**ICPC REFERENCE NOTEBOOK**

**Griffith Did Nothing Wrong**

**[University of Petroleum and Energy Studies, Dehradun]**

**MACROS**

**#ifdef** DEBUG

**#define** trace3(x,y,z) cerr<<\_\_FUNCTION\_\_<<":"<<\_\_LINE\_\_<<": "#x" = "<<x<<" | "#y" = "<<y<<" | "#z" = "<<z<<endl;

**#else** **#define** trace3(x,y,z) **#endif**

//Use emplace\_back instead of push\_back

**FUNCTIONS**

**FastRead**

**inline** **long** **long** **int** **frl**(){

**register** **long** **long** **int** c=**getchar**();

**long** **long** **int** x=0LL, neg=0LL; //Bool neg is better?

**for**(; ((c<48LL || c>57LL) && c != '-' && c!=EOF); c = **getchar**());

**if**(c==EOF) **return** EOF;

**if**(c=='-') {

neg = 1LL;

c = **getchar**();

}

**for**(; c>47LL && c<58LL ; c = **getchar**()) {

x = (x<<1LL) + (x<<3LL) + c - 48LL;

}

**if**(neg) x = -x;

**return** x; }

**GCD/LCM**

ll **gcd**(ll a,ll b){

**return** b==0?a:gcd(b,a%b);}

ll **lcm**(ll a,ll b){

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

**long** **long** C(**int** n, **int** r)

{

**if**(r>n) **return** 0;

**if**(r > n / 2) r = n - r; // because C(n, r) == C(n, n - r)

**long** **long** ans = 1;

**int** i;

**for**(i = 1; i <= r; i++) {

ans \*= n - r + i;

ans /= i;

}**return** ans;}

**Tobinary**

**int** tobinary(bitset<8>x)

{

string mystring=x.to\_string<**char**,std::string::traits\_type,std::string::allocator\_type>(); **return** atoi(mystring.c\_str());

}

**Minimum Excludant (MEX)**

**int** calculateMex(unordered\_set<**int**> Set)

{

**int** Mex = 0;

**while** (Set.find(Mex) != Set.end()) Mex++;

**return** (Mex);

}

**Modulo Arithmetic**

/\* Iterative Function to calculate (x^n)%p in O(logy) \*/

**int fast\_pow** (**int** x, **unsigned** **int** y, **int** p)

{ **int** res = 1;

x = x % p;

**while** (y > 0) {

**if** (y & 1)

res = (res\*x) % p;

y = y>>1;

x = (x\*x) % p;

} **return** res;}

**int** **findMMI\_fermat**(**int** n,**int** M)

{ **return** fast\_pow(n, M-2, M); }

**O(n log n) algorithm for The Longest Increasing Subseq**

// Binary search (note boundaries in the caller)

**int** CeilIndex(std::vector<**int**> &v, **int** l, **int** r, **int** key) {

**while** (r-l > 1) {

**int** m = l + (r-l)/2;

**if** (v[m] >= key) r = m;

**else** l = m;

} **return** r; }

**int** LongestIncreasingSubsequenceLength(std::vector<**int**> &v) {

**if** (v.size() == 0)

**return** 0;

std::vector<**int**> tail(v.size(), 0);

**int** length = 1; // always points empty slot in tail

tail[0] = v[0];

**for** (size\_t i = 1; i < v.size(); i++) {

**if** (v[i] < tail[0]) tail[0] = v[i];

**else** **if** (v[i] > tail[length-1]) tail[length++] = v[i];

**else** tail[CeilIndex(tail, -1, length-1, v[i])] = v[i];

}**return** length; }

**Manacher’s Algo O(2n)**

/// For example, S = "abba", T = "^#a#b#b#a#$".

/string preProcess(string s) {

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

**if** (n == 0) **return** "^$";

string ret = "^";

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

ret += "#" + s.substr(i, 1);

ret += "#$";

**return** ret; }

string longestPalindrome(string s){

string T = preProcess(s);

**int** n = T.length();

**int** \*P = **new** **int**[n];

**int** C = 0, R = 0;

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

**int** i\_mirror = 2\*C-i; // equals to i' = C - (i-C)

P[i] = (R > i) ? min(R-i, P[i\_mirror]) : 0;

// Attempt to expand palindrome centered at i

**while** (T[i + 1 + P[i]] == T[i - 1 - P[i]])

P[i]++;

**if** (i + P[i] > R) {

C = i;

R = i + P[i];

} }

**int** maxLen = 0; // Find the maximum element in P.

**int** centerIndex = 0;

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

**if** (P[i] > maxLen) {

maxLen = P[i];

centerIndex = i;

}}

**delete**[] P;

**return** s.substr((centerIndex - 1 - maxLen)/2, maxLen); }

**KMP O(n)**

**#define** MAX\_N 100010

**char** T[MAX\_N], P[MAX\_N]; // T = text, P = pattern

**int** b[MAX\_N], n, m; // b = back table, n = length of T, m = length of P

**void** **kmpPreprocess**() { // call this before calling kmpSearch()

**int** i = 0, j = -1; b[0] = -1; // starting values

**while** (i < m) { // pre-process the pattern string P

**while** (j >= 0 && P[i] != P[j]) j = b[j]; // if different, reset j using b

i++; j++; // if same, advance both pointers

b[i] = j; // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2, 3, 4

} } // in the example of P = "SEVENTY SEVEN" above

**void** **kmpSearch**() { // this is similar as kmpPreprocess(), but on string T

**int** i = 0, j = 0; // starting values

**while** (i < n) { // search through string T

**while** (j >= 0 && T[i] != P[j]) j = b[j]; // if different, reset j using b

i++; j++; // if same, advance both pointers

**if** (j == m) { // a match found when j == m

printf("P is found at index %d in T\n", i - j);

j = b[j]; // prepare j for the next possible match

} } }

// Usage:

strcpy(T, "I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN");

strcpy(P, "SEVENTY SEVEN");

n = (**int**)strlen(T);

m = (**int**)strlen(P);

kmpPreprocess();

kmpSearch();

**Prime Sieve of Eratosthenes O(n) - Segmented**

//this algorithm calculates an array lp[], which allows us to find factorization of any number in the segment [2;n] in the time of the size order of this factorization. Moreover, using just one extra array will allow us to avoid divisions when looking for factorization.

**const** **int** N = 10000000;

**int** lp[N+1];

vector<**int**> pr;

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

**if** (lp[i] == 0) {

lp[i] = i;

pr.push\_back (i);

}

**for** (**int** j=0; j<(**int**)pr.size() && pr[j]<=lp[i] && i\*pr[j]<=N; ++j)

lp[i \* pr[j]] = pr[j];

}

**String Trie**

**#define** ARRAY\_SIZE(a) **sizeof**(a)/**sizeof**(a[0])

**#define** ALPHABET\_SIZE (26)

**#define** CHAR\_TO\_INDEX(c) ((**int**)c - (**int**)'a')

**struct** TrieNode{

**struct** TrieNode \*children[ALPHABET\_SIZE];

**bool** isLeaf;

};

// Returns new trie node (initialized to NULLs)

**struct** TrieNode \***getNode**(**void**){

**struct** TrieNode \*pNode = NULL;

pNode = (**struct** TrieNode \*)**malloc**(**sizeof**(**struct** TrieNode));

**if** (pNode)

{

**int** i;

pNode->isLeaf = **false**;

**for** (i = 0; i < ALPHABET\_SIZE; i++)

pNode->children[i] = NULL;

} **return** pNode; }

// If not present, inserts key into trie

// If the key is prefix of trie node, just marks leaf node

**void** **insert**(**struct** TrieNode \*root, **const** **char** \*key){

**int** level;

**int** length = strlen(key);

**int** index;

**struct** TrieNode \*pCrawl = root;

**for** (level = 0; level < length; level++)

{

index = CHAR\_TO\_INDEX(key[level]);

**if** (!pCrawl->children[index])

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];

}

// mark last node as leaf

pCrawl->isLeaf = **true**;

}

// Returns true if key presents in trie, else false

**bool** **search**(**struct** TrieNode \*root, **const** **char** \*key){

**int** level;

**int** length = strlen(key);

**int** index;

**struct** TrieNode \*pCrawl = root;

**for** (level = 0; level < length; level++){

index = CHAR\_TO\_INDEX(key[level]);

**if** (!pCrawl->children[index])

**return** **false**;

pCrawl = pCrawl->children[index];

}

**return** (pCrawl != NULL && pCrawl->isLeaf); }

// Usage:// Input keys (use only 'a' through 'z' and lower case)

**char** keys[][8] = {"the", "a", "there", "answer", "any","by", "bye", "their"};

**char** output[][32] = {"Not present in trie", "Present in trie"};

**struct** TrieNode \*root = getNode();

**int** i; // Construct trie

**for** (i = 0; i < ARRAY\_SIZE(keys); i++)

insert(root, keys[i]);

printf("%s --- %s\n", "these", output[search(root, "these")] );

**Suffix Array**

// Suffix array construction in O(L log2 L) time. Routine for

// computing the length of the longest common prefix of any two

// suffixes in O(log L) time.

// INPUT: string s

// OUTPUT: array suffix[] such that suffix[i] = index (from 0 to L-1)

// of substring s[i...L-1] in the list of sorted suffixes.

// That is, if we take the inverse of the permutation suffix[],

// we get the actual suffix array.

**struct** SuffixArray {

**const** **int** L;

string s;

vector<vector<**int**> > P;

vector<pair<pair<**int**,**int**>,**int**> > M;

**SuffixArray**(**const** string &s) : L(s.length()), s(s), P(1, vector<**int**>(L, 0)), M(L) {

**for** (**int** i = 0; i < L; i++) P[0][i] = **int**(s[i]);

**for** (**int** skip = 1, level = 1; skip < L; skip \*= 2, level++) {

P.push\_back(vector<**int**>(L, 0));

**for** (**int** i = 0; i < L; i++)

M[i] = make\_pair(make\_pair(P[level-1][i], i + skip < L ? P[level-1][i + skip] : -1000), i);

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

**for** (**int** i = 0; i < L; i++)

P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[level][M[i-1].second] : i;

}}

vector<**int**> **GetSuffixArray**() { **return** P.back(); }

// returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]

**int** **LongestCommonPrefix**(**int** i, **int** j) {

**int** len = 0;

**if** (i == j) **return** L - i;

**for** (**int** k = P.size() - 1; k >= 0 && i < L && j < L; k--) {

**if** (P[k][i] == P[k][j]) {

i += 1 << k;

j += 1 << k;

len += 1 << k;

}}

**return** len; } };

**int** **main**() {

**int** T; cin >> T;

**for** (**int** caseno = 0; caseno < T; caseno++) {

string s;

cin >> s;

SuffixArray array(s);

vector<**int**> v = array.GetSuffixArray();

**int** bestlen = -1, bestpos = -1, bestcount = 0;

**for** (**int** i = 0; i < s.length(); i++) {

**int** len = 0, count = 0;

**for** (**int** j = i+1; j < s.length(); j++) {

**int** l = array.LongestCommonPrefix(i, j);

**if** (l >= len) {

**if** (l > len) count = 2; **else** count++;

len = l;

}}

**if** (len > bestlen || len == bestlen && s.substr(bestpos, bestlen) > s.substr(i, len)) {

bestlen = len;

bestcount = count;

bestpos = i;

}}

**if** (bestlen == 0) cout << "No reps found!";

**else** {

cout << s.substr(bestpos, bestlen) << " " << bestcount ;

}}}

**Dijkstra’s Algo O(V2+E)**

#define INF 1000000000

**int** main() {

**int** V, E, s, u, v, w;

vector<vii> AdjList;

scanf("%d %d %d", &V, &E, &s);

AdjList.assign(V, vii());

**for** (**int** i = 0; i < E; i++) {

scanf("%d %d %d", &u, &v, &w);

AdjList[u].push\_back(ii(v, w)); }

// Dijkstra routine

vi dist(V, INF); dist[s] = 0; // INF = 1B to avoid overflow

priority\_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));

**while** (!pq.empty()) { // main loop

ii front = pq.top(); pq.pop(); // greedy: pick shortest unvisited vertex

**int** d = front.first, u = front.second;

**if** (d > dist[u]) **continue**; // this check is important, see the explanation

**for** (**int** j = 0; j < (**int**)AdjList[u].size(); j++) {

ii v = AdjList[u][j]; // all outgoing edges from u

**if** (dist[u] + v.second < dist[v.first]) {

dist[v.first] = dist[u] + v.second; // relax operation

pq.push(ii(dist[v.first], v.first));

} } } // note: this variant can cause duplicate items in the priority queue

**for** (**int** i = 0; i < V; i++) // index + 1 for final answer

printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

**return** 0;}

**Prim & Kruskal MST O(ElogV)**

// UFDS written using both path compression and union by rank heuristics

**class** UnionFind { // OOP style

**private**:

vi p, rank, setSize;

**int** numSets;

**public**:

**UnionFind**(**int** N) {

setSize.assign(N, 1); numSets = N; rank.assign(N, 0);

p.assign(N, 0); **for** (**int** i = 0; i < N; i++) p[i] = i; }

**int** **findSet**(**int** i) { **return** (p[i] == i) ? i : (p[i] = findSet(p[i])); }

**bool** **isSameSet**(**int** i, **int** j) { **return** findSet(i) == findSet(j); }

**void** **unionSet**(**int** i, **int** j) {

**if** (!isSameSet(i, j)) { numSets--;

**int** x = findSet(i), y = findSet(j);

// rank is used to keep the tree short

**if** (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }

**else** { p[x] = y; setSize[y] += setSize[x];

**if** (rank[x] == rank[y]) rank[y]++; } } }

**int** **numDisjointSets**() { **return** numSets; }

**int** **sizeOfSet**(**int** i) { **return** setSize[findSet(i)]; }

};

vector<vii> AdjList;

vi taken; // global boolean flag to avoid cycle

priority\_queue<ii> pq; // priority queue to help choose shorter edges

**void** **process**(**int** vtx) { // so, we use -ve sign to reverse the sort order

taken[vtx] = 1;

**for** (**int** j = 0; j < (**int**)AdjList[vtx].size(); j++) {

ii v = AdjList[vtx][j];

**if** (!taken[v.first]) pq.push(ii(-v.second, -v.first));

} } // sort by (inc) weight then by (inc) id

**int** **main**() {

**int** V, E, u, v, w;

scanf("%d %d", &V, &E);

// Kruskal's algorithm merged with Prim's algorithm

AdjList.assign(V, vii());

vector< pair<**int**, ii> > EdgeList; // (weight, two vertices) of the edge

**for** (**int** i = 0; i < E; i++) {

scanf("%d %d %d", &u, &v, &w); // read the triple: (u, v, w)

EdgeList.push\_back(make\_pair(w, ii(u, v))); // (w, u, v)

AdjList[u].push\_back(ii(v, w));

AdjList[v].push\_back(ii(u, w));

}

sort(EdgeList.begin(), EdgeList.end()); // sort by edge weight O(E log E)

**int** mst\_cost = 0;

UnionFind UF(V); // all V are disjoint sets initially

**for** (**int** i = 0; i < E; i++) { // for each edge, O(E)

pair<**int**, ii> front = EdgeList[i];

**if** (!UF.isSameSet(front.second.first, front.second.second)) { // check

mst\_cost += front.first; // add the weight of e to MST

UF.unionSet(front.second.first, front.second.second); // link them

} } // note: the runtime cost of UFDS is very light

// note: the number of disjoint sets must eventually be 1 for a valid MST

printf("MST cost = %d (Kruskal's)\n", mst\_cost);

// inside int main() --- assume the graph is stored in AdjList, pq is empty

taken.assign(V, 0); // no vertex is taken at the beginning

process(0); // take vertex 0 and process all edges incident to vertex 0

mst\_cost = 0;

**while** (!pq.empty()) { // repeat until V vertices (E=V-1 edges) are taken

ii front = pq.top(); pq.pop();

u = -front.second, w = -front.first; // negate the id and weight again

**if** (!taken[u]) // we have not connected this vertex yet

mst\_cost += w, process(u); // take u, process all edges incident to u

} // each edge is in pq only once!

printf("MST cost = %d (Prim's)\n", mst\_cost); }

**LCA Offline O(1) Tarjan**

**const** **int** MAXN = …; bool u[MAXN]; //MAXN max vertices

vector<**int**> g[MAXN], q[MAXN];

**int** dsu[MAXN], ancestor[MAXN];

**int** **dsu\_get** (**int** v)

{ **return** v == dsu[v] ? v : dsu[v] = dsu\_get (dsu[v]); }

**void** **dsu\_unite** (**int** a, **int** b, **int** new\_ancestor) {

a = dsu\_get (a), b = dsu\_get (b);

**if** (rand() & 1) swap (a, b);

dsu[a] = b, ancestor[b] = new\_ancestor; }

**void** **dfs** (**int** v) {

dsu[v] = v, ancestor[v] = v;

u[v] = **true**;

**for** (size\_t i=0; i<g[v].size(); ++i)

**if** (!u[g[v][i]]) {

dfs (g[v][i]);

dsu\_unite (v, g[v][i], v);

}

**for** (size\_t i=0; i<q[v].size(); ++i)

**if** (u[q[v][i]]) {

printf ("%d %d -> %d\n", v+1, q[v][i]+1,

ancestor[ dsu\_get(q[v][i]) ]+1); }

**int** **main**() { //... Input Graph ...

**for** (;;) { // Read queries

**int** a, b = ...; --a, --b;

q[a].push\_back (b); q[b].push\_back (a); }

dfs (0); }// Answer queries

**FFT O(nlogn)**

struct Complex {

**double** re, im;

explicit Complex(**double** re = 0, **double** im = 0) : re(re), im(im) {}

Complex operator+(**const** Complex& o) **const** { **return** Complex(re + o.re, im + o.im); }

Complex operator-(**const** Complex& o) **const** { **return** Complex(re - o.re, im - o.im); }

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

**return** Complex(re \* o.re - im \* o.im, re \* o.im + im \* o.re); }

Complex operator/(**const** **double** o) **const** { **return** Complex(re / o, im / o); }

};

ostream& operator<<(ostream& out, **const** Complex& s) {

**return** out << fixed << setprecision(2) << "(" << s.re << ", " << s.im ; }

using Base = Complex;

**const** **int** K = 18; **const** **int** N = 1 << K; **const** **double** PI = acos(-1);

**int** rev[N]; Base roots[N];

**void** **prep**() {

rev[0] = 0;

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

rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (K - 1));

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

**double** angle = 2 \* PI \* i / N;

roots[i] = Base(cos(angle), sin(angle));

}}

Base a[N]; Base b[N];

**void** **fft**(Base\* a, const int k) {

**const** **int** n = 1 << k;

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

**int** j = rev[i] >> (K - k);

**if** (i > j) swap(a[i], a[j]);

}

**for** (**int** len = 2; len <= n; len <<= 1) {

**int** half = len / 2;

**int** nlen = N / len;

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

**for** (**int** j = 0; j < half; ++j) {

Base f = a[i + j];

Base s = a[i + j + half] \* roots[nlen \* j];

a[i + j] = f + s;

a[i + j + half] = f - s;

} } } }

**void** **mult**(**const** vector<**int**>& a, **const** vector<**int**>& b, vector<**int**>& res){

**int** sz = a.size() + b.size() - 1;

**int** k = 0;

**while** (sz > (1 << k)) ++k;

**const** **int** n = 1 << k;

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

::a[i].re = (i < **int**(a.size()) ? a[i] : 0);

::a[i].im = (i < **int**(b.size()) ? b[i] : 0);

}

fft(::a, k);

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

**int** j = n - i;

**if** (i == 0) j = 0;

Base x = ::a[i]; Base y = ::a[j]; Base f = (x + y) / 2;

x = x - f; y = y - f;

Base bi(f.im, -x.re); Base bj(f.im, -y.re);

::a[i] = Base(f.re, x.im) \* bi; ::a[j] = Base(f.re, y.im) \* bj;

}

fft(::a, k);

reverse(::a + 1, ::a + n);

res.resize(sz);

**for** (**int** i = 0; i < sz; ++i) { res[i] = ::a[i].re / n + 0.5; }

}

**void** **mult\_st**(**const** vector<**int**>& a, **const** vector<**int**>& b, vector<**int**>& res) {

**int** sz = a.size() + b.size() - 1;

res.assign(sz, 0);

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

**for** (**int** j = 0; j < **int**(b.size()); ++j) { res[i + j] += a[i] \* b[j]; }

}}

**const** **int** C = 2;

**const** **int** P = 100;

**int** **to\_int**(**const** string& s, **int** pos, **int** len) {

**int** res = 0;

**for** (**int** i = pos + len - 1; i >= pos; --i)

res = res \* 10 + (s[i] - '0');

**return** res; }

**void** **to\_polinom**(string& s, vector<**int**>& res) {

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

reverse(s.begin(), s.end());

res.resize((n + C - 1) / C);

**int** idx = 0;

**for** (**int** i = 0; i < n; i += C, idx++)

res[idx] = to\_int(s, i, min(C, n - i)); }

**void** **to\_string**(**int** x, string& s) {

**for** (**int** i = 0; i < C; ++i) {

s += **char**('0' + x % 10);

x /= 10; }}

**void** **to\_string**(**const** vector<**int**>& v, string& res) {

res.clear();

**int** carry = 0;

**for** (**int** x : v) {

**int** cur = x + carry;

to\_string(cur % P, res);

carry = cur / P;

}

**if** (carry) to\_string(carry, res);

**while** (res.size() > 1 && res.back() == '0')

res.pop\_back();

reverse(res.begin(), res.end());

}

**char** buf[N];

**int** main() {

prep(); **int** t; scanf("%d", &t);

vector<**int**> a, b, c; string an; string bn; string cn;

**for** (**int** i = 0; i < t; ++i) {

scanf("%s", buf); an = buf; to\_polinom(an, a);

scanf("%s", buf); bn = buf; to\_polinom(bn, b);

mult(a, b, c); to\_string(c, cn); printf("%s\n", cn.c\_str()); }}

**Dinics O(V2E)**

**const** **int** MAXN = ...;

**const** **int** INF = 1000000000;

struct edge { **int** a, b, cap, flow; };

**int** n, s, t, d[MAXN], ptr[MAXN], q[MAXN];

vector<edge> e;

vector<**int**> g[MAXN];

**void** **add\_edge** (**int** a, **int** b, **int** cap) {

edge e1 = { a, b, cap, 0 };

edge e2 = { b, a, 0, 0 };

g[a].push\_back ((**int**) e.size());

e.push\_back (e1);

g[b].push\_back ((**int**) e.size());

e.push\_back (e2);

}

**bool** **bfs**() {

**int** qh=0, qt=0;

q[qt++] = s;

memset (d, -1, n \* sizeof d[0]);

d[s] = 0;

**while** (qh < qt && d[t] == -1) {

**int** v = q[qh++];

**for** (size\_t i=0; i<g[v].size(); ++i) {

**int** id = g[v][i],

to = e[id].b;

**if** (d[to] == -1 && e[id].flow < e[id].cap) {

q[qt++] = to;

d[to] = d[v] + 1;

} } }

**return** d[t] != -1; }

**int** **dfs** (**int** v, **int** flow) {

**if** (!flow) **return** 0;

**if** (v == t) **return** flow;

**for** (; ptr[v]<(**int**)g[v].size(); ++ptr[v]) {

**int** id = g[v][ptr[v]],

to = e[id].b;

**if** (d[to] != d[v] + 1) **continue**;

**int** pushed = dfs (to, min (flow, e[id].cap - e[id].flow));

**if** (pushed) {

e[id].flow += pushed;

e[id^1].flow -= pushed;

**return** pushed;

} } **return** 0; }

**int** **dinic**() {

**int** flow = 0;

**for** (;;) {

**if** (!bfs()) **break**;

memset (ptr, 0, n \* sizeof ptr[0]);

**while** (**int** pushed = dfs (s, INF))

flow += pushed;

} **return** flow; }

**Topological Sort O(V+E)**

**int** n; vector<**int**> g[MAXN]; bool used[MAXN]; vector<**int**> ans;

**void** **dfs** (**int** v) {

used[v] = **true**;

**for** (size\_t i=0; i<g[v].size(); ++i) {

**int** to = g[v][i];

**if** (!used[to])

dfs (to);

}ans.push\_back (v); }

**void** **topological\_sort**() {

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

used[i] = **false**;

ans.clear();

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

**if** (!used[i])

dfs (i);

reverse (ans.begin(), ans.end()); }

**Bellman Ford O(VE)**

**#define** INF 1000000000

**int** main() {

**int** V, E, s, a, b, w;

vector<vii> AdjList;

scanf("%d %d %d", &V, &E, &s);

AdjList.assign(V, vii());

**for** (**int** i = 0; i < E; i++) {

scanf("%d %d %d", &a, &b, &w);

AdjList[a].push\_back(ii(b, w));

}

// Bellman Ford routine

vi dist(V, INF); dist[s] = 0;

**for** (**int** i = 0; i < V - 1; i++) // relax all E edges V-1 times, overall O(VE)

**for** (**int** u = 0; u < V; u++) // these two loops = O(E)

**for** (**int** j = 0; j < (**int**)AdjList[u].size(); j++) {

ii v = AdjList[u][j]; // we can record SP spanning here if needed

dist[v.first] = min(dist[v.first], dist[u] + v.second); // relax

}

bool hasNegativeCycle = **false**;

**for** (**int** u = 0; u < V; u++) // one more pass to check

**for** (**int** j = 0; j < (**int**)AdjList[u].size(); j++) {

ii v = AdjList[u][j];

**if** (dist[v.first] > dist[u] + v.second) // should be false

hasNegativeCycle = **true**; // but if true, then negative cycle exists!

}

printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");

**if** (!hasNegativeCycle)

**for** (**int** i = 0; i < V; i++)

printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

**return** 0; }

**Floyd Warshall O(V3)**

**void** floydWarshall ()

{

/\* d[][] will be the output matrix that will finally have the shortest

distances between every pair of vertices \*/

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

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

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

**if** ( d [i][k] < INF && d [k][j] < INF )

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

**MCBM O(V2+VE)**

vector<vi> AdjList;

vi match, vis; // global variables

**int** **Aug**(**int** l) { // return 1 if an augmenting path is found

**if** (vis[l]) **return** 0; // return 0 otherwise

vis[l] = 1;

**for** (**int** j = 0; j < (**int**)AdjList[l].size(); j++) {

**int** r = AdjList[l][j];

**if** (match[r] == -1 || Aug(match[r])) {

match[r] = l; **return** 1; // found 1 matching

} }

**return** 0; } //no matching

bool **isprime**(**int** v) {

**int** primes[10] = {2,3,5,7,11,13,17,19,23,29};

**for** (**int** i = 0; i < 10; i++)

**if** (primes[i] == v)

**return** **true**;

**return** **false**; }

**int** **main**() {

// build bipartite graph with directed edge from left to right set

// For bipartite graph in Figure 4.44, V = 5, Vleft = 3 (vertex 0 unused)

// AdjList[0] = {} // dummy vertex, but you can choose to use this vertex

// AdjList[1] = {3, 4}

// AdjList[2] = {3}

// AdjList[3] = {} // we use directed edges from left to right set only

// AdjList[4] = {}

**int** V = 5, Vleft = 3; // we ignore vertex 0

AdjList.assign(V, vi());

AdjList[1].push\_back(3); AdjList[1].push\_back(4);

AdjList[2].push\_back(3);

**int** MCBM = 0;

match.assign(V, -1); // V is the number of vertices in bipartite graph

**for** (**int** l = 0; l < Vleft; l++) { // Vleft = size of the left set

vis.assign(Vleft, 0); // reset before each recursion

MCBM += Aug(l);

} printf("Found %d matchings\n", MCBM);

**return** 0; }

**Hopcroft Karp O(E√V)**

#define NIL 0

#define INF INT\_MAX

**class** **BipGraph**{

**int** m, n; // m & n are number of vertices on left and right sides of Bipartite Graph

// adj[u] stores adjacents of left side vertex 'u'. The value of u ranges from 1 to m. 0 is used for dummy vertex

list<**int**> \*adj;

**int** \*pairU, \*pairV, \*dist;

**public**:

BipGraph(**int** m, **int** n); // Constructor

**void** addEdge(**int** u, **int** v); // To add edge

bool bfs();// Returns true if there is an augmenting path

bool dfs(**int** u);// Adds augmenting path if there is one beginning with u

**int** hopcroftKarp();// Returns size of maximum matcing

};

**int** **BipGraph::hopcroftKarp**(){

pairU = **new** **int**[m+1];

pairV = **new** **int**[n+1];

dist = **new** **int**[m+1];

**for** (**int** u=0; u<m; u++) // Or mayb <=

pairU[u] = NIL;

**for** (**int** v=0; v<n; v++) // Or mayb <=

pairV[v] = NIL;

**int** result = 0;

**while** (bfs()){

**for** (**int** u=1; u<=m; u++)

**if** (pairU[u]==NIL && dfs(u))

result++;

}

**return** result; }

bool **BipGraph::bfs**(){

queue<**int**> Q;

**for** (**int** u=1; u<=m; u++){

**if** (pairU[u]==NIL)

{

dist[u] = 0;

Q.push(u);

}

**else** dist[u] = INF;

}

dist[NIL] = INF;

**while** (!Q.empty())

{

**int** u = Q.front();

Q.pop();

**if** (dist[u] < dist[NIL])

{

list<**int**>::iterator i;

**for** (i=adj[u].begin(); i!=adj[u].end(); ++i)

{

**int** v = \*i;

**if** (dist[pairV[v]] == INF){

dist[pairV[v]] = dist[u] + 1;

Q.push(pairV[v]);

} } } }

**return** (dist[NIL] != INF); }

bool **BipGraph::dfs**(**int** u){

**if** (u != NIL){

list<**int**>::iterator i;

**for** (i=adj[u].begin(); i!=adj[u].end(); ++i){

**int** v = \*i;

**if** (dist[pairV[v]] == dist[u]+1){

**if** (dfs(pairV[v]) == **true**){

pairV[v] = u;

pairU[u] = v;

**return** **true**;

} } }

dist[u] = INF;

**return** **false**;

}

**return** **true**; }

**BipGraph::BipGraph**(**int** m, **int** n){

**this**->m = m;

**this**->n = n;

adj = **new** list<**int**>[m+1];

}

**void** **BipGraph::addEdge**(**int** u, **int** v){

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

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

}

**int** main(){

BipGraph g(4, 4);

g.addEdge(1, 2); g.addEdge(1, 3); g.addEdge(2, 1); g.addEdge(3, 2); g.addEdge(4, 2); g.addEdge(4, 4);

cout << "Size of maximum matching is " << g.hopcroftKarp();

**return** 0; }

**Detecting Bridge & Cut-point O(V+E)**

**const** **int** MAXN = ...;

vector<**int**> g[MAXN]; bool used[MAXN];

**int** timer, tin[MAXN], fup[MAXN];

**void** **dfs** (**int** v, **int** p = -1) {

used[v] = **true**;

tin[v] = fup[v] = timer++;

**int** children=0;

**for** (size\_t i = 0; i < g[v].size(); ++i) {

**int** to = g[v][i];

**if** (to == p) **continue**;

**if** (used[to]) fup[v] = min (fup[v], tin[to]);

**else** {

dfs (to, v);

fup[v] = min (fup[v], fup[to]);

//if (fup[to] > tin[v]) Bridge dfs if()

// IS\_BRIDGE(v,to); //Check if bridge is not multiple-edge

**if** (fup[to] >= tin[v] && p!=-1) //CutPoint dfs if()

IS\_CUTPOINT(v); //print Cutpoint, can be called multiple times for same vertex!

++children;

}}

**if**(p == -1 && children > 1)

IS\_CUTPOINT(v);

}

**void** **find\_bridges**() {

timer = 0;

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

used[i] = **false**;

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

**if** (!used[i])

dfs (i); }

**void** **find\_cutpoints**() {

timer = 0;

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

used[i] = **false**;

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

**if** (!used[i])

dfs (i); }

**Min-Cost Max Flow O(V3E)**

**const** **int** INF = 1000000000;

struct rib {

**int** b, u, c, f;

size\_t back;

};};

**void** **add\_rib** (vector <vector <rib>> & g, **int** a, **int** b, **int** u, **int** c) {

rib r1 = {b, u, c, 0, g [b] .size ()};

rib r2 = {a, 0, -c, 0, g [a] .size ()};

g [a] .push\_back (r1);

g [b] .push\_back (r2); }

**int** **main** (){

**int** n, m, k;

vector <vector <rib>> g (n);

**int** s, t;

//... read the graph ...

**int** flow = 0, cost = 0;

**while** (flow <k) {

vector <**int**> id (n, 0);

vector <**int**> d (n, INF);

vector <**int**> q (n);

vector <**int**> p (n);

vector <size\_t> p\_rib (n);

**int** qh = 0, qt = 0;

q [qt ++] = s;

d [s] = 0;

**while** (qh! = qt) {

**int** v = q [qh ++];

id [v] = 2;

**if** (qh == n) qh = 0;

**for** (size\_t i = 0; i <g [v] .size (); ++ i) {

rib & r = g [v] [i];

**if** (rf <ru && d [v] + rc <d [rb]) {

d [rb] = d [v] + rc;

**if** (id [rb] == 0) {

q [qt ++] = rb;

**if** (qt == n) qt = 0;

}

**else** **if** (id [rb] == 2) {

**if** (--qh == -1) qh = n-1;

q [qh] = rb;

}

id [rb] = 1;

p [rb] = v;

p\_rib [rb] = i;

} } }

**if** (d [t] == ​​INF) **break**;

**int** addflow = k - flow;

**for** (**int** v = t; v! = s; v = p [v]) {

**int** pv = p [v]; size\_t pr = p\_rib [v];

addflow = min (addflow, g [pv] [pr] .u - g [pv] [pr] .f); }

**for** (**int** v = t; v! = s; v = p [v]) {

**int** pv = p [v]; size\_t pr = p\_rib [v], r = g [pv] [pr] .back;

g [pv] [pr] .f + = addflow;

g [v] [r] .f - = addflow;

cost + = g [pv] [pr] .c \* addflow;

}

flow + = addflow; }

//... output the result ...}

**Z-Function O(n)**

vector<**int**> **z\_function**(string s) {

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

vector<**int**> z(n);

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

**if** (i <= r)

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

**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; }

**Segment Tree**

**class** SegmentTree { // the segment tree is stored like a heap array

**private**: vi st, A; // recall that vi is: typedef vector<int> vi;

**int** n;

**int** **left** (**int** p) { **return** p << 1; } // same as binary heap operations

**int** **right**(**int** p) { **return** (p << 1) + 1; }

**void** **build**(**int** p, **int** L, **int** R) { // O(n log n)

**if** (L == R) // as L == R, either one is fine

st[p] = L; // store the index

**else** { // recursively compute the values

build(left(p) , L , (L + R) / 2);

build(right(p), (L + R) / 2 + 1, R );

**int** p1 = st[left(p)], p2 = st[right(p)];

st[p] = (A[p1] <= A[p2]) ? p1 : p2;

} }

**int** **rmq**(**int** p, **int** L, **int** R, **int** i, **int** j) { // O(log n)

**if** (i > R || j < L) **return** -1; // current segment outside query range

**if** (L >= i && R <= j) **return** st[p]; // inside query range

// compute the min position in the left and right part of the interval

**int** p1 = rmq(left(p) , L , (L+R) / 2, i, j);

**int** p2 = rmq(right(p), (L+R) / 2 + 1, R , i, j);

**if** (p1 == -1) **return** p2; // if we try to access segment outside query

**if** (p2 == -1) **return** p1; // same as above

**return** (A[p1] <= A[p2]) ? p1 : p2; } // as as in build routine

**int** **update\_point**(**int** p, **int** L, **int** R, **int** idx, **int** new\_value) {

**int** i = idx, j = idx; // if curr interval doesn’t intersect update interval, return this st node value!

**if** (i > R || j < L)

**return** st[p];

// if current interval is included in the update range, update that st[node]

**if** (L == i && R == j) {

A[i] = new\_value; // update the underlying array

**return** st[p] = L; // this index

}

**int** p1, p2; // compute min pition in left and right part of the interval

p1 = update\_point(left(p) , L , (L + R) / 2, idx, new\_value);

p2 = update\_point(right(p), (L + R) / 2 + 1, R , idx, new\_value);

// return the pition where the overall minimum is

**return** st[p] = (A[p1] <= A[p2]) ? p1 : p2; }

**public**:

**SegmentTree**(**const** vi &\_A) {

A = \_A; n = (**int**)A.size(); // copy content for local usage

st.assign(4 \* n, 0); // create large enough vector of zeroes

build(1, 0, n - 1); // recursive build

}

**int** **rmq**(**int** i, **int** j) { **return** rmq(1, 0, n - 1, i, j); } // overloading

**int** **update\_point**(**int** idx, **int** new\_value) {

**return** update\_point(1, 0, n - 1, idx, new\_value); }

};

**Persistence Segment Tree**

**struct** Vertex {

Vertex \*l, \*r;

**int** sum;

Vertex(**int** val) : l(nullptr), r(nullptr), sum(0) {}

Vertex(Vertex \*l, Vertex \*r) : l(l), r(r), sum(0) {

**if** (l) sum += l->sum;

**if** (r) sum += r->sum; }

};

Vertex\* **build**(**int** a[], **int** tl, **int** tr) {

**if** (tl == tr)

**return** **new** Vertex(a[tl]);

**int** tmx = (tlx + trx) / 2;

**return** **new** Vertex(build(a, tl, tm), build(a, tm+1, tr));

}

**int** **get\_sum**(Vertex\* v, **int** tl, **int** tr, **int** l, **int** r) {

**if** (l > r)

**return** 0;

**if** (l == tl && tr == r)

**return** v->sum;

**int** tm = (tl + tr) / 2;

**return** getsum(t->l, tl, tm, l, min(r, tm))

+ getsum(t->r, tm+1, tr, max(l, tm+1), r); }

Vertex\* **update**(Vertex\* v, **int** tl, **int** tr, **int** pos, **int** new\_val) {

**if** (tl == tr)

**return** **new** Vertex(new\_val);

**int** tm = (tl + tr) / 2;

**if** (pos <= tm)

**return** **new** Vertex(update(t->l, tl, tm, pos, new\_val), v->r);

**else**

**return** **new** Vertex(v->l, update(t->r, tm+1, tr, pos, new\_val)); }

**Expected Value**

**float** calc\_Expectation(**float** a[], **float** n){

**float** prb = (1 / n);

**float** sum = 0;

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

sum += a[i] \* prb;

**return** sum; }

**Roman Numerals**

**void** **AtoR**(**int** A) {

map<**int**, string> cvt;

cvt[1000] = "M"; cvt[900] = "CM"; cvt[500] = "D"; cvt[400] = "CD";

cvt[100] = "C"; cvt[90] = "XC"; cvt[50] = "L"; cvt[40] = "XL";

cvt[10] = "X"; cvt[9] = "IX"; cvt[5] = "V"; cvt[4] = "IV";

cvt[1] = "I";

// process from larger values to smaller values

**for** (map<**int**, string>::reverse\_iterator i = cvt.rbegin();

i != cvt.rend(); i++)

**while** (A >= i->first) {

printf("%s", ((string)i->second).c\_str());

A -= i->first; } printf("\n"); }

**void** **RtoA**(**char** R[]) {

map<**char**, **int**> RtoA;

RtoA['I'] = 1; RtoA['V'] = 5; RtoA['X'] = 10; RtoA['L'] = 50;

RtoA['C'] = 100; RtoA['D'] = 500; RtoA['M'] = 1000;

**int** value = 0;

**for** (**int** i = 0; R[i]; i++)

**if** (R[i+1] && RtoA[R[i]] < RtoA[R[i+1]]) { // check next char first

value += RtoA[R[i+1]] - RtoA[R[i]]; // by definition

i++; } // skip this char

**else** value += RtoA[R[i]]; printf("%d\n", value); }

**PYTHON**

**Template**

**import** atexit

**import** io

**import** sys

from bisect **import** \*

from collections **import** \*

from fractions **import** gcd

from itertools **import** \*

from math **import** \*

**import** sys

**import** os

inf = **float**('inf')

\_inf = **float**('-inf')

\_INPUT\_LINES = sys.stdin.read().splitlines()

input = iter(\_INPUT\_LINES).\_\_next\_\_

\_OUTPUT\_BUFFER = io.StringIO()

sys.stdout = \_OUTPUT\_BUFFER

@atexit.register

def write():

sys.\_\_stdout\_\_.write(\_OUTPUT\_BUFFER.getvalue())

def main():

# code here

**if** \_\_name\_\_ == '\_\_main\_\_':

main()

**Extended Bezout Diophantine**

**import** math

from fractions **import** gcd

def ixgcd(a, b):

""" gcd(a, b) = a \* x + b \* y is bezout indentiy

find x, y **for** gcd(a, b) iteration (log(min(a, b))) """

x0, x1, y0, y1 = 1, 0, 0, 1

**while** b:

q, a, b = a // b, b, a % b

x0, x1 = x1, x0 - q \* x1

y0, y1 = y1, y0 - q \* y1

**return** a, x0, y0 # gcd, x0, y0

def diophan\_(a, b, c):

d, x0, y0 = ixgcd(a, b)

**if** c % d == 0 and min(a, b) <= c: # check **for** **int** soln

x0 \*= c // d

y0 \*= c // d

**if** x0 >= 0 and y0 >= 0: # **for** any soln

print("YES")

elif x0 < 0 and y0 < 0:

print("NO")

**else**: # **for** positive **int** soln

# **for** min(x+y), **if** a==b, x+y; a<b, min(k); a>b, max(k)

**if** x0 < 0:

k = (d \* x0) / b

elif y0 < 0:

k = (-1 \* d \* y0) / a

**if** k < 0:

k = math.floor(k) # ceil towards 0, floor towards -inf

elif k > 0:

k = math.ceil(k) # ceil towards +inf

x1 = x0 - (k \* b) // d # k E int{1..n} for all int soln

y1 = y0 + (k \* a) // d

**if** x1 < 0 or y1 < 0:

print("NO")

**else**:

print("YES")

**else**:

print("NO")

**Segmented Sieve**

**import** itertools

izip = itertools.zip\_longest

chain = itertools.chain.from\_iterable

compress = itertools.compress

def rwh\_primes\_jason(n):

""" Input n>=6, Returns a list of primes, 2 <= p < n """

zero = bytearray([False])

size = n//3 + (n % 6 == 2)

sieve = bytearray([True]) \* size

sieve[0] = False

**for** i in range(**int**(n\*\*0.5)//3+1):

**if** sieve[i]:

k=3\*i+1|1

start = (k\*k+4\*k-2\*k\*(i&1))//3

sieve[(k\*k)//3::2\*k]=zero\*((size - (k\*k)//3 - 1) // (2 \* k) + 1)

sieve[ start ::2\*k]=zero\*((size - start - 1) // (2 \* k) + 1)

ans = [2,3]

poss = chain(izip(\*[range(i, n, 6) **for** i in (1,5)]))

ans.extend(compress(poss, sieve))

**return** ans

**Factorization**

def prime\_factors(n): # sqrt factorization

start = 2

listing = []

**while** start\*start <= n :

**if**(n%start == 0):

listing.append(start)

**while**(n%start == 0):

n = n // start

start = start + 1

**if** (n != 1):

listing.append(n)

**return** listing

# Make primes list as primes

def phi(n, primes):

totient = n

**for** p in primes:

totient -= totient//p

**return** totient

**Base-Conversion**

def to\_digits(n, b):

"""Convert a positive number n to its digit representation in base b."""

**if** n == 0:

**return** [0]

digits = []

**while** n:

digits.append(n % b)

n //= b

**return** digits[::-1]

def from\_digits(digits, b):

"""Compute the number given by digits in base b."""

n = 0

**for** d in digits:

n = b \* n + d

**return** n

def convert\_base(digits, b, c):

"""Convert the digits representation of a number from base b to base c."""

**return** to\_digits(from\_digits(digits, b), c)

**Number Theory**

# A function f is called "number theoretic" or "multiplicative" **if** f(nm) = f(n) f(m) where n and m are coprime.

# Any number theoretic function can be computed in

# 1) O(sqrt(n)) **for** a certain n.

# 2) O((hi-lo) loglog hi) **for** n in [lo,hi).

# 1)

f(n) = 1;

**for** p = 1 ... sqrt(n):

**if** n % p == 0:

q = 1

**while** (n % k == 0)

q \*= p;

n /= p;

f(n) \*= f(q) // q is a prime power

# Complexity:

# increment of p: O(sqrt(n))

# division by p: O(log n)

# ==> O(sqrt(n)).

# 2)

f(n) = 1 **for** all n in [lo, hi)

r(n) = n **for** all n in [lo, hi)

**for** p in primes:

**for** n in [lo,hi) such that n%p == 0:

**if** r(n) < p: **break**;

q = 1

**while** r(n) % p == 0:

q \*= p

r(n) /= p

f(n) \*= f(q)

# Complexity:

# division by p: sum of exponents in n!, which is known to be O(n log log n)

# ==> O(n log log n).

**Linear Sort**

def counting\_sort(array, maxval):

"""in-place counting sort"""

m = maxval + 1

count = [0] \* m # init with zeros

**for** a in array:

count[a] += 1 # count occurences

i = 0

**for** a in range(m): # emit

**for** c in range(count[a]): # - emit 'count[a]' copies of 'a'

array[i] = a

i += 1

**return** (array,count)

**Modulo Inverse**

def modinverse\_query(m):

# m is prime, modular inverse in range[1, m-1]

r = [1]\*m-1

**for** i in range(2, m):

r[i] = (m - (m//i) \* r[m%i] % m) % m

**return** r

def powmod(a,e,n):

# same as pow(a, b, mod)

accum = 1; i = 0; apow2 = a

**while** ((e>>i)>0):

**if** ((e>>i) & 1):

accum = (accum\*apow2) % n

apow2 = (apow2\*apow2) % n

**return** accum

**Mul\_Mod**

def mulmod(a, b, c):

x = 0

y = a % c

**while**(b > 0):

**if**(b & 1):

x = (x+y)%c

y = (y<<2)%c

b >>= 2

**return** x%c

**Newton Root**

# %These choices depend on the problem being solved

# x0 = 1 %The initial value

# f = @(x) x^2 - 2 %The function whose root we are trying to find

# fprime = @(x) 2\*x %The derivative of f(x)

# tolerance = 10^(-7) %7 digit accuracy is desired

# epsilon = 10^(-14) %Don't want to divide by a number smaller than this

# maxIterations = 20 %Don't allow the iterations to continue indefinitely

# haveWeFoundSolution = **false** %Have not converged to a solution yet

# **for** i = 1 : maxIterations

# y = f(x0)

# yprime = fprime(x0)

# **if**(abs(yprime) < epsilon) %Don't want to divide by too small of a number

# % denominator is too small

# **break**; %Leave the loop

# end

# x1 = x0 - y/yprime %Do Newton's computation

# **if**(abs(x1 - x0) <= tolerance \* abs(x1)) %If the result is within the desired tolerance

# haveWeFoundSolution = **true**

# **break**; %Done, so leave the loop

# end

# x0 = x1 %Update x0 to start the process again

# end

# **if** (haveWeFoundSolution)

# ... % x1 is a solution within tolerance and maximum number of iterations

# **else**

# ... % did not converge

# end

**Probabilistic**

def is\_probable\_prime(n, k = 7):

"""use Rabin-Miller algorithm to return True (n is probably prime)

or False (n is definitely composite)"""

**if** n < 6: # assuming n >= 0 in all cases... shortcut small cases here

**return** [False, False, True, True, False, True][n]

elif n & 1 == 0: # should be faster than n % 2

**return** False

**else**:

s, d = 0, n - 1

**while** d & 1 == 0:

s, d = s + 1, d >> 1

# Use random.randint(2, n-2) **for** very large num, sys.maxsize == inf

**for** a in random.sample(range(2, min(n - 2, sys.maxsize)), min(n - 4, k)):

x = pow(a, d, n)

**if** x != 1 and x + 1 != n:

**for** r in range(1, s):

x = pow(x, 2, n)

**if** x == 1:

**return** False # composite **for** sure

elif x == n - 1:

a = 0 # so we know loop didn't continue to end

**break** # could be strong liar, **try** another a

**if** a:

**return** False # composite **if** we reached end of **this** loop

**return** True # probably prime **if** reached end of outer loop

**Math Formulas**

## TRIANGLE STUFF

* **Sum of the interior angles of a triangle:** 180º
* **Area:** **[0102](http://www.dummies.com/wp-content/uploads/0102.gif)**
* **Hero’s area formula:** **[0103](http://www.dummies.com/wp-content/uploads/0103.gif)**, where a, b, and c are the lengths of the triangle’s sides and **[0104](http://www.dummies.com/wp-content/uploads/0104.gif)** (S is the semiperimeter, half the perimeter)
* **Area of an equilateral triangle:** **[0105](http://www.dummies.com/wp-content/uploads/0105.gif)**, where s is a side of the triangle
* **The Pythagorean Theorem: a2 + b2 + c2**, where a and b are the legs of a right triangle and c is the hypotenuse
* **Common Pythagorean triples (side lengths in right triangles):**
  + 3–4–5
  + 5–12–13
  + 7–24–25
  + 8–15–17
* **Ratios of the sides in special right triangles:**
  + The sides opposite the angles in a 45º–45º–90º triangle are in the ratio of **[0112](http://www.dummies.com/wp-content/uploads/0112.gif)**.
  + The sides opposite the angles in a 30º–60º–90º triangle are in the ratio of **[0114](http://www.dummies.com/wp-content/uploads/0114.gif)**.
* **Altitude-on-Hypotenuse Theorem:** If an altitude is drawn to the hypotenuse of a right triangle, then
  + The two triangles formed are similar to the given triangle and to each other: **[0115](http://www.dummies.com/wp-content/uploads/0115.gif)**
  + h2 = xy
  + h2 = xy and b2 = xc

## POLYGON STUFF

* **Area formulas:**
  + **Parallelogram:** Area = base height
  + **Rectangle:** Area = base × height
  + **Kite or rhombus:** **[0116](http://www.dummies.com/wp-content/uploads/0116.gif)**
  + **Square:** **[0117](http://www.dummies.com/wp-content/uploads/0117.gif)**
  + **Trapezoid:** **[0118](http://www.dummies.com/wp-content/uploads/0118.gif)**
  + **Regular polygon:** **[0119](http://www.dummies.com/wp-content/uploads/0119.gif)**
* **Sum of the interior angles in an n-sided polygon:** SumInterior angles= (n – 2)180º
* **Measure of each interior angle of a regular (or other equiangular) n-sided polygon:**  
  **[0120](http://www.dummies.com/wp-content/uploads/0120.gif)** or **[0121](http://www.dummies.com/wp-content/uploads/0121.gif)** (the supplement of an exterior angle)
* **Sum of the exterior angles (one at each vertex) of any polygon:**SumExterior angles = 360º
* **Measure of each exterior angle of a regular (or other equiangular) n-sided polygon:  
  [0122](http://www.dummies.com/wp-content/uploads/0122.gif)**
* **Number of diagonals that can be drawn in an n-sided polygon:**  
  **[0123](http://www.dummies.com/wp-content/uploads/0123.gif)**

## CIRCLE STUFF

* **Circumference: C = 2πr** or πd, where r is the radius of the circle and d is its diameter
* **Area:** AreaCircle = πr2
* **Arc length:** The length of an arc (part of the circumference) is equal to the circumference of the circle (2πr) times the fraction of the circle represented by the arc.  
  **[0124](http://www.dummies.com/wp-content/uploads/0124.gif)**
* **Sector area:** The area of a sector (a pizza-slice shape cut out of a circle) is equal to the area of the circle (πr2) times the fraction of the circle represented by the sector.  
  **[0125](http://www.dummies.com/wp-content/uploads/0125.gif)**
* **Measure of an angle . . .**
  + **On a circle:** **[0126](http://www.dummies.com/wp-content/uploads/0126.gif)**
  + **Inside a circle:[0127](http://www.dummies.com/wp-content/uploads/0127.gif)**
  + **Outside a circle:[0128](http://www.dummies.com/wp-content/uploads/0128.gif)**
* **Chord-Chord Power Theorem:** When two chords of a circle intersect, the product of the parts of one chord is equal to the product of the parts of the other chord.
* **Tangent-Secant Power Theorem:** When a tangent and a secant of a circle meet at an external point, the measure of the tangent squared is equal to the product of the secant’s external part and its total length.
* **Secant-Secant Power Theorem:** When two secants of a circle meet at an external point, the product of one secant’s external part and its total length is equal to the product of the other secant’s external part and its total length.

## 3-D GEOMETRY STUFF

* **Flat-top objects (prisms and cylinders):**
  + **Volume:** VolFlat-top = areabase × height
  + **Surface area:** SAFlat-top = 2 × areabase + arealateral rectanglesFor a cylinder, the single lateral rectangle that wraps around the cylinder has a length equal to the circumference of the cylinder’s base; its width is the cylinder’s height.
* **Pointy-top objects (pyramids and cones):**
  + **Volume: [0129](http://www.dummies.com/wp-content/uploads/0129.gif)**
  + **Surface area:** SAPointy-top = areabase + arealateral trianglesFor a pyramid, the base of each lateral triangle is a side of the pyramid’s base, and the height of the triangle is the slant height of the pyramid. For a cone, one “triangle” wraps around the cone; its base is equal to the circumference of the cone’s base, and its height is equal to the slant height of the cone.
* **Sphere:**
  + **Volume:** **[0130](http://www.dummies.com/wp-content/uploads/0130.gif)**
  + **Surface area:**SASphere = 4πr2

## COORDINATE GEOMETRY STUFF

* **Slope formula:** Given two points (x1, y1) and (x2, y2), the slope of the line that goes through the points is  
  **[0131](http://www.dummies.com/wp-content/uploads/0131.gif)**  
  It doesn’t matter which point is designated as (x1, y1) and which is designated as (x2, y2).
  + The slopes of parallel lines are equal.
  + The slopes of perpendicular lines are opposite reciprocals of each other.
* **Midpoint formula:** Given a segment with endpoints (x1, y1) and (x2, y2), the coordinates of its midpoint are  
  **[0132](http://www.dummies.com/wp-content/uploads/0132.gif)**  
  It doesn’t matter which point is (x1, y1) and which is(x2, y2).
* **Distance formula:** Given two points (x1, y1) and (x2, y2), the distance between the points is  
  **[0133](http://www.dummies.com/wp-content/uploads/0133.gif)**  
  It doesn’t matter which point is (x1, y1) and which is (x2, y2).
* **Equations of a line:**
  + **Slope-intercept form:** y = mx + b, where m is the slope and b is the y-intercept
  + **Point-slope form:** y – y1 = m(x – x1), where m is the slope and (x1, y1) is a point on the line
  + **Horizontal line:**y = b, where b is the y-intercept
  + **Vertical line:**x = a, where a is the x-intercept
* **Equation of a circle:** (x – h)2 + (y – k)2 = r2, where (h, k) is the center of the circle and r is its radius

// define x, y as real(), imag()

**//Use Complex class from FFT**

typedef complex<double> point;

#define x real()

#define y imag()

// sample program

int main() {

point a = 2;

point b(3, 7);

cout << a << ' ' << b << endl; // (2, 0) (3, 7)

cout << a + b << endl; // (5, 7)

}

Oh goodie! We can use std:cout for debugging! We can also add points as vectors without having to define operator+. Nifty. And apparently, we can overall add points, subtract points, do scalar multiplication **without defining any operator**. Very nifty indeed.

#### **Example**

point a(3, 2), b(2, -7);

// vector addition and subtraction

cout << a + b << endl; // (5,-5)

cout << a - b << endl; // (1,9)

// scalar multiplication

cout << 3.0 \* a << endl; // (9,6)

cout << a / 5.0 << endl; // (0.6,0.4)

#### **Functions using std::complex**

What else can we do with complex numbers? Well, there's a lot that is really easy to code.

1. Vector addition: a + b
2. Scalar multiplication: r \* a
3. Dot product: (conj(a) \* b).x
4. Cross product: (conj(a) \* b).y   
   **notice**: *conj(a) \* b = (ax\*bx + ay\*by) + i (ax\*by — ay\*bx)*
5. Squared distance: norm(a - b)
6. Euclidean distance: abs(a - b)
7. Angle of elevation: arg(b - a)
8. Slope of line (a, b): tan(arg(b - a))
9. Polar to cartesian: polar(r, theta)
10. Cartesian to polar: point(abs(p), arg(p))
11. Rotation about the origin: a \* polar(1.0, theta)
12. Rotation about pivot p: (a-p) \* polar(1.0, theta) + p   
    **UPD**: added more useful functions
13. Angle ABC: abs(remainder(arg(a-b) - arg(c-b), 2.0 \* M\_PI))   
    [remainder](http://www.cplusplus.com/reference/cmath/remainder/) normalizes the angle to be between [-PI, PI]. Thus, we can get the positive non-reflex angle by taking its abs value.
14. Project p onto vector v: v \* dot(p, v) / norm(v);
15. Project p onto line (a, b): a + (b - a) \* dot(p - a, b - a) / norm(b - a)
16. Reflect p across line (a, b): a + conj((p - a) / (b - a)) \* (b - a)
17. Intersection of line (a, b) and (p, q):

point intersection(point a, point b, point p, point q) {

double c1 = cross(p - a, b - a), c2 = cross(q - a, b - a);

return (c1 \* q - c2 \* p) / (c1 - c2); // undefined if parallel

}