix b) {

return (a.rank[0] == b.rank[0]) ? (a.rank[1] < b.rank[1] ? 1 : 0) :

(a.rank[0] < b.rank[0] ? 1 : 0);

}

vector<int> suffixArray(const string& s) { // O(n log)

int n = s.size();

struct suffix suf[n];

for (int i = 0; i < n; i++) {

suf[i].index = i;

suf[i].rank[0] = s[i] - 'a';

suf[i].rank[1] = ((i + 1) < n) ? (s[i + 1] - 'a') : -1;

}

sort(suf, suf + n, cmp);

vector<int> ind(n);

for (int k = 4; k < 2 \* n; k = k \* 2) {

int rank = 0;

int prev\_rank = suf[0].rank[0];

suf[0].rank[0] = rank;

ind[suf[0].index] = 0;

for (int i = 1; i < n; i++) {

if (suf[i].rank[0] == prev\_rank && suf[i].rank[1] == suf[i - 1].rank[1]) {

prev\_rank = suf[i].rank[0];

suf[i].rank[0] = rank;

}

else {

prev\_rank = suf[i].rank[0];

suf[i].rank[0] = ++rank;

}

ind[suf[i].index] = i;

}

for (int i = 0; i < n; i++) {

int nxt = suf[i].index + k / 2;

suf[i].rank[1] = (nxt < n) ? suf[ind[nxt]].rank[0] : -1;

}

sort(suf, suf + n, cmp);

}

vector<int> retSuf(n);

for (int i = 0; i < n; i++) retSuf[i] = suf[i].index;

return retSuf;

}

## Suffix array & LCP (BucketPotato)

struct SuffixArray {

int n, LOG;

vector<char> s;

vector<int> sor, fv, lcparr, lo;

vector<vector<int>> sparse;

SuffixArray(string S) {

n = S.size();

s.resize(n + 1);

S.push\_back('\0');

for (int i = 0; i <= n; i++)

s[i] = S[i];

buildSA();

buildLCP();

}

void buildSA() {

sor.resize(n + 1);

fv.resize(n + 1);

iota(sor.begin(), sor.end(), 0);

sort(sor.begin(), sor.end(), [&](int a, int b) {

return (s[a] < s[b]);

});

int co = 0;

for (int i = 0; i <= n; i++) {

if (i && s[sor[i]] == s[sor[i - 1]]) co--;

fv[sor[i]] = co++;

}

int cv = 1;

while (cv < n) {

vector<int> nv(n + 1), cnt(n + 1), nfv(n + 1);

for (int i = 0; i <= n; i++) if (fv[i] < n) cnt[fv[i] + 1]++;

for (int i = 1; i <= n; i++) cnt[i] += cnt[i - 1];

for (int i = 0; i <= n; i++) {

int pi = (sor[i] + (n + 1) - cv) % (n + 1);

nv[cnt[fv[pi]]++] = pi;

}

int nco = 0;

for (int i = 0; i <= n; i++) {

if (i && fv[nv[i - 1]] == fv[nv[i]] &&

fv[(nv[i - 1] + cv) % (n + 1)] == fv[(nv[i] + cv) % (n + 1)])

nco--;

nfv[nv[i]] = nco++;

}

swap(nv, sor);

swap(nfv, fv);

cv \*= 2;

}

}

void buildLCP() {

lcparr.resize(n + 1, 0);

for (int i = 0; i < n; i++) {

int ci = fv[i];

int cv = 0;

if (i) cv = max(cv, lcparr[fv[i - 1]] - 1);

int pi = sor[fv[i] - 1];

while (pi + cv < n && i + cv < n) {

if (s[pi + cv] != s[i + cv]) break;

cv++;

}

lcparr[ci] = cv;

}

LOG = 1;

while ((1 << LOG) <= n) LOG++;

LOG++;

sparse.resize(LOG, vector<int>(n + 1, 0));

for (int i = 0; i <= n; i++) sparse[0][i] = lcparr[i];

for (int i = 1; i < LOG; i++) {

for (int j = 0; j <= n; j++) {

sparse[i][j] = sparse[i - 1][j];

if (j + (1 << (i - 1)) <= n)

sparse[i][j] = min(sparse[i][j], sparse[i - 1][j + (1 << (i - 1))]);

}

}

lo.resize(n + 1, 0);

int cv = -1, nv = 1;

for (int i = 1; i <= n; i++) {

if (i == nv) cv++, nv \*= 2;

lo[i] = cv;

}

}

int lcp(int a, int b) {

if (a == b) return (n - a);

a = fv[a];

b = fv[b];

if (a > b) swap(a, b);

a++;

int l = b - a + 1;

return min(sparse[lo[l]][a], sparse[lo[l]][b - (1 << lo[l]) + 1]);

}

};

## Aho-corasick algorithm

const int MAX = 26 + 1;//+1 to assign starting from 1

int cIdx(char c) {

return 1 + c - 'a';

}

struct Trie {

Trie\* child[MAX];

Trie\* fail;

vector<int> patIdx;

vector<char> chars;

Trie() {

memset(child, 0, sizeof child);

}

void insert(const string& s, int idx, int i = 0) {

if (i == s.size()) {

patIdx.push\_back(idx);

return;

}

int cur = cIdx(s[i]);

if (!child[cur]) {

child[cur] = new Trie();

chars.push\_back(s[i]);

}

child[cur]->insert(s, idx, i + 1);

}

};

void move(Trie\*& k, int c) {

while (!k->child[c]) k = k->fail;

k = k->child[c];

}

struct Aho {

Trie\* root;

int p\_sz;

vector<string> pat;

Trie\* buildAhoTree() {

Trie\* root = new Trie();

for (int i = 0; i < p\_sz; i++) {

root->insert(pat[i], i);

}

queue<Trie\*> q;

for (int i = 0; i < MAX; i++) {

if (root->child[i]) q.push(root->child[i]), root->child[i]->fail = root;

else root->child[i] = root;

}

Trie\* cur;

while (!q.empty()) {

cur = q.front();

q.pop();

for (int i = 0; i < cur->chars.size(); i++) {

char ch = cIdx(cur->chars[i]);

q.push(cur->child[ch]);

Trie\* k = cur->fail;

move(k, ch);

cur->child[ch]->fail = k;

for (auto j : k->patIdx) {

cur->child[ch]->patIdx.push\_back(j);

}

}

}

return root;

}

Aho(const vector<string>& p) {

p\_sz = p.size(), pat = p;

root = buildAhoTree();

}

vector<vector<int>> find\_patterns(const string& s) { // find all occurrences of each pattern

vector<vector<int>> res(p\_sz);

Trie\* k = root;

for (int i = 0; s[i]; i++) {

move(k, cIdx(s[i]));

for (auto& j : k->patIdx) res[j].push\_back(i);

}

for (int i = 0; i < p\_sz; i++) {

for (int& j : res[i]) j -= (int)pat[i].size() - 1;

}

return res;

}

};

# Data Structures

## DSU

struct DSU {

vector<int> par, size;

int groups;

DSU(int n) {

par.resize(n + 1);

size.resize(n + 1, 1);

iota(par.begin(), par.end(), 0);

groups = n;

}

int Leader(int child) {

return par[child] == child ? child : par[child] = Leader(par[child]);

}

bool mergeGroups(int x, int y) {

x = Leader(x), y = Leader(y);

if (x == y) return false;

if (size[y] > size[x]) swap(x, y);

par[y] = x;

size[x] += size[y];

groups--;

return true;

}

int getSize(int x) {

return size[Leader(x)];

}

};

## DSU with rollback

class DSU {

private:

int n;

vector<int>parent, siz;

stack <pair<int, int>>st;

public:

int cmp;

// constructor

DSU(int nt) {

parent.resize(nt + 5);

siz.resize(nt + 5);

this->n = nt;

cmp = n;

for (int i = 0; i < parent.size(); i++) {

parent[i] = i;

siz[i] = 1;

}

}

// find leader

int leader(int child) {

if (child == parent[child]) {

return child;

}

return leader(parent[child]);

}

// merge groups

bool merge(int ch1, int ch2) {

int leader1 = leader(ch1);

int leader2 = leader(ch2);

if (leader1 == leader2) {

return false; // not able to merge

}

if (siz[leader1] > siz[leader2]) {

parent[leader2] = leader1;

siz[leader1] += siz[leader2];

st.push({ leader1,leader2 });

}

else {

parent[leader1] = leader2;

siz[leader2] += siz[leader1];

st.push({ leader2,leader1 });

}

cmp--;

return true;

}

// get size

int getSize(int child) {

int lead = leader(child);

return siz[lead];

}

void stackIt(int a = -1, int b = -1) {

st.push({ a,b });

}

void undo() {

while (st.top().first != -1) {

int a = st.top().first;

int b = st.top().second;

st.pop();

parent[b] = b;

siz[a] -= siz[b];

cmp++;

}

st.pop();

}

};

## Fenwick tree

template<typename T> struct Fenwick {

int n;

vector<T> fen;

Fenwick(int sz) {

n = sz, fen.resize(n);

}

void add(int idx, T x) {

while (idx < n) fen[idx] += x, idx |= (idx + 1);

}

T get(int idx) {

T ret = 0;

while (idx >= 0) ret += fen[idx], idx &= (idx + 1), idx--;

return ret;

}

};

## 2D Fenwick tree

template<typename T> struct Fenwick\_2D {

int n, m;

vector<vector<T>> fen;

Fenwick\_2D(int \_n, int \_m) {

n = \_n, m = \_m;

fen.resize(n, vector<T>(m, 0));

}

void add(int i, int j, T x) {

while (i < n) {

int j2 = j;

while (j2 < m) fen[i][j2] += x, j2 |= (j2 + 1);

i |= (i + 1);

}

}

T get(int i, int j) {

T ret = 0;

while (i >= 0) {

int j2 = j;

while (j2 >= 0) ret += fen[i][j2], j2 &= (j2 + 1), j2--;

i &= (i + 1), i--;

}

return ret;

}

T Query(int x1, int y1, int x2, int y2) {

return get(x2, y2) - get(x2, y1 - 1) - (get(x1 - 1, y2) - get(x1 - 1, y1 - 1));

}

};

## Segment tree

template<typename T> struct segTree {

vector<T> seg;

int n;

T bad;

function<T(T, T)> merge;

static T Min(T a, T b) { return min(a, b); }

static T Max(T a, T b) { return max(a, b); }

segTree(int n\_, T bad\_, function<T(T, T)> op) {

n = n\_, bad = bad\_, merge = op;

seg.resize(4 \* n + 2);

}

T get(int l, int r, int idx = 0, int lw = 0, int hgh = -1) {

if (hgh == -1) hgh = n - 1;

if (lw >= l && hgh <= r) return seg[idx];

if (hgh < l || lw > r) return bad;

int md = (lw + hgh) >> 1;

return merge(get(l, r, 2 \* idx + 1, lw, md), get(l, r, 2 \* idx + 2, md + 1, hgh));

}

void update(int id, const T& val, int idx = 0, int lw = 0, int hgh = -1) {

if (hgh == -1) hgh = n - 1;

if (lw == hgh) return void(seg[idx] = val);

int md = (lw + hgh) >> 1;

if (id <= md) update(id, val, 2 \* idx + 1, lw, md);

else update(id, val, 2 \* idx + 2, md + 1, hgh);

seg[idx] = merge(seg[2 \* idx + 1], seg[2 \* idx + 2]);

}

};

## Segment tree with lazy

struct LST {

vector<ll> seg, lazy;

int n;

ll bad;

function<ll(ll, ll)> merge;

LST(int n\_, ll bad\_, function<ll(ll, ll)> op) {

n = n\_, bad = bad\_, merge = op;

seg.resize(4 \* n + 2, 0), lazy.resize(4 \* n + 2, 0);

}

void push(int idx, int lw, int hgh, ll val) {

seg[idx] += (hgh - lw + 1) \* val;

if (lw != hgh)

lazy[2 \* idx + 1] += val, lazy[2 \* idx + 2] += val;

}

ll get(int l, int r, int idx = 0, int lw = 0, int hgh = -1) {

if (hgh == -1) hgh = n - 1;

push(idx, lw, hgh, lazy[idx]), lazy[idx] = 0;

if (lw >= l && hgh <= r) return seg[idx];

if (hgh < l || lw > r) return bad;

int md = (lw + hgh) >> 1;

return merge(get(l, r, 2 \* idx + 1, lw, md), get(l, r, 2 \* idx + 2, md + 1, hgh));

}

void updateRange(int l, int r, ll val, int idx = 0, int lw = 0, int hgh = -1) {

if (hgh == -1) hgh = n - 1;

push(idx, lw, hgh, lazy[idx]), lazy[idx] = 0;

if (hgh < l || lw > r || lw > hgh) return;

if (lw >= l && hgh <= r)

return void(push(idx, lw, hgh, val));

int md = (lw + hgh) >> 1;

updateRange(l, r, val, 2 \* idx + 1, lw, md);

updateRange(l, r, val, 2 \* idx + 2, md + 1, hgh);

seg[idx] = merge(seg[2 \* idx + 1], seg[2 \* idx + 2]);

}

};

## Sparse table

template<typename T> struct SparseTable {

int n;

function<T(T, T)> merge;

vector<vector<T>> ST;

T bad;

static vector<int> lg;

static T Min(T a, T b) { return min(a, b); }

static T Max(T a, T b) { return max(a, b); }

SparseTable(const vector<T>& data, function<T(T, T)> \_merge, T \_bad) {

n = data.size();

if (lg.empty()) lg = { 0, 0 };

while (lg.size() <= 2 \* n) lg.push\_back(lg[lg.size() >> 1] + 1);

merge = \_merge;

bad = \_bad;

ST = vector<vector<T>>(n, vector<T>(lg[n] + 1));

for (int i = 0; i < n; i++) ST[i][0] = data[i];

for (int pw = 1; (1 << pw) <= n; pw++)

for (int i = 0; i + (1 << pw) <= n; i++)

ST[i][pw] = merge(ST[i][pw - 1], ST[i + (1 << (pw - 1))][pw - 1]);

}

T get1(int l, int r) {

int sz = r - l + 1;

T ret = bad;

for (int i = lg[sz]; i >= 0; i--) {

if (sz >> i & 1) {

ret = merge(ret, ST[l][i]);

l += 1 << i;

}

}

return ret;

}

T get2(int l, int r) {

int sz = r - l + 1;

int pw = lg[sz];

return merge(ST[l][pw], ST[r - (1 << pw) + 1][pw]);

}

};

template<typename T>

vector<int> SparseTable<T>::lg;

## SQRT decomposition

template<typename T> struct SQRT\_decomp {

int n, SQ;

vector<vector<T>> b;

vector<T> blk;

vector<T> a;

T bad;

function<T(T, T)> merge;

SQRT\_decomp(const vector<T>& \_a, T \_bad, function<T(T, T)> \_merge) {

a = \_a;

n = a.size();

SQ = sqrt(n);

if (SQ \* SQ != n) SQ++;

blk.resize(SQ);

merge = \_merge;

bad = \_bad;

build();

}

void build() {

fill(blk.begin(), blk.end(), bad);

for (int i = 0; i < n; i++) {

int blkIdx = i / SQ;

blk[blkIdx] = merge(blk[blkIdx], a[i]);

}

}

ll get(int l, int r) {

ll ret = bad;

for (int i = l; i <= r; i++) {

if (i % SQ == 0 && i + SQ - 1 <= r)

ret = merge(ret, blk[i / SQ]), i += SQ - 1;

else

ret = merge(ret, a[i]);

}

return ret;

}

void update(int idx, const T& val) {

int blkIdx = idx / SQ;

a[idx] = val;

int l = blkIdx \* SQ, r = min(l + SQ - 1, n - 1);

blk[blkIdx] = bad;

for (int i = l; i <= r; i++) {

blk[blkIdx] = merge(blk[blkIdx], a[i]);

}

}

};

## Mo’s algorithm

const int MAX\_N = 2e5 + 5;

const int MAX\_Q = 2e5 + 5;

const int MAX\_a = 1e6;

const int SQ = sqrt(MAX\_N) + 5;

struct Query {

int l, r, B\_idx, Q\_idx;

Query() {}

Query(int \_l, int \_r, int idx) {

l = \_l, r = \_r, Q\_idx = idx, B\_idx = l / SQ;

}

bool operator<(const Query& q) const {

if (B\_idx != q.B\_idx) return B\_idx < q.B\_idx;

return r < q.r;

}

};

int a[MAX\_N], res;

int freq[MAX\_N];

Query Q[MAX\_Q];

int ans[MAX\_Q];

// change Add and Remove functions

void Add(int idx) {

if (freq[a[idx]] == 0) res++;

freq[a[idx]]++;

}

void Remove(int idx) {

freq[a[idx]]--;

if (freq[a[idx]] == 0) res--;

}

void MO(int q) {

sort(Q, Q + q);

int l = 1, r = 0;

for (int i = 0; i < q; i++) {

while (l < Q[i].l) Remove(l++);

while (l > Q[i].l) Add(--l);

while (r < Q[i].r) Add(++r);

while (r > Q[i].r) Remove(r--);

ans[Q[i].Q\_idx] = res;

}

}

## Mo’s algorithm (faster)

constexpr int logn = 20;

constexpr int maxn = 1 << logn;

long long hilbertorder(int x, int y)

{

long long d = 0;

for (int s = 1 << (logn - 1); s; s >>= 1)

{

bool rx = x & s, ry = y & s;

d = d << 2 | rx \* 3 ^ static\_cast<int>(ry);

if (!ry)

{

if (rx)

{

x = maxn - x;

y = maxn - y;

}

swap(x, y);

}

}

return d;

}

int B; // sqrt the size

struct query {

int l, r, idx;

int ord;

const bool operator<(const query& o) const {

if (l / B == o.l / B) {

if ((l / B) % 2 == 0) return r < o.r;

else return r > o.r;

}

else {

return l < o.l;

}

}

// or

void getOrder() {

this->ord = hilbertorder(l, r);

// then sort according to order in main

}

};

//

void Run(vector<query>& qu) {

sort(all(qu));

int cnt = 0;

int l = 0, r = 0;

for (auto& [lq, rq, idx] : qu) {

while (lq <= l - 1) {

//add();

l--;

}

while (lq > l) {

//remove();

l++;

}

while (rq >= r + 1) {

//add();

r++;

}

while (rq < r) {

//remove();

r--;

}

ans[idx] = cnt;

}

}

## Priority queue custom sort

class Compare

{

public:

bool operator()(int below, int above)

{

return below < above;

}

};

priority\_queue<int, vector<int>, Compare> pq;

## Set & multiset custom sort

struct Compare

{

bool operator()(const int& x, const int& y) const

{

return x < y;

}

};

set<int, Compare> st;

multiset<int, Compare> ms;

## Ordered set & multiset

#include <ext/pb\_ds/assoc\_container.hpp>

#include <ext/pb\_ds/tree\_policy.hpp>

using namespace \_\_gnu\_pbds;

template<typename T>

using ordered\_set = tree<T, null\_type, less<T>, rb\_tree\_tag, tree\_order\_statistics\_node\_update>;

template<typename T>

using ordered\_multiset = tree<T, null\_type, less\_equal<T>, rb\_tree\_tag, tree\_order\_statistics\_node\_update>;

template<class T> struct Multiset {

ordered\_multiset<T> ms;

void insert(const T& x) {

ms.insert(x);

}

bool exist(const T& x) const {

auto it = ms.upper\_bound(x);

if (it == ms.end()) return false;

return \*it == x;

}

bool erase(const T& x) {

if (!exist(x)) return false;

ms.erase(ms.upper\_bound(x));

return true;

}

T operator [] (int p) const {

assert(p >= 0 && p < (int)ms.size());

return \*ms.find\_by\_order(p);

}

typename ordered\_multiset<T>::iterator begin() { return ms.begin(); }

typename ordered\_multiset<T>::iterator end() { return ms.end(); }

int first(const T& x) const {

if (!exist(x)) return -1;

return ms.order\_of\_key(x);

}

int last(const T& x) const {

if (!exist(x)) return -1;

if ((\*this)[ms.size() - 1] == x) return ms.size() - 1;

return first(\*ms.lower\_bound(x)) - 1;

}

int lower(const T& x) const { // returns the index

if ((\*this)[ms.size() - 1] < x) return -1;

return ms.order\_of\_key(x);

}

int count(const T& x) const {

if (!exist(x)) return 0;

return last(x) - first(x) + 1;

}

int size() const { return ms.size(); }

void clear() { ms.clear(); }

};

## Monotonic queue

template<typename T> struct Monotonic\_queue { // FIFO & finding max or min

stack<pair<T, T>> s1, s2;

function<T(T, T)> merge;

static T Min(T a, T b) { return min(a, b); }

static T Max(T a, T b) { return max(a, b); }

Monotonic\_queue(function<T(T, T)> op) {

merge = op;

}

void push(const T& x) {

if (s1.empty()) s1.push({ x, x });

else s1.push({ x, merge(x, s1.top().second) });

}

void pop() {

if (s2.empty()) {

while (!s1.empty()) {

T e = s1.top().first, best = e;

s1.pop();

if (!s2.empty()) best = merge(best, s2.top().second);

s2.push({ e, best });

}

}

s2.pop();

}

T get() {

assert(!s1.empty() || !s2.empty());

if (s1.empty()) return s2.top().second;

if (s2.empty()) return s1.top().second;

return merge(s1.top().second, s2.top().second);

}

};

## 2D prefix sum

template<typename T = ll> struct pref\_2D {

int n, m;

vector<vector<T>> pref;

pref\_2D(const vector<vector<T>>& x) {

n = x.size();

m = x[0].size();

pref.resize(n, vector<T>(m, 0));

for (int j = 0; j < m; j++) {

pref[0][j] = x[0][j];

for (int i = 1; i < n; i++)

pref[i][j] = pref[i - 1][j] + x[i][j];

}

for (int i = 0; i < n; i++)

for (int j = 1; j < m; j++)

pref[i][j] += pref[i][j - 1];

}

ll get(int x, int y) {

if (x < 0 || y < 0) return 0;

return pref[x][y];

}

ll get(int i1, int j1, int i2, int j2) {

T ret = get(i2, j2) - get(i2, j1 - 1);

ret -= get(i1 - 1, j2) - get(i1 - 1, j1 - 1);

return ret;

}

};

## 2D partial sum

struct partial\_sum\_2D{

int n, m;

vector<vector<ll>> upd;

partial\_sum\_2D(int \_n, int \_m) {

n = n, m = \_m;

upd.resize(n, vector<ll>(m));

}

void update(int x1, int y1, int x2, int y2, ll val) {

upd[x1][y1] += val;

if (x2 + 1 < n) upd[x2 + 1][y1] -= val;

if (y2 + 1 < m) upd[x1][y2 + 1] -= val;

if (x2 + 1 < n and y2 + 1 < m) upd[x2 + 1][y2 + 1] += val;

}

void calc() {

for (int i = 1; i < n; i++)

for (int j = 0; j < m; j++) upd[i][j] += upd[i - 1][j];

for (int i = 0; i < n; i++)

for (int j = 1; j < m; j++) upd[i][j] += upd[i][j - 1];

}

};

# Math

## Big int functions

string addStrings(string num1, string num2) { // doesn’t handle negative numbers

string r = "";

int sz1 = num1.size(), sz2 = num2.size(), i = 0, j = 0, c = 0;

while (i < sz1 || j < sz2 || c > 0) {

int t = c;

t += i < sz1 ? num1[sz1 - i - 1] - '0' : 0;

t += j < sz2 ? num2[sz2 - j - 1] - '0' : 0;

r = char(t % 10 + '0') + r;

c = t / 10; // carry over next digit

i++;

j++;

}

return r;

}

string multiply(string num1, string num2) { // doesn’t handle negative numbers

int len1 = num1.size(), len2 = num2.size();

if (len1 == 0 || len2 == 0) return "0";

vector<int> result(len1 + len2, 0);

int i\_n1 = 0, i\_n2 = 0;

for (int i = len1 - 1; i >= 0; i--) {

int carry = 0;

int n1 = num1[i] - '0';

i\_n2 = 0;

for (int j = len2 - 1; j >= 0; j--) {

int n2 = num2[j] - '0';

int sum = n1 \* n2 + result[i\_n1 + i\_n2] + carry;

carry = sum / 10;

result[i\_n1 + i\_n2] = sum % 10;

i\_n2++;

}

if (carry > 0)

result[i\_n1 + i\_n2] += carry;

i\_n1++;

}

int i = result.size() - 1;

while (i >= 0 && result[i] == 0)

i--;

if (i == -1) return "0";

string s = "";

while (i >= 0)

s += std::to\_string(result[i--]);

return s;

}

## Rand

mt19937 rng(chrono::steady\_clock::now().time\_since\_epoch().count());

ll rand(ll l, ll r) {

return uniform\_int\_distribution<ll>(l, r)(rng);

}

## Matrix power

struct Matrix {

int n, m, mod;

vector<vector<int>> a;

Matrix() {}

Matrix(int n, int m, int mod, int val) : n(n), m(m), mod(mod), a(n, vector<int>(m, val)) {}

Matrix operator\*(const Matrix& b) {

Matrix res(n, b.m, mod, 0);

for (int i = 0; i < n; ++i)

for (int j = 0; j < b.m; ++j)

for (int k = 0; k < m; ++k)

res.a[i][j] = (res.a[i][j] + 1ll \* a[i][k] \* b.a[k][j]) % mod;

return res;

}

Matrix operator^(int e) {

Matrix res(n, n, mod, 0), b = \*this;

for (int i = 0; i < n; ++i)

res.a[i][i] = 1;

for (; e > 0; e /= 2, b = b \* b)

if (e % 2) res = res \* b;

return res;

}

void IdentifyD(int i, int j) {

while (i < n && j < m) {

a[i][j] = 1;

i++, j++;

}

}

};

## Calculate the determinant of matrix using gauss O(n3)

const double EPS = 1e-9;

#define matrix vector<vector<double>>

double det(matrix mat) { // O(n ^ 3)

int n = mat.size();

assert(n != 0 && mat[0].size() == n);

double res = 1;

for (int i = 0; i < n; i++) {

// Find the row with the maximum element in the current column

int maxRow = i;

for (int j = i + 1; j < n; j++)

if (abs(mat[j][i]) > abs(mat[maxRow][i])) maxRow = j;

// If the maximum element is near zero, the determinant is zero

if (abs(mat[maxRow][i]) < EPS) return 0;

// Swap the current row with the row of the maximum element

swap(mat[i], mat[maxRow]);

if (i != maxRow) res = -res;

// Multiply the result by the pivot element

res \*= mat[i][i];

// Normalize the pivot row

for (int j = i + 1; j < n; j++) mat[i][j] /= mat[i][i];

// Eliminate the current column in the other rows

for (int j = 0; j < n; j++)

if (j != i && abs(mat[j][i]) > EPS)

for (int k = i + 1; k < n; k++)

mat[j][k] -= mat[i][k] \* mat[j][i];

}

return res;

}

## Find matrix inverse O(n3)

/\*check if a matrix has an inverse before trying to calculate it.

you need to ensure that the matrix is square and that its determinant is non-zero.\*/

matrix inverse(matrix mat) { // O(n ^ 3)

int n = mat.size();

assert(n != 0 && mat[0].size() == n);

// Create an augmented matrix with the identity matrix on the right

matrix augmented(n, vector<double>(2 \* n));

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++)

augmented[i][j] = mat[i][j];

augmented[i][n + i] = 1.0;

}

// Perform Gaussian elimination to transform the matrix into row echelon form

for (int i = 0; i < n; i++) {

// Pivoting to avoid division by zero

int maxRow = i;

for (int k = i + 1; k < n; k++)

if (abs(augmented[k][i]) > abs(augmented[maxRow][i])) maxRow = k;

swap(augmented[i], augmented[maxRow]);

// Make the diagonal contain all 1s

double diagVal = augmented[i][i];

for (int k = 0; k < 2 \* n; k++) augmented[i][k] /= diagVal;

// Eliminate the other rows

for (int j = 0; j < n; j++) {

if (i != j) {

double factor = augmented[j][i];

for (int k = 0; k < 2 \* n; k++)

augmented[j][k] -= factor \* augmented[i][k];

}

}

}

// Extract the right half of the augmented matrix, which is now the inverse

matrix inv(n, vector<double>(n));

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++) inv[i][j] = augmented[i][n + j];

return inv;

}

## MillerRabin (Check if a number is prime) O(log n)

using u128 = \_\_uint128\_t;

/// return (s \* m) % mod

ll mult(ll s, ll m, ll mod) {

if (!m) return 0;

ll ret = mult(s, m / 2, mod);

ret = (ret + ret) % mod;

if (m & 1) ret = (ret + s) % mod;

return ret;

}

ll fp(ll b, ll p, ll mod) {

ll res = 1;

b %= mod;

while (p > 0) {

if (p & 1) res = (u128)res \* b % mod;

b = (u128)b \* b % mod;

p >>= 1;

}

return res;

}

bool check(ll n, ll a, ll d, ll s) {

ll x = fp(a, d, n);

if (x == 1 || x == n - 1) return 0;

for (int r = 1; r < s; r++) {

x = (u128)x \* x % n;

if (x == n - 1) return 0;

}

return 1;

}

bool MillerRabin(ll n) {

if (n < 2) return 0;

int r = 0;

ll d = n - 1;

while ((d & 1) == 0) {

d >>= 1;

r++;

}

for (ll a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {

if (n == a) return 1;

if (check(n, a, d, r)) return 0;

}

return 1;

}

## Phi

ll phi(ll n) {

ll res = n;

for (ll 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;

}

## Phi (preprocessing)

const int N = 1e5 + 5;

int phi[N];

for (int i = 0; i < N; i++) 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;

## Sieve

const int N = 1e6 + 6;

bool isPrime[N];

vector<int> primes;

void sieve() {

memset(isPrime, 1, sizeof isPrime);

isPrime[0] = isPrime[1] = 0;

for (int i = 2; i \* i < N; i++) {

if (isPrime[i])

for (int j = i \* i; j < N; j += i) isPrime[j] = false;

}

for (int i = 2; i < N; i++)

if (isPrime[i]) primes.push\_back(i);

}

## SPF

vector<int> SPF(int N) {

vector<int> spf(N + 1, 0);

for (int i = 0; i <= N; i += 2) spf[i] = 2;

for (int i = 3; i \* i <= N; i += 2) {

if (spf[i]) continue;

for (int j = i \* i; j <= N; j += i)

if (spf[j] == 0) spf[j] = i;

}

for (int i = 1; i <= N; i++)

if (spf[i] == 0) spf[i] = i;

return spf;

}

## Divisors (n)

vector<ll> get\_divisors(ll x) {

vector<ll> d = { 1 };

for (ll i = 2; i \* i <= x; i++) {

if (x % i == 0) {

d.push\_back(i);

if (x / i != i) d.push\_back(x / i);

}

}

if (x > 1) d.push\_back(x);

return d;

}

## Divisors [2, n]

vector<vector<int>> Get\_Divisors(int N) {

vector<vector<int>> d(N + 1);

for (int i = 2; i <= N; ++i)

for (int j = i; j <= N; j += i)

d[j].push\_back(i);

return d;

}

## Prime factors (n)

vector<pair<ll, int>> prime\_factors(ll x) {

vector<pair<ll, int>> fac;

int cnt = 0;

for (ll i = 2; i \* i <= x; i++) {

cnt = 0;

while (x % i == 0) cnt++, x /= i;

if (cnt) fac.push\_back({i, cnt});

}

if (x > 1) fac.push\_back({x, 1});

return fac;

}

## Sum [l, r]

ll sum(ll l, ll r) {

assert(l <= r);

return (r \* (r + 1) >> 1) - (l \* (l - 1) >> 1);

}

ll sumOdd(ll l, ll r) {

assert(l <= r);

r++;

ll x = r >> 1;

ll y = l >> 1;

return x \* x - y \* y;

}

ll sumEven(ll l, ll r) {

return sum(l, r) - sumOdd(l, r);

}

**extended euclidean algorithm**

// a \* x + b \* y = gcd(a , b)

ll extended\_euclid(ll a, ll b, ll& x, ll& y) {

if (b == 0) {

x = 1, y = 0;

return a;

}

ll x1, y1;

ll g = extended\_euclid(b, a % b, x1, y1);

x = y1, y = x1 - (a / b) \* y1;

return g;

}

## Fast Power (log p)

int mul(int a, int b, int m = Mod) {

return 1ll \* a \* b % m;

}

int Pow(int a, ll p, int m = Mod) {

int res = 1;

while (p) {

if (p & 1) res = mul(res, a, m);

a = mul(a, a, m);

p >>= 1;

}

return res;

}

## Factorial & nCr & nPr (preprocessing)

vector<int> F, invF;

void pre(int n) {

F.resize(n + 1);

invF.resize(n + 1);

F[0] = 1;

for (int i = 1; i <= n; i++)

F[i] = mul(F[i - 1], i, Mod);

invF[n] = inv(F[n]);

invF[0] = 1;

for (int i = n - 1; i; --i)

invF[i] = mul(invF[i + 1], i + 1);

}

int nCr(int n, int r) {

if (n < r) return 0;

return mul(F[n], mul(invF[r], invF[n - r]));

}

int nPr(int n, int r) {

if (n < r) return 0;

return mul(F[n], invF[n - r]);

}

## nCr & nPr

ll nCr(ll n, ll r) {

if (r > n) return 0;

ll p = 1, k = 1;

r = min(r, n - r);

while (r) {

p \*= n, k \*= r;

ll m = \_\_gcd(p, k);

p /= m, k /= m, n--, r--;

}

return p;

}

ll nPr(ll n, ll r) {

if (r > n) return 0;

ll ret = 1;

while (r) ret \*= n--, r--;

return ret;

}

## Fibonacci O(log n)

int Fib(int n) { // 0, 1, 1, 2, 3, 5 | call it with(n+1) to start with 1

ll i = 1, h = 1, j = 0, k = 0, t;

while (n > 0) {

if (n % 2 == 1) {

t = (j \* h) % MOD;

j = (i \* h + j \* k + t) % MOD;

i = (i \* k + t) % MOD;

}

t = (h \* h) % MOD;

h = (2 \* k \* h + t) % MOD;

k = (k \* k + t) % MOD;

n = n / 2;

}

return j;

}

## Divisabilty rules

2 The last digit should be even.

3 The sum of the digits should be divisible by 3.

4 The last two digits should be divisible by 4.

5 The last digit should either be 0 or 5.

6 The number should be divisible by both 2 and 3.

7 The double of the last digit, when subtracted by the rest of the number, the difference obtained should be divisible by 7.

8 The last three digits should be divisible by 8.

9 The sum of the digits should be divisible by 9.

10 The last digit should be 0.

11 The difference of the alternating sum of digits should be divisible by 12.

12 The number should be divisible by both 3 and 4.

13 The four times of the last digit, when added to the rest of the number, the result obtained should be divisible by 13.

17 The five times of the last digit, when subtracted by the rest of the number, the difference obtained should be divisible by 17.

19 The double of the last digit, when added to the rest of the number, the result obtained should be divisible by 19.

## Catalan number

A math equation with numbers and symbols

Description automatically generated 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …

The Catalan number Cn is the solution for:

**1. Balanced Parentheses:** The number of correct bracket sequence consisting of `**n`** opening and `**n**` closing brackets.

**2. Word Problems**: Counting the number of valid sequences of balanced words in formal language theory.

**3. Bracketing Problems:** The number of ways to fully parenthesize an expression with `**n**` pairs of operands and operators.

**4. Binary Search Trees:** The number of distinct binary search trees that can be formed with `**n**` nodes.

**5. Sorted Trees:** The number of different ways to insert `**n**` elements into an empty binary search tree.

**6. Full Binary Trees:** The number of full binary trees with `**n+1**` leaves.

**7. Triangulations of a Polygon:** The number of ways to triangulate a covex polygon with `**n+2**` sides. (i.e. the number of partitions of polygon into disjoint triangles by using the diagonals).

**8. Non-Crossing Handshakes:** The number of ways `**n**` pairs of people can shake hands without any of the handshakes crossing.

**9. Dyck Paths:** The number of paths along the edges of a grid that do not pass above the diagonal.

**10. Stack Permutations:** The number of valid permutations of a sequence of `**1**` to `**n**` where each prefix is a valid stack permutation.

**11. Valid Sequences:** The number of valid sequences of operations that can be performed on a stack to produce a given permutation of elements.

**12. Mountain Ranges:** The number of ways to draw mountain ranges with `**n**` upstrokes and `**n**` downstrokes that never go below the starting point.

**13. Schroeder's Second Problem:** The number of ways to insert `**n**` pairs of parentheses into a sequence such that the resulting expression is correctly matched.

**14. Planar Graphs:** Counting certain types of planar graphs, such as non-crossing partitions of a set of points in the plane.

**15. Matrix Chain Multiplication:** The number of ways to fully parenthesize a product of `**n+1**` matrices.

**16. Catalan Paths:** The number of paths along the edges of a grid that start at the origin and do not pass above the line `y = x`.

# Trees

## LCA (Binary lifting)

struct LCA {

vector <vector<int>> adj;

vector<int> dis;

int LG;

vector <vector<int>> up;

LCA(int n) {

n += 5;

adj.resize(n);

LG = 1;

while ((1 << LG) < n) LG++;

dis.resize(n);

up.resize(n, vector<int>(LG));

}

LCA(const vector <vector<int>>& \_adj) {

LCA(\_adj.size());

adj = \_adj;

}

void addEdge(int u, int v) {

adj[u].push\_back(v);

adj[v].push\_back(u);

}

void build(int node = 1, int p = 0) {

dis[node] = dis[p] + 1;

up[node][0] = p;

for (int i = 1; i < LG; i++) {

up[node][i] = up[up[node][i - 1]][i - 1];

}

for (int i : adj[node])

if (i != p) build(i, node);

}

int lca(int u, int v) {

if (dis[u] < dis[v]) swap(u, v);

int diff = dis[u] - dis[v];

for (int i = 0; i < LG; i++)

if ((1 << i) & diff) u = up[u][i];

if (u == v) return u;

for (int i = LG - 1; i >= 0; i--) {

if (up[u][i] != up[v][i])

u = up[u][i], v = up[v][i];

}

return up[u][0];

}

int kth\_anc(int u, int k) {

for (int i = LG - 1; i >= 0; i--)

if ((1 << i) & k) u = up[u][i];

return !u ? -1 : u;

}

};

## LCA with segment tree (Euler tour)

struct LCA {

vector<int> height, euler, first, segtree, tree;

vector<bool> visited;

int n;

LCA(vector<vector<int>>& adj, int root = 0) {

n = adj.size();

height.resize(n);

first.resize(n);

euler.reserve(n \* 2);

visited.assign(n, false);

dfs(adj, root);

int m = euler.size();

segtree.resize(m \* 4);

build(1, 0, m - 1);

}

void dfs(vector<vector<int>>& adj, int node, int h = 0) {

visited[node] = true;

height[node] = h;

first[node] = euler.size();

euler.push\_back(node);

for (auto to : adj[node]) {

if (!visited[to]) {

dfs(adj, to, h + 1);

euler.push\_back(node);

}

}

}

void build(int node, int b, int e) {

if (b == e) {

segtree[node] = euler[b];

}

else {

int mid = (b + e) / 2;

build(node << 1, b, mid);

build(node << 1 | 1, mid + 1, e);

int l = segtree[node << 1], r = segtree[node << 1 | 1];

segtree[node] = (height[l] < height[r]) ? l : r;

}

}

int query(int node, int b, int e, int L, int R) {

if (b > R || e < L)

return -1;

if (b >= L && e <= R)

return segtree[node];

int mid = (b + e) >> 1;

int left = query(node << 1, b, mid, L, R);

int right = query(node << 1 | 1, mid + 1, e, L, R);

if (left == -1) return right;

if (right == -1) return left;

return height[left] < height[right] ? left : right;

}

int lca(int u, int v) {

int left = first[u], right = first[v];

if (left > right)

swap(left, right);

return query(1, 0, euler.size() - 1, left, right);

}

};

## HLD

const int N = 200000 + 9;

const int SEG\_MAX = 4 \* N + 9; //L = Ceil(log2(V)). TREE\_SIZE = 2 + (1<<(L+1))

int interval[SEG\_MAX];

int mxV; // current # of tree nodes

struct SegmentTree { // Range Max Query

int idx, val, from, to;

void init(int n) {

mxV = n;

memset(interval, 0, n \* sizeof(int));

}

// Initialize idx & val before update

int update(int s = 0, int e = mxV, int p = 1) {

if (s == e)

return interval[p] = val;

int mid = (s + e) / 2;

int left = (p << 1), right = left + 1;

if (idx <= mid)

update(s, mid, left);

else if (idx > mid)

update(mid + 1, e, right);

return interval[p] = max(interval[left], interval[right]);

}

// Initialize from & to before query

int query(int s = 0, int e = mxV, int p = 1) {

if (from <= s && to >= e)

return interval[p];

int mid = (s + e) / 2;

int left = (p << 1), right = left + 1;

if (to <= mid)

return query(s, mid, left);

if (from > mid)

return query(mid + 1, e, right);

int a = query(s, mid, left);

int b = query(mid + 1, e, right);

return max(a, b);

}

};

/////////////////////////////////////////////////////////////////////////////////////

const int isValueOnEdge = 1; // switch to 0 for value on node

vector<vector<int>> tree; // undirected tree

// For values on edge

vector<vector<int>> treeEdgeIdx;

vector<int> edge\_to; // which end point used in directing the edge

vector<int> edge\_cost;

struct HeavyLight {

int parent[N], depth[N], heavy[N], root[N], segTreePos[N];

int queryRes; // e.g. max value on path

SegmentTree segTree;

int dfs\_hld(int v) {

int size = 1, maxSubtree = 0;

for (int k = 0; k < (int)tree[v].size(); ++k) {

int u = tree[v][k], edgeIdx = treeEdgeIdx[v][k];

if (u != parent[v]) {

edge\_to[edgeIdx] = u;

parent[u] = v, depth[u] = depth[v] + 1;

int childTreeSize = dfs\_hld(u);

if (childTreeSize > maxSubtree)

heavy[v] = u, maxSubtree = childTreeSize;

size += childTreeSize;

}

}

return size;

}

void buildChains() {

int n = tree.size();

memset(heavy, -1, n \* sizeof(int));

parent[0] = -1, depth[0] = 0;

dfs\_hld(0);

// Connect chains to its root. Map chain to segment tree part

for (int chainRoot = 0, pos = 0; chainRoot < n; ++chainRoot) {

if (parent[chainRoot] == -1 || heavy[parent[chainRoot]] != chainRoot) {

for (int j = chainRoot; j != -1; j = heavy[j]) // iterate on a chain

root[j] = chainRoot, segTreePos[j] = pos++;

}

}

segTree.init(n);

}

void queryChain(int l, int r) {

segTree.from = l, segTree.to = r;

queryRes = max(queryRes, segTree.query());

}

int queryPath(int u, int v) {

queryRes = 0; // be careful from u = v for isValueOnEdge

for (; root[u] != root[v]; v = parent[root[v]]) {

if (depth[root[u]] > depth[root[v]])

swap(u, v);

queryChain(segTreePos[root[v]], segTreePos[v]);

}

if (depth[u] > depth[v])

swap(u, v);

if (!isValueOnEdge || u != v)

queryChain(segTreePos[u] + isValueOnEdge, segTreePos[v]);

return queryRes; // u = LCA node

}

// For value on DIRECTED edge (f, t, value), call update\_node(t, value)

void updatePos(int treeNode, int value) {

segTree.idx = segTreePos[treeNode], segTree.val = value;

segTree.update();

}

};

## Centroid decomposition

struct Centroid {

vector<int> sz, par;

vector<bool> vis;

vector<vector<int>> adj, tree;

int root;

Centroid(int n) {

n += 5;

sz.resize(n, 0);

par.resize(n, -1);

tree.resize(n);

vis.resize(n, false);

adj.resize(n);

}

void addEdge(int u, int v) {

adj[u].push\_back(v);

adj[v].push\_back(u);

}

int build(int u = 1) {

dfs(u, -1);

int ctr = find\_centroid(u, -1, sz[u]);

vis[ctr] = true;

for (auto v : adj[ctr]) {

if (!vis[v]) {

int ch = build(v);

par[ch] = ctr;

tree[ctr].push\_back(ch);

}

}

return ctr;

}

void dfs(int u, int p) {

sz[u] = 1;

for (auto v : adj[u]) {

if (v != p && !vis[v]) {

dfs(v, u);

sz[u] += sz[v];

}

}

}

int find\_centroid(int u, int p, int n) {

for (auto v : adj[u]) {

if (v != p && !vis[v] && 2 \* sz[v] > n)

return find\_centroid(v, u, n);

}

return u;

}

};

## Small to large

const int N = 1e6 + 5;

struct Node {

int ans = 0;

vector<int>Child;

map<int, int>fr; // dpth, cnt

set<pair<int, int>>st; // mx,idx

}tree[N];

void merge(int& a, int b) {

if (tree[a].fr.size() < tree[b].fr.size())

swap(a, b);

for (auto& [hi, cnt] : tree[b].fr) {

auto c = tree[a].fr[hi];

auto it = tree[a].st.find({ c,-hi });

if (it != tree[a].st.end())tree[a].st.erase(it);

c += cnt;

tree[a].fr[hi] = c;

tree[a].st.insert({ c,-hi });

}

}

void process(int par, int node, int dpth) {

auto it = tree[node].st.end();

it--;

int hi = it->second;

tree[par].ans = -hi - dpth;

}

void dfs(int& par, int root, int dpth = 0) {

// pre

tree[par].fr[dpth]++;

tree[par].st.insert({ 1,-dpth });

int node = par;

// go

for (auto to : tree[par].Child) {

if (to == root) continue;

dfs(to, par, dpth + 1);

merge(node, to);

}

// compute ans

process(par, node, dpth);

par = node;

}

# Graphs

## Floyd-Warshall Algorithm O(n3)

for (int k = 1; k <= n; k++)

for (int i = 1; i <= n; i++)

for (int j = 1; j <= n; j++)

if (cost[i][k] < INF && cost[k][j] < INF)

cost[i][j] = min(cost[i][j], cost[i][k] + cost[k][j]);

## Bellman Ford O(EV)

class Bellman {

public:

struct Edge {

ll x, y, c;

};

ll n;

const ll INF = 1e15;

vector<Edge>edges;

vector<ll>dst;

vector<vector<ll>>v;

set<ll>path;

Bellman(int nn) {

this->n = nn;

dst.resize(n + 1);

v.resize(n + 1);

}

bool bellman(int src = 1) {

for (int i = 0; i < n - 1; i++)

for (auto& e : edges)

if (dst[e.x] + e.c < dst[e.y])

dst[e.y] = dst[e.x] + e.c;

vector<ll> on;

for (auto& e : edges)

if (dst[e.x] + e.c < dst[e.y])

on.push\_back(e.x);

if (on.empty()) return 1; // no negative cycles

vector<bool>vis(n + 1);

for (auto& i : on)

if (!vis[i]) dfs(i, vis);

for (int i = 1; i <= n; i++)

if (vis[i]) path.insert(i);

// path: containing any vertex appear in a negative cycle

return 0;

}

void dfs(int par, vector<bool>& vis) {

vis[par] = 1;

for (auto& i : v[par])if (!vis[i])dfs(i, vis);

}

};

## SPFA

const int INF = 1000000000;

vector<vector<pair<int, int>>> adj;

bool spfa(int s, vector<int>& d) {

int n = adj.size();

d.assign(n, INF);

vector<int> cnt(n, 0);

vector<bool> inqueue(n, false);

queue<int> q;

d[s] = 0;

q.push(s);

inqueue[s] = true;

while (!q.empty()) {

int v = q.front();

q.pop();

inqueue[v] = false;

for (auto edge : adj[v]) {

int to = edge.first;

int len = edge.second;

if (d[v] + len < d[to]) {

d[to] = d[v] + len;

if (!inqueue[to]) {

q.push(to);

inqueue[to] = true;

cnt[to]++;

if (cnt[to] > n)

return false; // negative cycle

}

}

}

}

return true;

}

## SCC using Kosaraju's Algorithm

struct SCC {

int n;

vector<bool> vis;

vector<vector<int>> adj[2], com;

stack<int> stk;

SCC(int \_n) {

n = \_n;

adj[0].resize(n + 1);

adj[1].resize(n + 1);

}

void addEdge(int u, int v) {

adj[0][u].push\_back(v);

adj[1][v].push\_back(u);

}

void dfs(int u) {

vis[u] = true;

for (auto v : adj[0][u]) if (!vis[v]) dfs(v);

stk.push(u);

}

void dfs(int u, vector<int>& group) {

vis[u] = true;

group.push\_back(u);

for (auto v : adj[1][u]) if (!vis[v]) dfs(v, group);

}

void build() {

vis.assign(n + 1, false);

for (int i = 1; i <= n; i++) if (!vis[i]) dfs(i);

fill(vis.begin(), vis.end(), false);

while (stk.size()) {

int u = stk.top();

stk.pop();

if (!vis[u]) {

com.emplace\_back();

dfs(u, com.back());

}

}

}

};

## 2-sat

struct two\_sat {

int n, id, t;

vector<vector<int>> adj;

vector<int> scc\_id, low, time;

stack<int> stk;

vector<bool> is;

two\_sat(int \_n) {

n = \_n, id = 0, t = 0;

adj.resize(2 \* n);

is.resize(n);

}

int Not(int i) {

return i < n ? i + n : i - n;

}

void addEdge(int a, int b) {

adj[a].push\_back(b);

}

void set\_true(int i) {

addEdge(Not(i), i);

}

void either(int a, int b) {

addEdge(Not(a), b);

addEdge(Not(b), a);

}

void make\_equal(int a, int b) {

addEdge(a, b), addEdge(Not(a), Not(b)),

addEdge(b, a), addEdge(Not(b), Not(a));

}

void dfs(int u) {

time[u] = low[u] = t++, stk.push(u);

for (auto& v : adj[u]) if (scc\_id[v] == -1) {

if (time[v] == -1) dfs(v);

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

}

if (low[u] == time[u]) {

int cur = id++, v;

do {

v = stk.top(), stk.pop(), scc\_id[v] = cur;

} while (v != u);

}

}

bool solve() {

scc\_id = time = low = vector<int>(2 \* n, -1);

for (int i = 0; i < 2 \* n; i++) {

if (time[i] == -1) dfs(i);

}

for (int i = 0; i < n; i++) {

if (scc\_id[i] == scc\_id[Not(i)]) return false;

is[i] = scc\_id[i] < scc\_id[Not(i)];

}

return true;

}

};

## Articulation points

vector<vector<int>> adj;

vector<int> low, disc, ap;

int Time;

int dfsAP(int u, int p) {

int children = 0;

low[u] = disc[u] = ++Time;

for (int& v : adj[u]) {

if (v == p) continue; // we don't want to go back through the same path.

// if we go back is because we found another way back

if (!disc[v]) { // if V has not been discovered before

children++;

dfsAP(v, u); // recursive DFS call

if (disc[u] <= low[v]) // condition #1

ap[u] = 1;

low[u] = min(low[u], low[v]); // low[v] might be an ancestor of u

}

else // if v was already discovered means that we found an ancestor

low[u] = min(low[u], disc[v]); // finds the ancestor with the least discovery time

}

return children;

}

void AP() {

ap = low = disc = vector<int>(adj.size());

Time = 0;

for (int u = 0; u < adj.size(); u++)

if (!disc[u])

ap[u] = dfsAP(u, u) > 1; // condition #2

}

## Bridges

vector<vector<int>> adj;

vector<int> low, disc, ap;

int Time;

vector<pair<int, int>> br;

int dfsBR(int u, int p) {

low[u] = disc[u] = ++Time;

for (int& v : adj[u]) {

if (v == p) continue; // we don't want to go back through the same path.

// if we go back is because we found another way back

if (!disc[v]) { // if V has not been discovered before

dfsBR(v, u); // recursive DFS call

if (disc[u] < low[v]) // condition to find a bridge

br.push\_back({ u, v });

low[u] = min(low[u], low[v]); // low[v] might be an ancestor of u

}

else // if v was already discovered means that we found an ancestor

low[u] = min(low[u], disc[v]); // finds the ancestor with the least discovery time

}

}

void BR() {

low = disc = vector<int>(adj.size());

Time = 0;

for (int u = 0; u < adj.size(); u++)

if (!disc[u]) dfsBR(u, u);

}

## Minimum spanning arborescence (MSA) - Zhu-Liu algorithm O(EV)

/\* it takes a directed graph as input, where each edge has a weight associated with it.

The goal is to find a subset of edges that form a spanning arborescence

(a rooted tree that spans all vertices) with the minimum total weight,

starting from a designated root node.

\*/

vector<array<int, 3>> edges;

ll ZhuLiu(int n, int m, int root) {

ll ans = 0;

vector<int> in(n + 1), id(n + 1), vis(n + 1), par(n + 1);

const int INF = 1e8;

while (true) {

for (int i = 1; i <= n; i++) {

in[i] = INF, id[i] = vis[i] = -1;

}

for (int i = 0; i < m; i++) {

int u = edges[i][1], v = edges[i][2], c = edges[i][0];

if (u != v && c < in[v]) {

in[v] = c, par[v] = u;

}

}

for (int i = 1; i <= n; i++) {

if (i != root && in[i] == INF) return -1;

}

in[root] = 0;

int x = 0;

for (int i = 1; i <= n; i++) {

ans += in[i];

int v = i;

while (v != root && id[v] == -1 && vis[v] != i)

vis[v] = i, v = par[v];

if (v != root && id[v] == -1) {

id[v] = ++x;

for (int u = par[v]; u != v; u = par[u]) id[u] = x;

}

}

if (x == 0) break;

for (int i = 1; i <= n; i++) {

if (id[i] == -1) id[i] = ++x;

}

int i = 0;

while (i < m) {

int u = edges[i][1], v = edges[i][2];

edges[i][1] = id[u], edges[i][2] = id[v];

if (u != v) edges[i++][0] -= in[v];

else swap(edges[i], edges[--m]);

}

n = x;

root = id[root];

}

return ans;

}

## Ford-Fulkerson Max-flow O(E \* F)

const int INF = 2e9;

struct Ford\_Fulkerson {

struct edge {

int to, rev;

int cap;

bool d;

edge() {}

edge(int \_to, int \_rev, int \_cap, bool \_d) {

to = \_to, rev = \_rev, cap = \_cap, d = \_d;

}

};

int n;

vector<vector<edge>> adj;

Ford\_Fulkerson(int \_n) {

n = \_n, adj.resize(n);

}

void addEdge(int from, int to, int cap) {

int id1 = adj[from].size();

int id2 = adj[to].size();

adj[from].push\_back(edge(to, id2, cap, 1));

adj[to].push\_back(edge(from, id1, 0, 0));

}

int maxFlow(int s, int t) {

int flow = 0;

while (true) {

vector<int> m(n, INF);

vector<int> pv(n, -1), pe(n, -1);

vector<bool> used(n, false);

queue<int> q;

q.push(s);

used[s] = true;

while (!q.empty()) {

int u = q.front();

q.pop();

int cnt = adj[u].size();

for (int i = 0; i < cnt; i++) {

int v = adj[u][i].to;

int cap = adj[u][i].cap;

if (!used[v] && cap > 0) {

used[v] = true;

m[v] = min(m[u], cap);

pv[v] = u, pe[v] = i;

q.push(v);

}

}

}

if (!used[t]) break;

int f = m[t];

for (int i = t; i != s; i = pv[i]) {

adj[pv[i]][pe[i]].cap -= f;

adj[i][adj[pv[i]][pe[i]].rev].cap += f;

}

flow += f;

}

return flow;

}

};

## Dinic Max-flow (um\_nik) O(V2 E)

struct Dinic {

struct Edge {

int from, to;

ll cap;

Edge() {}

Edge(int \_from, int \_to, ll \_cap) {

from = \_from, to = \_to, cap = \_cap;

}

};

const ll INF = 1e18;

vector<Edge> edges;

vector<vector<int>> adj;

int E;

int n, s, t;

ll maxFlow;

vector<int> dis, idx;

Dinic(int \_n) {

n = \_n, E = 0, maxFlow = 0;

adj.resize(n);

}

void addEdge(int v, int to, ll cap) {

adj[v].push\_back(E++);

edges.emplace\_back(Edge(v, to, cap));

adj[to].push\_back(E++);

edges.emplace\_back(Edge(to, v, 0));

}

bool bfs() {

dis.resize(n);

fill(dis.begin(), dis.end(), -1);

queue<int> q;

dis[s] = 0, q.push(s);

while (!q.empty()) {

int u = q.front();

q.pop();

for (int id : adj[u]) {

Edge e = edges[id];

int v = e.to;

if (e.cap > 0 && dis[v] == -1)

dis[v] = dis[u] + 1, q.push(v);

}

}

return ~dis[t];

}

ll dfs(int u, ll flow) {

if (u == t || flow == 0) return flow;

ll res = 0;

for (int& i = idx[u]; i < adj[u].size(); i++) {

int id = adj[u][i];

Edge e = edges[id];

int to = e.to;

if (dis[to] != dis[u] + 1) continue;

ll df = dfs(to, min(flow, e.cap));

flow -= df, res += df;

edges[id].cap -= df, edges[id ^ 1].cap += df;

if (flow == 0) return res;

}

return res;

}

ll Flow(int \_s, int \_t) {

s = \_s, t = \_t;

idx.resize(n);

maxFlow = 0;

while (bfs()) {

fill(idx.begin(), idx.end(), 0);

maxFlow += dfs(s, INF);

}

return maxFlow;

}

};

## Max-flow min-cost

const int N = 500;

int n, src, sink;

int flow[N][N];

int cost[N][N];

pair<int, int> belleman\_ford()

{

vector<int> d(n, 1e9);

d[src] = 0;

vector<int> par(n);

while (1)

{

bool any = 0;

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (i == j or !flow[i][j])

continue;

if (d[j] > d[i] + cost[i][j])

{

d[j] = d[i] + cost[i][j];

par[j] = i;

any = 1;

}

}

}

if (!any)

break;

}

if (d[sink] == 1e9)

return { 0, 0 };

int node = sink;

int mn = 1e9;

while (node)

{

int p = par[node];

mn = min(mn, flow[p][node]);

node = p;

}

int ret = 0;

node = sink;

while (node)

{

int p = par[node];

flow[p][node] -= mn;

flow[node][p] += mn;

ret += mn \* cost[p][node];

node = p;

}

return { mn, ret };

}

pair<int, int> fl()

{

int curfl = 0;

int cost = 0;

while (true)

{

auto f = belleman\_ford();

if (!f.first)

break;

curfl += f.first;

cost += f.second;

}

return {curfl, cost};

}

MATH HELPER:  
class MathHelper {

public:

vector<int>fact, inv, power;

const int MOD = N;

int n;

MathHelper(int n) {

fact.resize(n + 5);

inv.resize(n + 5);

this->n = n;

power.resize(n + 5);

pre();

}

void pre() {

fact[0] = power[0] = 1;

inv[0] = Inv(fact[0]);

int pw = 1;

for (int i = 1; i <= n; i++) {

fact[i] = mul(i, fact[i - 1]);

inv[i] = Inv(fact[i]);

pw = mul(pw, 10);

power[i] = add(power[i - 1], pw);

}

}

int add(int a, int b) {

a += MOD;

b += MOD;

return ((a % MOD) + (b % MOD)) % MOD;

}

int mul(int a, int b) {

return ((a % MOD) \* (b % MOD)) % MOD;

}

int Exp(int base, int pw) {

int ret = 1;

while (pw > 0) {

if (pw & 1LL)ret = mul(ret, base);

base = mul(base, base);

pw /= 2;

}

return ret;

}

int gcd(int a, int b) { if (b == 0) return a; return gcd(b, a % b); }

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

int Inv(int n) {

//assert(gcd(n, MOD) == 1);

return Exp(n, MOD - 2);

}

int nCr(int n, int r) {

// !n / !r\*!(n-r)

if (n < r)return 0;

return mul(fact[n], mul(inv[r], inv[n - r]));

}

int nPr(int n, int r) {

return mul(fact[n], inv[n - r]);

}

};

N(n+1)(2N+1)/6

Area of the Intersection of Two Circles :

No overlap: d≥r1​+r2​ → area = 0

One circle entirely inside the other: d≤∣r1​−r2​∣ → area = area of the smaller circle

else:

r12​cos−1(​d2+r12​−r22​​ / 2d\*r1) +

r22​cos−1(​d2+r22​−r12​​ / 2d\*r2) - 1/2\*sqrt​((−d+r1​+r2​)(d+r1​−r2​)(d−r1​+r2​)(d+r1​+r2​))​

PI = 3.14159265359

cos-1 in C++ → acosl();