ACM-ICPC Team Reference

2014 South America/Brazil Regional

Guns N' Codes

Universidade Federal do Rio de Janeiro

Summary

[Configuration files 2](#_Toc398310257)

[.vimrc 2](#_Toc398310258)

[Template 2](#_Toc398310259)

[Useful facts 2](#_Toc398310260)

[Graph 3](#_Toc398310261)

[Tarjan 3](#_Toc398310262)

[Articulation 3](#_Toc398310263)

[Bridge 4](#_Toc398310264)

[Hopcroft-Karp 4](#_Toc398310265)

[Dinic 5](#_Toc398310266)

[Lowest Common Ancestor 5](#_Toc398310267)

[Min-Cost Max-Flow 6](#_Toc398310268)

[Heavy-Light Decomposition 7](#_Toc398310269)

[String 9](#_Toc398310270)

[KMP 9](#_Toc398310271)

[String Hashing 9](#_Toc398310272)

[Manacher’s Algorithm 9](#_Toc398310273)

[Aho-Corasick 10](#_Toc398310274)

[Suffix Array and Longest Common Prefix 11](#_Toc398310275)

[Dynamic Programming 12](#_Toc398310276)

[Optimal Binary Search Tree 12](#_Toc398310277)

[Longest Increasing Subsequence 12](#_Toc398310278)

[Longest Common Increasing Subsequence 13](#_Toc398310279)

[Weighted Activity Selection 13](#_Toc398310280)

[Data Structure 14](#_Toc398310281)

[Segment Tree with Lazy Propagation 14](#_Toc398310282)

[Geometry 14](#_Toc398310283)

[Template 14](#_Toc398310284)

[Monotone Chain Convex Hull 16](#_Toc398310285)

[Smallest Enclosing Circle 16](#_Toc398310286)

[Closest Pair of Points 16](#_Toc398310287)

[Math 17](#_Toc398310288)

[Sieve, primality, factorization, phi 17](#_Toc398310289)

[Chinese Remainder Algorithm 18](#_Toc398310290)

[FFT 18](#_Toc398310291)

[Polynomial 19](#_Toc398310292)

[Bignum 20](#_Toc398310293)

# Configuration files

## .vimrc

set nocp

set mouse=a

filetype plugin indent on

syntax on

set smd ls=2 nu sm is nohls bg=dark

set et sw=4 sts=4 ts=8 sta ai ci

set nowb nobk noswf

## Template

#include <bits/stdc++.h>

using namespace std;

#define INF 0x3f3f3f3f

#define MOD 1000000007

#define PI M\_PI

#define mset(a, x) memset(a, x, sizeof(a))

#define pb push\_back

#define mp make\_pair

#define fi first

#define se second

typedef long long ll;

typedef unsigned long long ull;

typedef pair<int, int> pii;

const double inf = 1.0/0.0;

int cmp\_double(double a, double b, double eps = 1e-9) {

return a + eps > b ? b + eps > a ? 0 : 1 : -1;

}

int main() {

}

# Useful facts

**Erdös-Gallai theorem**: A sequence of non-negative integers can be represented as the degree sequence of a finite simple graph on vertices if and only if is even and holds for .

**Split graph property**: A split graph can be recognized solely from their degree sequence. Let the degree sequence of a graph G be and is the largest value of such that . Then G if a split graph if and only if .

**Stirling’s approximation**:

**2-SAT**: Algorithm for solving Boolean expression in 2-CNF form (example: ).

1. Transform each term of conjunctions into
2. Construct graph such that each literal is a vertex and each implication is an edge
3. Run SCC algorithm. If there is a SCC such that and are in it, so the expression cannot be evaluated TRUE. Otherwise, it is possible.

**Pick’s Theorem:**

# Graph

## Tarjan

Complexity: O(V+E)

int n, m;

vector<int> g[MAXN];

int lbl[MAXN], low[MAXN], idx, cnt\_scc;

stack<int> st;

bool inSt[MAXN];

void dfs(int u) {

lbl[u] = low[u] = idx++;

st.push(u);

inSt[u] = 1;

for (int i = 0; i < g[u].size(); i++) {

int v = g[u][i];

if (lbl[v] == -1) {

dfs(v);

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

} else if (inSt[v]) {

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

}

}

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

printf("%d -> ", ++cnt\_scc);

int v;

do {

v = st.top();

st.pop();

inSt[v] = 0;

printf("%d; ", v);

} while (v != u);

putchar('\n');

}

}

void tarjan() {

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

lbl[i] = -1;

inSt[i] = 0;

}

idx = cnt\_scc = 0;

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

if (lbl[i] == -1)

dfs(i);

}

## Articulation

Complexity: O(V+E)

int n, m;

vector<int> g[MAXN];

int lbl[MAXN], low[MAXN], parent[MAXN], idx;

bool art[MAXN], has\_art;

void dfs(int u) {

int cnt = 0;

lbl[u] = low[u] = idx++;

for (int i = 0; i < g[u].size(); i++) {

int v = g[u][i];

if (lbl[v] == -1) {

parent[v] = u;

dfs(v);

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

if (low[v] >= lbl[u])

cnt++;

} else if (v != parent[u]) {

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

}

}

if (cnt > 1 || (lbl[u] != 0 && cnt > 0)) {

art[u] = 1;

has\_art = 1;

}

}

void articulation() {

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

lbl[i] = -1;

art[i] = 0;

}

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

if (lbl[i] == -1) {

idx = 0;

parent[i] = i;

dfs(i);

}

}

}

## Bridge

Complexity: O(V+E)

int n, m;

vector<int> g[MAXN];

int lbl[MAXN], low[MAXN], parent[MAXN], idx;

bool has\_bridge;

void dfs(int u) {

lbl[u] = low[u] = idx++;

bool parent\_found = 0;

for (int i = 0; i < g[u].size(); i++) {

int v = g[u][i];

if (lbl[v] == -1) {

parent[v] = u;

dfs(v);

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

if (low[v] == lbl[v]) {

printf("%d -> %d\n", u, v);

has\_bridge = 1;

}

} else if (!parent\_found && v == parent[u]) {

parent\_found = 1;

} else {

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

}

}

}

void bridge() {

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

lbl[i] = -1;

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

if (lbl[i] == -1) {

idx = 0;

parent[i] = i;

dfs(i);

}

}

}

## Hopcroft-Karp

Complexity: O(E sqrt(V))

int n, m;

vector<int> g1[MAXN];

int pair\_g1[MAXN], pair\_g2[MAXM], dist[MAXN];

bool bfs() {

queue<int> q;

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

dist[u] = INF;

if (pair\_g1[u] == 0) {

dist[u] = 0;

q.push(u);

}

}

dist[0] = INF;

while (!q.empty()) {

int u = q.front(); q.pop();

for (int i = 0; i < g1[u].size(); i++) {

int v = g1[u][i];

if (dist[pair\_g2[v]] == INF) {

dist[pair\_g2[v]] = dist[u] + 1;

q.push(pair\_g2[v]);

}

}

}

return dist[0] != INF;

}

bool dfs(int u) {

if (u == 0)

return 1;

for (int i = 0; i < g1[u].size(); i++) {

int v = g1[u][i];

if (dist[pair\_g2[v]] == dist[u] + 1 && dfs(pair\_g2[v])) {

pair\_g1[u] = v;

pair\_g2[v] = u;

return 1;

}

}

dist[u] = INF;

return 0;

}

int hk() {

memset(pair\_g1, 0, sizeof(pair\_g1));

memset(pair\_g2, 0, sizeof(pair\_g2));

int matching = 0;

while (bfs())

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

if (pair\_g1[u] == 0 && dfs(u))

matching++;

return matching;

}

## Dinic

Complexity: O(V^2 E)

struct Edge {

int v, c, f, next;

Edge() {}

Edge(int v, int c, int f, int next) : v(v), c(c), f(f), next(next) {}

};

int n, m, head[MAXN], lvl[MAXN], src, snk, work[MAXN];

Edge e[MAXM];

void init(int \_n, int \_src, int \_snk) {

n = \_n;

m = 0;

src = \_src;

snk = \_snk;

memset(head, -1, sizeof(head));

}

void addEdge(int u, int v, int c) {

e[m] = Edge(v, c, 0, head[u]);

head[u] = m++;

e[m] = Edge(u, 0, 0, head[v]);

head[v] = m++;

}

bool bfs() {

queue<int> q;

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

lvl[src] = 0;

q.push(src);

while (!q.empty()) {

int u = q.front(); q.pop();

for (int i = head[u]; i != -1; i = e[i].next) {

if (e[i].f < e[i].c && lvl[e[i].v] == -1) {

lvl[e[i].v] = lvl[u] + 1;

q.push(e[i].v);

if (e[i].v == snk)

return 1;

}

}

}

return 0;

}

int dfs(int u, int f) {

if (u == snk)

return f;

for (int &i = work[u]; i != -1; i = e[i].next) {

if (e[i].f < e[i].c && lvl[u] + 1 == lvl[e[i].v]) {

int minf = dfs(e[i].v, min(f, e[i].c - e[i].f));

if (minf > 0) {

e[i].f += minf;

e[i^1].f -= minf;

return minf;

}

}

}

return 0;

}

int dinic() {

int f, ret = 0;

while (bfs()) {

memcpy(work, head, n \* sizeof(int));

while (f = dfs(src, INF))

ret += f;

}

return ret;

}

## Lowest Common Ancestor

Complexity: < O(N logN), O(logN) >

#define MAXN 50000

#define LOGMAXN 16

int n, m, u, v, w;

int ancestor[MAXN][LOGMAXN], parent[MAXN], level[MAXN], dist[MAXN];

vector<pair<int, int> > g[MAXN];

void dfs(int u) {

for (int i = 0; i < g[u].size(); i++) {

int v = g[u][i].first, w = g[u][i].second;

if (v != parent[u]) {

parent[v] = u;

level[v] = level[u] + 1;

dist[v] = dist[u] + w;

dfs(v);

}

}

}

void pre() {

parent[0] = level[0] = dist[0] = 0;

dfs(0);

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

ancestor[i][0] = parent[i];

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

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

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

}

int lca(int u, int v) {

if (level[u] < level[v])

swap(u, v);

int log;

for (log = 1; 1<<log <= level[u]; log++);

log--;

for (int i = log; i >= 0; i--)

if (level[u] - (1<<i) >= level[v])

u = ancestor[u][i];

if (u == v)

return u;

for (int i = log; i >= 0; i--)

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

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

return parent[u];

}

## Min-Cost Max-Flow

Complexity: O(V E + f E log V), f = maximum flow

struct Edge {

int u, v, cap, flow, cost, next;

Edge() {}

Edge(int u, int v, int cap, int flow, int cost, int next)

: u(u), v(v), cap(cap), flow(flow), cost(cost), next(next) {}

};

int n, m, head[MAXN], src, snk;

Edge e[MAXM];

int pi[MAXN], dist[MAXN], path[MAXN], mincap[MAXN], vis[MAXN];

void init(int \_n, int \_src, int \_snk) {

n = \_n;

m = 0;

src = \_src;

snk = \_snk;

memset(head, -1, sizeof(head));

}

void addEdge(int u, int v, int cap, int cost) {

e[m] = Edge(u, v, cap, 0, cost, head[u]);

head[u] = m++;

e[m] = Edge(v, u, 0, 0, -cost, head[v]);

head[v] = m++;

}

int bellman\_ford() {

memset(pi, INF, sizeof(pi));

pi[src] = 0;

int flag = 1;

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

flag = 0;

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

if (e[j].cap == e[j].flow) continue;

if (pi[e[j].u] + e[j].cost < pi[e[j].v]) {

pi[e[j].v] = pi[e[j].u] + e[j].cost;

flag = 1;

}

}

}

return flag;

}

int dijkstra() {

priority\_queue<pii, vector<pii>, greater<pii> > heap;

memset(dist, INF, sizeof(dist));

memset(vis, 0, sizeof(vis));

dist[src] = 0;

mincap[src] = INF;

heap.push(mp(0, src));

while (!heap.empty()) {

int u = heap.top().se;

heap.pop();

if (vis[u]) continue;

vis[u] = 1;

for (int i = head[u]; i != -1; i = e[i].next) {

int v = e[i].v;

if (vis[v] || e[i].flow == e[i].cap) continue;

int w = dist[u] + e[i].cost + pi[u] - pi[v];

if (w < dist[v]) {

dist[v] = w;

path[v] = i;

mincap[v] = min(mincap[u], e[i].cap - e[i].flow);

heap.push(mp(dist[v], v));

}

}

}

// update potencials

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

pi[i] += dist[i];

return dist[snk] < INF;

}

pii mcmf() {

// set potencials

if (bellman\_ford())

return mp(-1, -1);

int cost = 0, flow = 0;

while (dijkstra()) {

// augment path and update cost

int f = mincap[snk];

for (int v = snk; v != src; ) {

int idx = path[v];

e[idx].flow += f;

e[idx^1].flow -= f;

cost += e[idx].cost \* f;

v = e[idx].u;

}

flow += f;

}

return mp(cost, flow);

}

## Heavy-Light Decomposition

struct SegTree {

vector<int> data, tree;

int sz;

SegTree(int tsz) : sz(1) {

while (sz < tsz) sz \*= 2;

data.resize(sz);

tree.resize(2\*sz);

}

inline int left(int u) { return u << 1; }

inline int right(int u) { return left(u) + 1; }

void init(int u, int l, int r) {

if (l == r) { tree[u] = data[l]; return; }

int m = (l + r) >> 1;

init(left(u), l, m);

init(right(u), m+1, r);

tree[u] = max(tree[left(u)], tree[right(u)]);

}

void init() { init(1, 0, sz-1); }

int query(int u, int l, int r, int a, int b) {

if (a <= l && r <= b) return tree[u];

int m = (l + r) >> 1, ret = 0;

if (a <= m) ret = query(left(u), l, m, a, b);

if (m < b) ret = max(ret, query(right(u), m+1, r, a, b));

return ret;

}

int query(int a, int b) { return query(1, 0, sz-1, a, b); }

void update(int u, int l, int r, int pos, int val) {

if (l == r) { tree[u] = val; return; }

int m = (l + r) >> 1;

if (pos <= m) update(left(u), l, m, pos, val);

else update(right(u), m+1, r, pos, val);

tree[u] = max(tree[left(u)], tree[right(u)]);

}

void update(int pos, int val) { update(1, 0, sz-1, pos, val); }

};

struct edge {

int u, v, w, next;

edge() {}

edge(int u, int v, int w, int next) : u(u), v(v), w(w), next(next) {}

};

int n, m, head[MAX];

edge e[2\*MAX];

int dad[MAX], lvl[MAX], chd[MAX], sz[MAX], heavy[MAX];

int nump, psize[MAX], pfirst[MAX], path[MAX], offset[MAX];

vector<int> walk;

vector<SegTree> ptree;

void init() {

m = 0;

memset(head, -1, sizeof(head));

memset(dad, 0, sizeof(dad));

}

void addEdge(int u, int v, int w, bool rev = false) {

e[m] = edge(u, v, w, head[u]);

head[u] = m++;

if (!rev) addEdge(v, u, w, true);

}

void dfs(int u) {

walk.push\_back(u);

sz[u] = 1;

for (int i = head[u]; i != -1; i = e[i].next) {

int v = e[i].v;

if (!dad[v]) {

dad[v] = u,lvl[v] = lvl[u] + 1, dfs(v);

sz[u] += sz[v];

}

}

for (int i = head[u]; i != -1; i = e[i].next) {

int v = e[i].v;

if (dad[v] == u && 2\*sz[v] >= sz[u]) {

heavy[v] = 1;

break;

}

}

heavy[u] = 0;

}

void hl\_init() {

walk.clear();

dad[1] = 1, lvl[1] = 0, dfs(1);

nump = 0;

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

int u = walk[i];

if (!heavy[u]) {

offset[u] = 0;

path[u] = nump;

pfirst[nump] = u;

psize[nump++] = 1;

}

else {

offset[u] = offset[dad[u]] + 1;

path[u] = path[dad[u]];

psize[path[u]]++;

}

}

ptree.clear(); ptree.reserve(nump);

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

ptree.push\_back(SegTree(psize[i]));

for (int i = 0; i < m; i += 2) {

int u = e[i].u, v = e[i].v;

if (u != dad[v]) swap(u, v);

ptree[path[v]].data[offset[v]] = e[i].w;

}

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

ptree[i].init();

}

int lca(int u, int v) {

int fpu = pfirst[path[u]], fpv = pfirst[path[v]];

while (fpu != fpv) {

if (lvl[fpu] > lvl[fpv])

u = dad[fpu], fpu = pfirst[path[u]];

else

v = dad[fpv], fpv = pfirst[path[v]];

}

return lvl[u] < lvl[v] ? u : v;

}

void update(int idx, int val) {

int u = e[idx].u, v = e[idx].v;

if (u != dad[v]) swap(u, v);

ptree[path[v]].update(offset[v], val);

}

int query(int u, int v, bool up = false) {

if (!up) {

int w = lca(u, v);

return max(query(u, w, true), query(v, w, true));

}

int ret = 0;

int fpu = pfirst[path[u]], fpv = pfirst[path[v]];

while (fpu != fpv) {

ret = max(ret, ptree[path[u]].query(0, offset[u]));

u = dad[fpu], fpu = pfirst[path[u]];

}

if (u != v)

ret = max(ret, ptree[path[u]].query(offset[v]+1, offset[u]));

return ret;

}

# String

## KMP

Complexity: O(N)

int pi[MAXS];

void kmp\_table(char \*pattern, int m) {

pi[0] = -1;

for (int i = 1, k = -1; i < m; i++) {

while (k >= 0 && pattern[k+1] != pattern[i])

k = pi[k];

if (pattern[k+1] == pattern[i])

k++;

pi[i] = k;

}

}

void kmp\_search(char \*text, char \*pattern) {

int n = strlen(text), m = strlen(pattern);

kmp\_table(pattern, m);

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

while (k >= 0 && pattern[k+1] != text[i])

k = pi[k];

if (pattern[k+1] == text[i])

k++;

if (k+1 == m) {

printf("Match at %d\n", i-k);

k = pi[k];

}

}

}

## String Hashing

#define B 33

ull powB[MAX];

void init() {

powB[0] = 1;

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

powB[i] = B\*powB[i-1];

}

void calc\_hash(char \*str, ull \*h) {

h[0] = 0;

for (int i = 0; str[i]; i++)

h[i+1] = B\*h[i] + str[i];

}

ull get\_hash(ull \*h, int l, int r) {

return h[r] - h[l]\*powB[r-l];

}

## Manacher’s Algorithm

Complexity: O(N)

char s[MAXN];

int p[2\*MAXN]; // length of the palindrome centered at position (i-1)/2;

void manacher() {

int m = 0;

char t[2\*MAXN];

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

t[m++] = '#';

t[m++] = s[i];

p[i] = 0;

}

t[m++] = '#';

int c = 0, r = 0;

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

int i\_ = 2 \* c - i;

p[i] = r > i ? min(r-i, p[i\_]) : i & 1;

while (0 <= i-p[i]-1 && i+p[i]+1 < m && t[i-p[i]-1] == t[i+p[i]+1])

p[i] += 2;

if (i + p[i] > r) {

c = i;

r = i + p[i];

}

}

}

## Aho-Corasick

Complexity: < O(|S|), O(sum(|Si|)), O(|S|) >

struct Node {

map<char, int> adj;

int fail;

pii match;

int next;

void init() {

adj.clear();

fail = -1;

match = mp(-1, -1);

next = -1;

}

int getChild(const char &c) {

map<char, int>::iterator it = adj.find(c);

if (it != adj.end())

return it->second;

return -1;

}

};

int qntNodes, qntPatts;

Node trie[MAX];

void init() {

trie[0].init();

qntNodes = 1;

qntPatts = 0;

}

void addWord(const char \*word) {

int node = 0, aux = -1;

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

aux = trie[node].getChild(word[i]);

if (aux == -1) {

trie[qntNodes].init();

aux = qntNodes++;

trie[node].adj[word[i]] = aux;

}

node = aux;

}

trie[node].match = mp(qntPatts++, strlen(word));

}

void build() {

queue<int> q;

map<char, int>::iterator it;

trie[0].fail = -1;

q.push(0);

while (!q.empty()) {

int u = q.front();

q.pop();

for (it = trie[u].adj.begin(); it != trie[u].adj.end(); it++) {

int v = it->second;

char c = it->first;

q.push(v);

int f = trie[u].fail;

while (f >= 0 && trie[f].getChild(c) == -1)

f = trie[f].fail;

f = f >= 0 ? trie[f].getChild(c) : 0;

trie[v].fail = f;

trie[v].next = trie[f].match.fi >= 0 ? f : trie[f].next;

}

}

}

void search(const char \*text) {

int node = 0;

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

while (node >= 0 && trie[node].getChild(text[i]) == -1)

node = trie[node].fail;

node = node >= 0 ? trie[node].getChild(text[i]) : 0;

int aux = node;

while (aux >= 0) {

if (trie[aux].match.fi >= 0) {

// do something with the match

printf("patt: %d, pos: %d\n",

trie[aux].match.fi,

i - trie[aux].match.se + 1);

}

aux = trie[aux].next;

}

}

}

## Suffix Array and Longest Common Prefix

Complexity: < O(N logN), O(N) >

//Output:

// pos = The suffix array. Contains the n suffixes of str sorted in

// lexicographical order. Each suffix is represented as a

// single integer (the position of str where it starts).

// rank = The inverse of the suffix array.

// rank[i] = the index of the suffix str[i..n) in the pos array.

// (In other words, pos[i] = k <==> rank[k] = i)

// With this array, you can compare two suffixes in O(1):

// Suffix str[i..n) is smaller than str[j..n) iff rank[i] < rank[j]

int n; // length of the string

char str[MAXN];

int rank[MAXN], pos[MAXN], cnt[MAXN], next[MAXN];

bool bh[MAXN], b2h[MAXN];

bool cmp(int a, int b) {

return str[a] < str[b];

}

void suffix\_array() {

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

pos[i] = i;

sort(pos, pos+n, cmp);

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

bh[i] = (i == 0 || str[pos[i]] != str[pos[i-1]]);

b2h[i] = 0;

}

for (int h = 1; h < n; h <<= 1) {

int buckets = 0;

for (int i = 0, j; i < n; i = j) {

j = i + 1;

while (j < n && !bh[j])

j++;

next[i] = j;

buckets++;

}

if (buckets == n)

break;

for (int i = 0; i < n; i = next[i]) {

cnt[i] = 0;

for (int j = i; j < next[i]; j++)

rank[pos[j]] = i;

}

cnt[rank[n-h]]++;

b2h[rank[n-h]] = 1;

for (int i = 0; i < n; i = next[i]) {

for (int j = i; j < next[i]; j++) {

int s = pos[j] - h;

if (s >= 0) {

int head = rank[s];

rank[s] = head + cnt[head]++;

b2h[rank[s]] = 1;

}

}

for (int j = i; j < next[i]; j++) {

int s = pos[j] - h;

if (s >= 0 && b2h[rank[s]]) {

for (int k = rank[s] + 1; !bh[k] && b2h[k]; k++)

b2h[k] = 0;

}

}

}

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

pos[rank[i]] = i;

bh[i] |= b2h[i];

}

}

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

rank[pos[i]] = i;

}

int height[MAXN];

void getHeight() {

height[0] = 0;

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

if (rank[i] > 0) {

int j = pos[rank[i] - 1];

while (i + h < n && j + h < n && str[i+h] == str[j+h])

h++;

height[rank[i]] = h;

if (h > 0)

h--;

}

}

}

# Dynamic Programming

## Optimal Binary Search Tree

Complexity: O(N^3)

int n, p[MAXN];

int c[MAXN][MAXN], f[MAXN][MAXN], r[MAXN][MAXN];

void obst() {

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

c[i][i-1] = 0;

c[n+1][n] = 0;

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

c[i][i] = p[i];

f[i][i] = p[i];

r[i][i] = i;

}

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

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

int j = i+d;

c[i][j] = INF;

f[i][j] = f[i][j-1] + p[j];

int rmin = r[i][j-1], rmax = r[i+1][j];

for (int k = rmin; k <= rmax; k++) {

int t = c[i][k-1] + c[k+1][j];

if (t < c[i][j]) {

c[i][j] = t;

r[i][j] = k;

}

}

c[i][j] += f[i][j];

}

}

}

## Longest Increasing Subsequence

Complexity: O(N logN)

int n, m, a[MAXN], b[MAXN], p[MAXN];

void lis() {

int u, v;

b[m++] = 0;

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

if (a[b[m-1]] < a[i]) {

p[i] = b[m-1];

b[m++] = i;

continue;

}

for (u = 0, v = m-1; u < v; ) {

int c = (u + v)/2;

if (a[b[c]] < a[i])

u = c + 1;

else

v = c;

}

if (a[i] < a[b[u]]) {

if (u > 0)

p[i] = b[u-1];

b[u] = i;

}

}

for (u = m, v = b[m-1]; u--; v = p[v]) {

b[u] = v;

}

}

## Longest Common Increasing Subsequence

Complexity: O(N^2)

int n, m, a[MAXN], b[MAXN];

int c[MAXN], prev[MAXN], seq[MAXN];

void lcis() {

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

c[j] = 0;

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

int actual = 0, last = -1;

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

if (a[i] == b[j] && actual+1 > c[j]) {

c[j] = actual+1;

prev[j] = last;

} else if (a[i] > b[j] && actual < c[j]) {

actual = c[j];

last = j;

}

}

}

int length = 0, index = -1;

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

if (c[j] > length) {

length = c[j];

index = j;

}

}

int len = length;

while (index != -1) {

seq[--len] = b[index];

index = prev[index];

}

printf("length: %d\n", length);

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

printf("%d ", seq[i]);

printf("\n");

}

## Weighted Activity Selection

Complexity: O(N logN)

#include <cstdio>

#include <algorithm>

using namespace std;

#define MAXN 10005

struct Event {

int b, e, w;

Event () {}

Event (int b, int e, int w) : b(b), e(e), w(w) {}

bool operator< (const Event& o) const {

return e != o.e ? e < o.e ? b < o.b;

}

};

int n;

Event e[MAXN];

int dp[MAXN];

int main() {

scanf("%d", &n);

e[0] = Event(0, 0, 0);

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

scanf("%d %d %d", &e[i].b, &e[i].e, &e[i].w);

sort(e+1, e+n+1);

dp[0] = 0;

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

int lo = 0, hi = i-1;

while (lo < hi) {

int mid = (lo + hi + 1)/2;

if (e[mid].e > e[i].b)

hi = mid - 1;

else

lo = mid;

}

dp[i] = max(dp[i-1], e[i].w + dp[lo]);

}

printf("Max weight: %d\n", dp[n]);

}

# Data Structure

## Segment Tree with Lazy Propagation

Complexity: < O(N), O(logN) >

#define left(x) ((x) << 1)

#define right(x) (left(x) + 1)

ll tree[4\*MAXN], lazy[4\*MAXN];

void propagate(int node, int lo, int hi) {

tree[node] += lazy[node] \* (hi-lo+1);

if (lo != hi) {

lazy[left(node)] += lazy[node];

lazy[right(node)] += lazy[node];

}

lazy[node] = 0;

}

void update(int node, int lo, int hi, int i, int j, int val) {

if (i <= lo && hi <= j) {

lazy[node] += val;

return;

}

int mid = (lo + hi)/2;

if (i <= mid)

update(left(node), lo, mid, i, j, val);

if (j > mid)

update(right(node), mid+1, hi, i, j, val);

propagate(left(node), lo, mid);

propagate(right(node), mid+1, hi);

tree[node] = tree[left(node)] + tree[right(node)];

}

ll query(int node, int lo, int hi, int i, int j) {

propagate(node, lo, hi);

if (i <= lo && hi <= j)

return tree[node];

ll ret = 0;

int mid = (lo + hi)/2;

if (i <= mid)

ret = query(left(node), lo, mid, i, j);

if (j > mid)

ret += query(right(node), mid+1, hi, i, j);

return ret;

}

# Geometry

## Template

struct Point {

double x, y;

Point() {}

Point(double x, double y) : x(x), y(y) {}

Point operator+ (const Point &o) const { return Point(x + o.x, y + o.y); }

Point operator- (const Point &o) const { return Point(x - o.x, y - o.y); }

Point operator\* (const double &o) const { return Point(x \* o, y \* o); }

Point operator/ (const double &o) const { return Point(x / o, y / o); }

double operator\* (const Point &o) const { return x \* o.x + y \* o.y; }

double operator% (const Point &o) const { return x \* o.y - o.x \* y; }

bool operator== (const Point &o) const {

return cmp\_double(x, o.x) == 0 && cmp\_double(y, o.y) == 0;

}

bool operator< (const Point &o) const {

return x != o.x ? x < o.x : y < o.y;

}

};

typedef Point Vector;

double abs(Point p) {

return sqrt(p \* p);

}

Vector norm(Vector v) {

return v / abs(v);

}

double ccw(Point p, Point q, Point r) {

return (q - p) % (r - p);

}

struct Segment {

Point p, q;

Segment() {}

Segment(Point p, Point q) : p(p), q(q) {}

};

bool in\_segment(Point p, Segment s) {

double t;

Vector v = s.q - s.p;

if (cmp\_double(v.x, 0) != 0)

t = (p.x - s.p.x) / v.x;

else

t = (p.y - s.p.y) / v.y;

return cmp\_double(t, 0) >= 0 && cmp\_double(t, 1) <= 0 && s.p + v \* t == p;

}

struct Line {

Vector v;

Point p;

int a, b, c;

void init() {

a = -v.y;

b = v.x;

c = a \* p.x + b \* p.y;

int d = abs(\_\_gcd(a, \_\_gcd(b, c)));

if (d != 1)

a /= d, b /= d, c /= d;

if (a < 0)

a = -a, b = -b, c = -c;

else if (a == 0 && b < 0)

b = -b, c = -c;

}

Line() {}

Line(Point p, Point q) : v(q-p), p(p) {

init();

}

Line(Segment s) : v(s.q-s.p), p(p) {

init();

}

Point operator() (double t) const { return p + v \* t; }

Vector normal() {

return Vector(-v.y, v.x);

}

};

pair<double, double> line\_intersection(Line a, Line b) {

double den = a.v % b.v;

if (den == 0)

return make\_pair(inf, inf);

double t = -(b.v % (b.p - a.p)) / den;

double s = -(a.v % (b.p - a.p)) / den;

return make\_pair(t, s);

}

Point segment\_intersection(Segment a, Segment b) {

Line la = Line(a), lb = Line(b);

pair<double, double> pdd = line\_intersection(la, lb);

double t = pdd.first, s = pdd.second;

if (t == inf) {

if (in\_segment(b.p, a))

return b.p;

if (in\_segment(b.q, a))

return b.q;

if (in\_segment(a.p, b))

return a.p;

if (in\_segment(a.q, b))

return a.q;

return Point(inf, inf);

}

if (cmp\_double(t, 0) < 0 || cmp\_double(t, 1) > 0)

return Point(inf, inf);

if (cmp\_double(s, 0) < 0 || cmp\_double(s, 1) > 0)

return Point(inf, inf);

return la(t);

}

double distPointToLine(Point p, Line l) {

Vector n = l.normal();

return (l.p - p) \* n / abs(n);

}

struct Circle {

Point p;

double r;

Circle() {}

Circle(Point p, double r) : p(p), r(r) {}

};

bool in\_circle(const Circle &c, const Point &p) {

return cmp\_double(abs(c.p - p), c.r) <= 0;

}

Point circumcenter(Point p, Point q, Point r) {

Point a = p - r, b = q - r, c = Point(a\*(p+r)/2, b\*(q+r)/2);

return Point(c % Point(a.y, b.y), Point(a.x, b.x) % c)/(a % b);

}

Point incenter(Point p, Point q, Point r) {

double a = abs(r - q), b = abs(r - p), c = abs(q - p);

return (p \* a + q \* b + r \* c) / (a + b + c);

}

## Monotone Chain Convex Hull

Complexity: O(N logN)

int n, k;

Point p[MAXN], h[MAXN];

void convex\_hull() {

sort(p, p+n);

k = 0;

h[k++] = p[0];

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

if (i != n-1 && ccw(p[0], p[n-1], p[i]) >= 0) continue;

while (k > 1 && ccw(h[k-2], h[k-1], p[i]) <= 0) k--;

h[k++] = p[i];

}

for (int i = n-2, lim = k; i >= 0; i--) {

if (i != 0 && ccw(p[n-1], p[0], p[i]) >= 0) continue;

while (k > lim && ccw(h[k-2], h[k-1], p[i]) <= 0) k--;

h[k++] = p[i];

}

}

## Smallest Enclosing Circle

Complexity: O(N^2)

int n;

Point p[MAXN];

Circle spanning\_circle() {

random\_shuffle(p, p+n);

Circle c(Point(), -1);

for (int i = 0; i < n; i++) if (!in\_circle(c, p[i])) {

c = Circle(p[i], 0);

for (int j = 0; j < i; j++) if (!in\_circle(c, p[j])) {

c = Circle((p[i] + p[j])/2, abs(p[i] - p[j])/2);

for (int k = 0; k < j; k++) if (!in\_circle(c, p[k])) {

Point o = circumcenter(p[i], p[j], p[k]);

c = Circle(o, abs(o - p[k]));

}

}

}

return c;

}

## Closest Pair of Points

Complexity: O(N logN)

#include <cstdio>

#include <cmath>

#include <algorithm>

#include <set>

using namespace std;

struct Point {

int x, y;

Point(int x = 0, int y = 0) : x(x), y(y) {}

Point operator- (const Point &o) const { return Point(x - o.x, y - o.y); }

int operator\* (const Point &o) const { return x \* o.x + y \* o.y; }

bool operator< (const Point &o) const {

return y != o.y ? y < o.y : x < o.x;

}

};

bool cmpx(const Point &p, const Point &q) {

return p.x != q.x ? p.x < q.x : p.y < q.y;

}

double abs(const Point &p) {

return sqrt(p \* p);

}

int main() {

int n;

Point pnts[MAXN];

set<Point> box;

set<Point>::iterator it;

scanf("%d", &n);

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

scanf("%d %d", &pnts[i].x, &pnts[i].y);

sort(pnts, pnts+n, cmpx);

double best = inf;

box.insert(pnts[0]);

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

while (j < i && pnts[i].x - pnts[j].x > best)

box.erase(pnts[j++]);

for (it = box.lower\_bound(Point(pnts[i].x-best, pnts[i].y-best));

it != box.end() && it->y <= pnts[i].y + best; it++) {

best = min(best, abs(pnts[i] - \*it));

}

box.insert(pnts[i]);

}

printf("%.2lf\n", best);

}

# Math

## Sieve, primality, factorization, phi

int np, p[MAXN];

int lp[MAXN];

void sieve(int n) {

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

lp[i] = i;

for (int i = 4; i < n; i += 2)

lp[i] = 2;

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

if (lp[i] == i)

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

lp[j] = i;

np = 0;

p[np++] = 2;

for (int i = 3; i < n; i += 2)

if (lp[i] == i)

p[np++] = i;

}

int nf, f[MAXN], e[MAXN];

void factor(int n) {

nf = 0;

for (int i = 0; n != 1 && p[i]\*p[i] <= n; i++) {

if (n % p[i] == 0) {

f[nf] = p[i];

e[nf] = 1;

n /= p[i];

while (n % p[i] == 0) {

e[nf]++;

n /= p[i];

}

nf++;

}

}

if (n != 1) {

f[nf] = n;

e[nf] = 1;

nf++;

}

}

int \_phi(int n) {

int ret = 1;

for (int i = 0; n != 1 && p[i]\*p[i] <= n; i++) {

if (n % p[i] == 0) {

int pk = p[i];

n /= p[i];

while (n % p[i] == 0) {

pk \*= p[i];

n /= p[i];

}

ret \*= pk - pk/p[i];

}

}

if (n != 1)

ret \*= n-1;

return ret;

}

int phi[MAXN];

void build\_phi(int 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 \* (i-1);

}

## Chinese Remainder Algorithm

#include <cstdio>

#include <algorithm>

using namespace std;

const int MAXN = 100010;

typedef pair<int, int> tpii;

struct teq {

int r, n; // x = r (mod n)

};

int qnt;

teq eqs[MAXN];

tpii egcd(int a, int b) {

int x = 0, lastx = 1, auxx;

int y = 1, lasty = 0, auxy;

while (b) {

int q = a / b, r = a % b;

a = b, b = r;

auxx = x;

x = lastx - q\*x, lastx = auxx;

auxy = y;

y = lasty - q\*y, lasty = auxy;

}

return make\_pair(lastx, lasty);

}

int chinese\_remainder\_algorithm() {

int beta, sum = 0, n = 1;

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

n \*= eqs[i].n;

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

beta = egcd(eqs[i].n, n/eqs[i].n).second;

while (beta < 0)

beta += eqs[i].n;

sum += (eqs[i].r \* beta \* n/eqs[i].n) % n;

}

return sum;

}

int main() {

scanf("%d", &qnt);

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

scanf("%d %d", &eqs[i].r, &eqs[i].n);

printf("%d\n", chinese\_remainder\_algorithm());

}

## FFT

typedef complex<long double> pt;

pt tmp[1<<20];

void fft(pt \*in, int sz, bool inv) {

if (sz == 1)

return;

for (int i = 0, j = 0, h = sz >> 1; i < sz; i += 2, j++) {

in[j] = in[i];

tmp[h+j] = in[i+1];

}

for (int i = sz >> 1; i < sz; i++)

in[i] = tmp[i];

sz >>= 1;

pt \*even = in, \*odd = in + sz;

fft(even, sz, inv);

fft(odd, sz, inv);

long double p = (inv ? -1 : 1) \* M\_PI / sz;

pt w = pt(cosl(p), sinl(p)), w\_i = 1;

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

pt conv = w\_i \* odd[i];

odd[i] = even[i] - conv;

even[i] += conv;

w\_i \*= w;

}

}

## Polynomial

struct Poly {

int n;

double a[MAXN];

Poly(int n = 0) : n(n) { memset(a, 0, sizeof(a)); }

Poly(const Poly &o) : n(o.n) { memcpy(a, o.a, sizeof(a)); }

const double& operator[] (int i) const { return a[i]; }

double& operator[] (int i) { return a[i]; }

double operator() (double x) const {

double ret = 0;

for (int i = n; i >= 0; i--)

ret = ret \* x + a[i];

return ret;

}

Poly operator+ (const Poly &o) const {

Poly ret = o;

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

ret[i] += a[i];

ret.n = max(n, o.n);

return ret;

}

Poly operator- (const Poly &o) const {

Poly ret = o;

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

ret[i] -= a[i];

ret.n = max(n, o.n);

return ret;

}

Poly operator\* (const Poly &o) const {

Poly ret(n + o.n);

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

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

ret[i+j] += a[i] \* o[j];

return ret;

}

};

Poly fastMult(const Poly &p, const Poly &q) {

int sz = 1 << (32 - \_\_builtin\_clz(p.n + q.n + 1));

pt pin[sz], qin[sz];

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

if (i <= p.n)

pin[i] = p[i];

else

pin[i] = 0;

if (i <= q.n)

qin[i] = q[i];

else

qin[i] = 0;

}

fft(pin, sz, 0);

fft(qin, sz, 0);

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

pin[i] \*= qin[i];

fft(pin, sz, 1);

Poly ret(p.n + q.n);

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

ret[i] = pin[i].real() / sz;

while (ret.n > 0 && cmp(ret[ret.n], 0) == 0)

ret.n--;

return ret;

}

Poly diff(const Poly &p) {

Poly ret(p.n-1);

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

ret[i-1] = i \* p[i];

return ret;

}

pair<Poly, double> ruffini(const Poly &p, double x) {

if (p.n == 0)

return make\_pair(Poly(), 0);

Poly ret(p.n-1);

for (int i = p.n; i > 0; i--)

ret[i-1] = ret[i] \* x + p[i];

return make\_pair(ret, ret[0] \* x + p[0]);

}

/\*\*

\* Find a root in range [lo, hi] assuming that exists only one root in [lo, hi]

\* pair::second is true if exists a root in the given range or false otherwise

\* pair::first is the root if pair::second is true or 0 if false

\*/

pair<double, int> findRoot(const Poly &p, double lo, double hi) {

if (cmp(p(lo), 0) == 0)

return make\_pair(lo, 1);

if (cmp(p(hi), 0) == 0)

return make\_pair(hi, 1);

if (cmp(p(lo), 0) == cmp(p(hi), 0))

return make\_pair(0, 0);

if (cmp(p(lo), p(hi)) > 0)

swap(lo, hi);

while (cmp(lo, hi) != 0) {

double mid = (lo + hi) / 2;

double val = p(mid);

if (cmp(val, 0) == 0)

lo = hi = mid;

else if (cmp(val, 0) < 0)

lo = mid;

else

hi = mid;

}

return make\_pair(lo, 1);

}

/\*\*

\* Return a vector of all real roots with their multiplicity in ascending order

\*/

vector<double> roots(const Poly &p) {

vector<double> ret;

if (p.n == 1) {

ret.push\_back(-p[0] / p[1]);

}

else {

vector<double> r = roots(diff(p));

r.push\_back(-MAXX);

r.push\_back(MAXX);

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

for (int i = 0, j = 1; j < (int) r.size(); i++, j++) {

pair<double, int> pr = findRoot(p, r[i], r[j]);

if (pr.second)

ret.push\_back(pr.first);

}

}

return ret;

}

## Bignum

#include <cstring>

#include <algorithm>

#include <limits>

using namespace std;

typedef long long ll;

typedef unsigned long long ull;

const int MAXD = 1005, DIG = 9, BASE = 1000000000;

const ull BOUND = numeric\_limits <ull> :: max() - (ull) BASE \* BASE;

struct bignum

{

int D, digits[MAXD / DIG + 2];

int sign;

inline void trim () {

while (D > 1 && digits[D - 1] == 0)

D--;

}

inline void init (ll x) {

memset(digits, 0, sizeof(digits));

D = 0;

if (x < 0) {

sign = -1;

x = -x;

}

else {

sign = 1;

}

do {

digits[D++] = x % BASE;

x /= BASE;

} while (x > 0);

}

inline bignum (ll x) {

init(x);

}

inline bignum (int x = 0) {

init(x);

}

inline bignum (char \*s) {

memset(digits, 0, sizeof(digits));

if (s[0] == '-') {

sign = -1;

s++;

}

else {

sign = 1;

}

int len = strlen(s), first = (len + DIG - 1) % DIG + 1;

D = (len + DIG - 1) / DIG;

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

digits[D - 1] = digits[D - 1] \* 10 + s[i] - '0';

for (int i = first, d = D - 2; i < len; i += DIG, d--)

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

digits[d] = digits[d] \* 10 + s[j] - '0';

trim();

}

inline char \*str () {

trim();

char \*buf = new char[DIG \* D + 2];

int pos = 0, d = digits[D - 1];

if (sign == -1)

buf[pos++] = '-';

do {

buf[pos++] = d % 10 + '0';

d /= 10;

} while (d > 0);

reverse(buf + (sign == -1 ? 1 : 0), buf + pos);

for (int i = D - 2; i >= 0; i--, pos += DIG)

for (int j = DIG - 1, t = digits[i]; j >= 0; j--) {

buf[pos + j] = t % 10 + '0';

t /= 10;

}

buf[pos] = '\0';

return buf;

}

inline bool operator < (const bignum &o) const {

if (sign != o.sign)

return sign < o.sign;

if (D != o.D)

return sign \* D < o.sign \* o.D;

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

if (digits[i] != o.digits[i])

return sign \* digits[i] < o.sign \* o.digits[i];

return false;

}

inline bool operator > (const bignum &o ) const {

if (sign != o.sign)

return sign > o.sign;

if (D != o.D)

return sign \* D > o.sign \* o.D;

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

if (digits[i] != o.digits[i])

return sign \* digits[i] > o.sign \* o.digits[i];

return false;

}

inline bool operator == (const bignum &o) const {

if (sign != o.sign)

return false;

if (D != o.D)

return false;

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

if (digits[i] != o.digits[i])

return false;

return true;

}

inline bignum operator << (int p) const {

bignum temp;

temp.D = D + p;

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

temp.digits[i + p] = digits[i];

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

temp.digits[i] = 0;

return temp;

}

inline bignum operator >> (int p) const {

bignum temp;

temp.D = D - p;

for (int i = 0; i < D - p; i++)

temp.digits[i] = digits[i + p];

for (int i = D - p; i < D; i++)

temp.digits[i] = 0;

return temp;

}

inline bignum range (int a, int b) const {

bignum temp = 0;

temp.D = b - a;

for (int i = 0; i < temp.D; i++)

temp.digits[i] = digits[i + a];

return temp;

}

inline bignum abs () const {

bignum temp = \*this;

temp.sign = 1;

return temp;

}

inline bignum operator + (const bignum &o) const {

if (sign != o.sign) {

if (sign == 1)

return \*this - o.abs();

else

return o - this->abs();

}

bignum sum = o;

int carry = 0;

for (sum.D = 0; sum.D < D || carry > 0; sum.D++) {

sum.digits[sum.D] += (sum.D < D ? digits[sum.D] : 0) + carry;

carry = 0;

if (sum.digits[sum.D] >= BASE) {

sum.digits[sum.D] -= BASE;

carry = 1;

}

}

sum.D = max(sum.D, o.D);

sum.trim();

return sum;

}

inline bignum operator - (const bignum &o) const {

if (sign != o.sign) {

if (sign == 1)

return \*this + o.abs();

else

return -(this->abs() + o);

}

else if (sign == -1) {

return o.abs() - this->abs();

}

bignum diff, temp;

if (o > \*this) {

diff = o;

diff.sign = -1;

temp = \*this;

}

else {

diff = \*this;

temp = o;

}

for (int i = 0, carry = 0; i < temp.D || carry > 0; i++) {

diff.digits[i] -= (i < temp.D ? temp.digits[i] : 0) + carry;

carry = 0;

if (diff.digits[i] < 0) {

diff.digits[i] += BASE;

carry = 1;

}

}

diff.trim();

return diff;

}

inline bignum operator - () const {

bignum temp = \*this;

temp.sign = -temp.sign;

return temp;

}

inline bignum operator \* (const bignum &o) const {

bignum prod = 0;

ull sum = 0, carry = 0;

for (prod.D = 0; prod.D < D + o.D - 1 || carry > 0; prod.D++) {

sum = carry % BASE;

carry /= BASE;

for (int j = max(prod.D-o.D+1, 0); j <= min(D-1, prod.D); j++) {

sum += (ull) digits[j] \* o.digits[prod.D - j];

if (sum >= BOUND) {

carry += sum / BASE;

sum %= BASE;

}

}

carry += sum / BASE;

prod.digits[prod.D] = sum % BASE;

}

prod.sign = sign \* o.sign;

prod.trim();

return prod;

}

inline double double\_div (const bignum &o) const {

double val = 0, oval = 0;

int num = 0, onum = 0;

for (int i = D - 1; i >= max(D - 3, 0); i--, num++)

val = val \* BASE + digits[i];

for (int i = o.D - 1; i >= max(o.D - 3, 0); i--, onum++)

oval = oval \* BASE + o.digits[i];

return sign \* o.sign \* val / oval \* (D - num > o.D - onum ? BASE : 1);

}

inline pair<bignum, bignum> divmod (const bignum &o) const {

if (sign != o.sign) {

pair<bignum, bignum> p = (this->abs()).divmod(o.abs());

p.first.sign = -1;

p.second.sign = sign;

return p;

}

else if (sign == -1) {

pair<bignum, bignum> p = (this->abs()).divmod(o.abs());

p.second.sign = sign;

return p;

}

bignum quot = 0, rem = \*this, temp;

for (int i = D - o.D; i >= 0; i--) {

temp = rem.range(i, rem.D);

int div = (int) temp.double\_div(o);

bignum mult = o \* div;

while (div > 0 && temp < mult) {

mult = mult - o;

div--;

}

while (div + 1 < BASE && !(temp < mult + o)) {

mult = mult + o;

div++;

}

rem = rem - (o \* div << i);

if (div > 0) {

quot.digits[i] = div;

quot.D = max(quot.D, i + 1);

}

}

quot.trim();

rem.trim();

return make\_pair(quot, rem);

}

inline bignum operator / (const bignum &o) const {

return divmod(o).first;

}

inline bignum operator % (const bignum &o) const {

return divmod(o).second;

}

inline bignum power (int exp) const {

bignum p = 1, temp = \*this;

while (exp > 0) {

if (exp & 1) p = p \* temp;

if (exp > 1) temp = temp \* temp;

exp >>= 1;

}

return p;

}

};

inline bignum gcd (bignum a, bignum b) {

bignum t;

while (!(b == 0)) {

t = a % b;

a = b;

b = t;

}

return a;

}