Universidad de La Habana

UH++

Team Reference

**Daniel Otero Baguer**

**Ramón Alejandro Reyes Fajardo**

**Leandro Castillo Valdés**

# String Matching

* **Knuth-Morris-Pratt**

**#include**<cstdio>

**#include**<cstring>

**const int**

MaxN = 100000 + 10;

**int** N, M, i, j, fail[MaxN];

**char** text[MaxN], patt[MaxN];

**int** main() {

scanf("%s", text + 1);

scanf("%s", patt + 1);

N = strlen(text + 1);

M = strlen(patt + 1);

j = 0;

**for** (i = 2; i <= M; i++) {

**while** (j > 0 && patt[j+1] != patt[i])

j = fail[j];

fail[i] = (j += patt[j+1] == patt[i]);

}

j = 0;

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

**while** (j > 0 && patt[j+1] != text[i])

j = fail[j];

**if** ((j += patt[j+1] == text[i]) == M)

printf("match = %d\n", i - M + 1);

}

**return** 0;

}

* **Z-function**[Tested: PRETILE (SPOJ)]

**voidZfunction**(**char**\* str, **int**\* z)

{

**int** L = 0, R = 0, n = **strlen**(str);

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

z[i] = 0;

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

{

**if** (i <= R)

z[i] = min(z[i-L], R - i + 1);

**while** (i + z[i] < n && str[ i+z[i] ] == str[ z[i] ])

z[ i ]++;

**if** (i + z[i] - 1 > R)

L = i, R = i + z[i] - 1;

}

}

• **Manacher O(N)** [Tested: PLD (SPOJ), EPALIN(SPOJ)]

**voidmanacher**(**int**\* radius, **char**\* S, **bool** even)

{

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

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

radius[ i ] = 0;

**int** L = 0, R = -1;

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

{

**if** (i <= R)

radius[i] = min(radius[ L + R - i + even ], R - i + even);

**while** (i - radius[i] - even >= 0 && i + radius[i] < n)

**if** (S[i - radius[i] - even] == S[i + radius[i]])

radius[i]++;

**elsebreak**;

**if** (i + radius[i] - 1 > R)

{

L = i - radius[i] + !even;

R = i + radius[i] - 1;

}

}

}

* **Suffix Array + LCP O(Nlog^2N)**

**#include**<cstdio>

**#include**<algorithm>

**#define** OVERALL\_TIME "O(Nlog2^2N)"

**using** std :: sort;

**constint**

MaxN = 100000 + 10;

**int** N;

**int** i, k;

**char** s[MaxN];

**int** ord[MaxN];

**int** pos[MaxN];

**int** buc[MaxN];

**int** lcp[MaxN];

**boolcmp**(**constint**& a, **constint**& b) {

**if** (pos[a] != pos[b])

**return** pos[a] < pos[b];

**if** (a + k < N && b + k < N)

**return** pos[a + k] < pos[b + k];

**return** a > b;

}

**boolcom**(**constint**& a, **const int**& b) {

**if** (a > N) **returnfalse**;

**if** (b > N) **returnfalse**;

**return** s[a] == s[b];

}

**intmain**() {

**scanf**("%d", &N);

**scanf**("%s", &s);

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

ord[i] = i;

pos[i] = s[i];

}

**for** (k = 0; k < N; k = k ? 2 \* k : 1) {

sort(ord, ord + N, cmp);

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

buc[i] = buc[i-1] + cmp(ord[i-1], ord[i]);

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

pos[ord[i]] = buc[i];

}

**for** (i = k = 0; i < N; i++)

**if** (pos[i] != N-1) {

**while** (com(i + k, ord[pos[i]+1] + k))

k++;

lcp[pos[i]] = k;

**if** (k >= 1) k--;

}

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

**printf**("%s (%d)\n", s + ord[i], lcp[i]);

**return** 0;

}

• **Suffix Array + LCP O(NlogN)**[Tested: DISUBSTR (SPOJ)]

**#include**<iostream>

**usingnamespace** std;

**constint** MaxC = 255;

**constint** MaxN = 100000 + 10;

**int** offset, n, i;

**char** text[ MaxN ];

**int** suma[ MaxN ], tmp[ MaxN ];

**int** rank1[ MaxN ], rank2[ MaxN ];

**int** sa[ MaxN ], bucket[ MaxN ], lcp[ MaxN ];

**bool** different( **int** a, **int** b ) {

**if** ( b + offset >= n ) **return true**;

**if** ( rank1[ a ] != rank1[ b ] ) **return true**;

**if** ( rank2[ a ] != rank2[ b ] ) **return true**;

**return false**;

}

**int** main() {

cin>> text;

n = strlen( text );

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

suma[i] = 0;

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

suma[ text[i] ]++;

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

suma[ i+1 ] += suma[ i ];

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

bucket[ i ] = suma[ text[i] ] - 1;

**for** ( offset = 1; offset <= n; offset <<= 1 ) {

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

{

rank1[ i ] = bucket[ i ];

rank2[ i ] = (i + offset < n) ? bucket[ i + offset ] : 0;

}

fill( suma, suma + n, 0 );

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

suma[ rank2[i] ]++;

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

suma[ i+1 ] += suma[ i ];

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

tmp[ --suma[ rank2[i] ] ] = i;

fill( suma, suma + n, 0 );

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

suma[ rank1[i] ]++;

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

suma[ i+1 ] += suma[ i ];

**for** (**int** i = n - 1; i >= 0; i--)

sa[ --suma[ rank1[ tmp[i] ] ] ] = tmp[ i ];

bucket[ sa[0] ] = 0;

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

{

bucket[ sa[i] ] = bucket[ sa[i - 1] ];

**if** ( different( sa[i], sa[i - 1] ) )

bucket[ sa[i] ]++;

}

}

offset = 0;

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

{

**if** ( bucket[i] == n-1 )

{

lcp[ n-1 ] = 0; **continue**;

}

**while** (text[i+offset] == text[sa[ bucket[i] + 1] + offset ])

offset++;

lcp[ bucket[i] ] = offset;

**if** ( offset > 0 ) offset--;

}

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

cout<< lcp[ i ] <<" "<< text + sa[ i ] << endl;

**return** 0;

}

* **Aho-Corasick**  [Tested: Growing Strings (COJ)]

*Recordar el caso:*

Text: “abc”

Patterns: {“abcd”, “c”}

**#include**<cstdio>

**#include**<queue>

**#include**<iostream>

**#include**<vector>

**usingnamespace** std;

**const int**

MaxN = 1000005;

**struct**node

{

**int**pos;

node\* fail;

node\* link;

node\* next[ 26 ];

**node**()

{

pos = -1;

fail = NULL;

link = NULL;

**for** (**int** i = 0; i < 26; i++)

next[ i ] = NULL;

}

};

**int** n;

**char** text[MaxN];

queue<node\*> Q;

node\* root = **new** node();

**voidinsert**(**char**\* patt, **int** index)

{

node\* current = root;

**for** (**int** j = 0; patt[j]; j++)

{

**if** (current->next[patt[j] - 'a'] == NULL)

current->next[patt[j] - 'a'] = **new** node();

current = current->next[patt[j] - 'a'];

}

current->pos = index;

}

**intmain**(){

cin>> n;

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

cin>> text;

insert( text, i );

}

**for** (**int** i = 0; i < 26; i++)

**if** ( root->next[i] )

{

root->next[i]->fail = root;

Q.push( root->next[i] );

} **else** root->next[i] = root;

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

node\* t = Q.front(); Q.pop();

**for** (**int** i = 0; i < 26; i++)

**if** ( t->next[i] )

{

Q.push( t->next[i] );

node\* r = t->fail;

**while** ( !r->next[i] )

r = r->fail;

t->next[i]->fail = r->next[i];

/////multiple matches in the same node/////

**if** ( r->next[i]->pos != -1 )

t->next[i]->link = r->next[i];

**else** t->next[i]->link = r->next[i]->link;

/////multiple matches in the same node/////

}

}

cin>> text;

n = **strlen**( text );

node\* state = root;

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

**while** (state->next[ text[i]-'a' ] == NULL)

state = state->fail;

state = state->next[ text[i]-'a' ];

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

**if** (state->pos != -1)

cout<< state->pos<<" found at "<< i << endl;

**for** (node\* r = state->link; r != NULL; r = r->link)

cout<< r->pos<<" found in position "<< i << endl;

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

}

**return** 0;

}

* **Suffix Automaton O(N)**

**struct**state{

**int**lenght;

state \*link;

state \*next[DICT];

};

**int** size;

state \*init, \*last, \*nlast;

state buff[STATES];

state \***newState**(){ **return**&buff[size++]; }

**voidextend**(**constchar** c){

nlast = newState();

nlast->lenght = last->lenght + 1;

state \*p;

**for** (p = last; p && !p->next[c]; p = p->link)

p->next[c] = nlast;

**if** (!p)

nlast->link = init;

**else**{

state \*q = p->next[c];

**if** (p->lenght + 1 == q->lenght)

nlast->link = q;

**else**{

state \*clone = newState();

\*clone = \*q;

clone->lenght = p->lenght + 1;

**for** (; p && p->next[c] == q; p = p->link)

p->next[c] = clone;

q->link = clone;

nlast->link = clone;

}

}

last = nlast;

}

**voidbuild**(**char** \*str){

init = last = newState();

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

extend(str[i] - 'a');

}

**SUFFIX AUTOMATON – DISTINCT SUBSTRINGS**

**intsolve**(state \*p){

**int** ret = 0, \_new = 0;

**for** (**char** c = 0; c < DICT; ++c)

**if** (p->next[c]){

++\_new;

ret += solve(p->next[c]);

}

**return** \_new ? ret + \_new : 0;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SUFFIX AUTOMATON – LONGEST COMMON SUBTRING**

**voidlcs**(**constchar** \*a, **constchar** \*b){

build(a);

**int** len = 0, best = 0;

state \*p = init;

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

**char** c = b[i]-'a';

**if** (!p->next[c]){

**while** (p && !p->next[c]) p = p->link;

**if** (!p){

len = 0;

p = init;

**continue**;

}

len = p->lenght;

}

++len;

p = p->next[c];

**if** (best < len) best = len;

}

**printf**("%d\n", best);

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SUFFIX AUTOMATON – SHORTEST STRING NOT INCLUDED IN S**

**intsolve**(state \*p){

**int** ret = OO;

**for** (**char** c = 0; c < DICT; ++c)

**if** (p->next[c])

ret = min(ret, solve(p->next[c]));

**elsereturn** 1;

**return** 1 + ret;

}

* **LCS2**

**int main**()

{

**gets**(s);

build(s);

**int** p, q, dif;

**char** c;

**for** (**int** i = init; i < size; ++i) bst[i] = 1 << 30;

**while** (**gets**(s))

{

**for** (**int** i = init; i < size; ++i) cur[i] = 0;

dif = 0;

p = init;

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

{

c = s[i] - 'a';

**while** (p != init && !st[p].next[c])

{

q = st[p].link;

// since we have in dif the current difference for

// state p then p->lenght + dif is the real match.

// if such match is greater than the lenght of the

// link suffix then the difference becomes 0.

dif = min( (st[p].lenght + dif) - st[q].lenght, 0);

p = q;

}

**if** (st[p].next[c])

{

q = st[p].next[c];

// dif represents how much differ the current match

// from p->lenght ie. p->lenght + dif represents the

// current match so adding one we havethe next

// transition.

// finally p->lenght + dif + 1 is the lenght of the

// match after adding C so if we substract q->lenght

// then we have the difference solved for the next

// state.

dif = (st[p].lenght + dif + 1) - st[q].lenght;

p = q;

}

cur[p] = max(cur[p], st[p].lenght + dif);

}

// finding leaves

front = tail = 0;

**for** (**int** i = init; i < size; ++i)

++pointed[ st[p].link ];

**for** (**int** i = init; i < size; ++i)

**if** (!pointed[i]) Q[tail++] = i;

// updating fail nodes

**while** (front < tail)

{

p = Q[front++];

q = st[p].link;

cur[q] = max(cur[q], min(st[q].lenght, cur[p]));

--pointed[q];

**if** (!pointed[q]) Q[tail++] = q;

}

**for** (**int** i = init; i < size; ++i)

bst[i] = min(bst[i], cur[i]);

}

**int** sol = 0;

**for** (**int** i = init; i < size; ++i) sol = max(sol, bst[i]);

**if** (sol == (1 << 30)) sol = **strlen**(s);

**printf**("%d\n", sol);

**return** 0;

}// (++UH)++ :: agarwaen ... spoj 1812 - LCS2

* **Suffix Array**

**bool cmp**(**int** a, **int** b)

{

a += offset;

b += offset;

**if** (a >= N || b >= N)

**return** a > b;

**return**bu[a] <bu[b];

}

**int main**()

{

// dread

**scanf**("%s", s);

N = **strlen**(s);

// suffix array

REP(i, N)

{

sa[i] = i;

ta[i] = 0;

bu[i] = s[i];

}

offset = 0;

sort(sa, sa + N, cmp);

**for** (offset = 1; ta[N-1] != N-1; offset<<= 1)

{

**for** (**int** i = 0, j; i <N; i = j)

{

j = i+1;

**while** (j <N&&bu[ sa[j] ] == bu[ sa[i] ]) ++j;

sort(sa + i, sa + j, cmp);

}

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

**if** (bu[ sa[i] ] != bu[ sa[i+1] ])

ta[i+1] = ta[i] + 1;

**else**ta[i+1] = ta[i] + cmp(sa[i], sa[i+1]);

REP(i, N)bu[ sa[i] ] = ta[i];

}

**return** 0;

}// (++UH)++ :: agarwaen

* **Lyndon Decomposition**

**voidduval**(**constchar** \*s, vii&lyndon)

{

**int** N = **strlen**(s);

**int** i=0;

**while** (i < N)

{

**int** j=i+1, k=i;

**while** (j < N && s[k] <= s[j])

{

**if** (s[k] < s[j])

k = i;

**else** ++k;

++j;

}

**while** (i <= k)

{

lyndon.push\_back(MP(i, j-k));

i += j - k;

}

}

}

**intlowest\_cyclic\_shift**(**constchar** \*s)

{

**int** N = **strlen**(s);

**char** \*str = **newchar**[N<<1];

**strcpy**(str, s);

**strcpy**(str+N, s);

vii l;

duval(str, l);

REP(i, N)

**if** (l[i].STA< N && l[i].STA + l[i].LEN>= N)

**return** l[i].STA;

//err

**return** N;

}

**intmain**()

{

vii l;

**constchar** \*s = "ababa";

duval(s, l);

// print lyndon decomposition

REPI(i, l)

{

REP(j, i->LEN)

**printf**("%c", s[i->STA + j]);

**printf**("\n");

}

DB("---");

// print lowest cyclic shift

**int** N = **strlen**(s);

**int** sta = lowest\_cyclic\_shift(s);

REP(i, N)**printf**("%c", s[(sta+i)%N]);

**return** 0;

}// (++UH)++ :: agarwaen

/\*

--- Lyndon Decomposition ---

- Line is called simple if it is strictly less than any of their own suffix . Examples of simple lines: a, b, ab, aab, abb, ababb, abcd.

We can show that the string is simple if and only if it is strictly less than all of its non-trivial cyclic shifts .

- Let a string S. Then Lyndon decomposition string S is

W = W1, W2, ...Wk, where the rows of W are simple, while

W1 >= W2 >= ... >= Wk.

- For any string S a decomposition exists and is unique.

--- Lowest Cyclic Shift ---

- Construct Lyndon decomposition for string S+S.

- Find block Wi that starts at the position less than N and

ends at positions greater than or equal to N.

The position of the beginning of this block will be the beginning of

the desired cyclic shift.

\*/

# Geometry

* **Incenter**

C:\Users\Dany\Desktop\NumberedEquation2.gif

* **Distance between de Incenter and the Circumcenter**

**C:\Users\Dany\Desktop\Inline7.gif**

* **Calcular el centro de un circulo**

pnt**calc**(pnt a, pnt b, pnt c)

{

**double** A1 = b.X – a.X;

**double** B1 = b.Y – a.Y;

**double** C1 = (A1\*(a.X + b.X) + B1\*(a.Y + b.Y)) / 2;

**double** A2 = c.X - b.X;

**double** B2 = c.Y - b.Y;

**double** C2 = (A2\*(b.X + c.X) + B2\*(b.Y + c.Y)) / 2;

**double** det = A2 \* B1 - A1 \* B2;

**double** xo = (C2 \* B1 - C1 \* B2) / det;

**double** yo = (C1 \* A2 - C2 \* A1) / det;

**return** pnt(xo, yo);

}

* **Formula de las Medianas**

Ma = Sqrt(2(b^2 + c^2) – a^2) / 2

a = Sqrt(2(b^2 + c^2) – 4Ma^2)

a = Sqrt(b^2/2 – c^2 + 2Mb^2)

a = Sqrt(-b^2 + c^2/2 + 2Mc^2)

* **Winding Number**

**IntwindingNumber**(point p, vector<point>& poly)

{

**int** wn = 0;

REP(i, poly.size())

{

point cur = poly[i];

point nxt = poly[(i+1) % poly.size()];

// if (contains(cur, nxt, p)) return 0; // on

**if** (cur.yy > p.yy && nxt.yy <= p.yy && area2(cur, nxt, p) > 0) wn++;

**if** (cur.yy <= p.yy && nxt.yy > p.yy && area2(cur, nxt, p) < 0) wn--;

}

**return** wn;

}

* **Complex**

double dot(point a, point b) { return real(conj(a) \* b); }

double cross(point a, point b) { return imag(conj(a) \* b); }

point rotate\_by(point p, point about, double radians)

{

return (p - about) \* exp(point(0, radians)) + about;

}

point reflect(point p, point about1, point about2)

{

point z = p - about1;

point w = about2 - about1;

return conj(z / w) \* w + about1;

}

point intersect(point a, point b, point p, point q)

{

double d1 = cross(p - a, b - a);

double d2 = cross(q - a, b - a);

return (d1 \* q - d2 \* p) / (d1 - d2);

}

**double pointTOsegment(point a, point b, point c)**

**{**

**if (dot(b-a, c-b) > 0) return abs(b-c);**

**if (dot(a-b, c-a) > 0) return abs(a-c);**

**return abs( cross(b-a, c-a) / abs(a-b) );**

**}**

* **Convex Hull**

**typedef** pair<**long**, **long**>point;

**boolcw**(**const**point&a, **const**point&b, **const**point&c)

{

**return** (b.first - a.first) \* (c.second - a.second) –

(b.second - a.second) \* (c.first - a.first) < 0;

}

vector<point>**convexHull**(vector<point> p)

{

**int** n = p.size();

**if** (n <= 1)

**return** p;

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

**int** cnt = 0;

vector<point> q(n \* 2);

**for** (**int** i = 0; i < n; q[cnt++] = p[i++])

**for** (; cnt >= 2 && !cw(q[cnt - 2], q[cnt - 1], p[i]); --cnt)

;

**for** (**int** i = n - 2, t = cnt; i >= 0; q[cnt++] = p[i--])

**for** (; cnt > t && !cw(q[cnt - 2], q[cnt - 1], p[i]); --cnt)

;

q.resize(cnt - 1 - (q[0] == q[1]));

**return** q;

}

* **Cambio de Coordenadas**

Coordenadas cilindricas

X = p \* Cos(alpha)

Y = p \* Sin(alpha)

Z = Z

Coordenadas esfericas

X = r \* Sin(lat) \* Cos(lon)

Y = r \* Sin(lat) \* Sin(lon)

Z = r \* Cos(lat)

Relacion

r = sqrt( p^2 + z^2 ) lat = arctg( p/z ) lon = alpha

p = r \* Sin(lat) alpha = lon z = r \* Cos(lat)

**lat**: *latitud,* angulo entre el punto y el semieje positivo Z. (0 <= lat < 180)

**lon**: *longitud*, angulo entre el punto y el meridiano.(0 <= lon < 360)

**Number Theory**

// computes x such that a^x=b (mod m)

**int** discrete\_log ( **int** a, **int** b, **int** m )

{

**int** s = (**int**)Math.*sqrt* ( m) + 1 ;

Pair[] vals1 = **new** Pair[s];

**int** c = b;

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

{

vals1[i] = **new** Pair(c,i);

c = (c\*a)%m;

}

Arrays.*sort*(vals1);

Pair[] vals2 = **new** Pair[s];

**int** p = (**int**)powmod(a,s,m);

**int** d = p;

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

{

vals2[i-1] = **new** Pair(d,i);

d = (d\*p);

}

Arrays.*sort*(vals2);

**int** i = 0,j = 0;

**while**(i < s && j < s)

{

**if**(vals1[i].first ==vals2[j].first)

**return** (vals2[j].second\*s - vals1[i].second+m)%m;

**elseif**(vals1[i].first< vals2[j].first)i++;

**else** j++;

}

**return** -1;

}

Chinese remainder algorithm

**long** solve\_modular\_system(**int**[] a, **int**[] m)

{

**int** n = a.length;

**long** M = 1;

**for**(**int** i = 0; i < n; i++) M\*=m[i];

**long**[] x = **newlong**[n];

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

x[i] = (M/m[i])\*inverse1(M/m[i],m[i]);

**long** ans = 0;

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

ans = (ans + x[i]\*a[i])%M;

**return** ans;

}

Extended Euclid Algorithm

i64\* **euclid**(i64 a, i64 b)

{

**if** (b == 0)

{

i64 ret[] = {a, 1, 0};

**return** ret;

}

i64\* c = euclid(b, a % b);

i64 ret[] = {c[0], c[2], c[1] - c[2] \* (a / b)};

**return** ret;

}

Solves a^b modulo c [**tested** Quadratic Equations(CodeChef)]

i64**powMod**(i64 a, i64 b, i64 c)

{

**if** (b == 0)

**return** 1;

**if** (b & 1LL)

**return** (a \* powMod(a, b - 1, c)) % c;

i64 x = powMod(a, b >> 1, c);

**return** (x \* x) % c;

}

Computes phi(n)

**intPhi** ( **int** n ) {

**int** result = n ;

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

**if** ( n % I == 0 ) {

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

n / = I ;

result - = result / I ;

}

**if** ( n > 1 )

result - = result / n ;

Return result ;

}

Computes inverse of a modulo p [**tested** Quadratic Equations(CodeChef)]

i64**inverse**(i64 a, **int** p)

{

**return** powMod(a, p - 2, p);

}

i64**inverse**(i64 a, i64 n)

{

i64\* r = euclid(a, n);

i64 x = ( r[1] % n + n ) % n;

**return** x;

}

Solves a\*x = b (mod p)[**tested** Quadratic Equations(CodeChef)]

i64**solve\_linear**(i64 a, i64 b, **int**p)

{

**return** (b \* inverse(a, p)) % p;

}

Returns true if a is quadratic residue modulo p (prime)

**boolis\_quadratic\_residue**(i64 a, i64 p)

{

**if** (a == 0) **returntrue**;

**return** powMod(a, (p - 1) / 2, p) == 1;

}

Computes the kth discrete roots of a (mod n) when (k,phi(n))=1 [**tested** Timus 1141]

**int** discrete\_root(**int** k, **int** a, **int** n)

{

**int** phi = phi(n);

**int** s = (**int**)inverse(k,phi);

**return** (**int**)powmod(a,s,n);

}

Solves x^2=a (mod p)[**tested** Quadratic Equations(CodeChef), Timus 1132]

**long** solve\_quadratic(**long** a, **int** p)

{

**if**(a==0) **return** 0;

**if**(p==2) **return** a;

**if**(powmod(a,(p-1)/2,p)!= 1) **return** -1;

**int** phi = p-1;

**int** n = 0,k = 0;

**while**(phi%2==0)

{

phi/=2;

n++;

}

k = phi;

**int** q = 0;

**for**(**int** j = 2; j < p; j++)

**if**(powmod(j,(p-1)/2,p)==p-1)

{

q = j;

**break**;

}

**long** t = powmod(a,(k+1)/2,p);

**long** r = powmod(a,k,p);

**while**(r!=1)

{

**int** i = 0;

**int** v = 1;

**while**(powmod(r,v,p)!=1){v\*=2;i++;}

**long** e = powmod(2,n-i-1,p);

**long** u = powmod(q,k\*e,p);

t = (t\*u)%p;

r = (r\*u\*u)%p;

}

**return** (**int**)t;

}

Solves a\*x^2 + b\*x +c=0 (mod p) [**tested** Quadratic Equations(CodeChef)]

Set<Long> solve\_quadratic(**long** a, **long** b, **long** c, **int** p)

{

Set<Long> ans = **new** TreeSet<Long>();

**if**(c==0) ans.add(0L);

**if**(a==0) ans.add(solve\_linear((p-b)%p,c,p));

**else**

**if**(p==2 && (a+b+c)%2==0) ans.add(1L);

**else**

{

**long** r = ((b\*b)%p -(4\*a\*c)%p+ p)%p;

**long** x = solve\_quadratic(r,p);

**if**(x==-1) **returnnull**;

**long** w = solve\_linear((2\*a)%p,(x-b+p)%p,p);

ans.add(w);

w = solve\_linear((2\*a)%p,(p-x-b+p)%p,p);

ans.add(w);

}

**return** ans;

}

Primitive roots [**Tested** Timus 1268]

**int** generator ( **int** p )

{

List<Integer>Fact =**new** ArrayList<Integer>();

**int** Phi = p - 1, n = Phi;

**while**(prime[n]!= 0)

{

**int** f = prime[n];

Fact.add(f);

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

n/=f;

}

**if**(n>1) Fact.add(n);

**for** (**int** res = p-1; res >=2; res--)

{

**boolean** OK = **true** ;

**for** (**int** i = 0 ; i < Fact. size ( ) && OK ; i++)

OK &= powmod ( res, Phi / Fact.get(i), p ) != 1 ;

**if** ( OK ) **return** res;

}

**return** -1;

}

Factorial (mod p)

int factmod ( int n, int p ) {

int res = 1 ;

while ( n > 1 ) {

res = ( res \* ( ( n / p ) % 2 ? p - 1 : 1 ) ) % p ;

for ( int I = 2 ; I <= n % p ; + + I )

res = ( res \* I ) % p ;

n / = p ;

}

Return res % p ;

}

## Fibonacci Numbers

Consider the Fibonacci sequence F_isome modulo p. We prove that it is periodic, and moreover, the period begins withF_1=1(Ie preperiod contains only F_0).

We prove this by contradiction. Considerp^2+1pairs of Fibonacci numbers, taken modulo p:

(F_1,F_2),\ (F_2,F_3),\ \ldots,\ (F_{p^2+1},F_{p^2[...]

Since the modulo pcan only be p^2different pairs, including this sequence, there are at least two matching pairs. This already means that the sequence is periodic.

We now choose among all such identical pairs of two identical pairs with the lowest numbers. Let this pair with some rooms(F_a,F_{a+1})and (F_b,F_{b+1}). Prove thata=1. Indeed, otherwise, for they will have a pair of previous(F_{a-1},F_a)and (F_{b-1},F_b)Which, by the property of Fibonacci numbers will be equal to each other. However, this contradicts the fact that we chose the matching pairs with the lowest number, as required.

## Properties

Fibonacci numbers have many interesting mathematical properties.

Here are just some of them:

* Cassini Value:

F_{n+1} F_{n-1} - F_n^2 = (-1)^n.

* The rule of "adding:

F_{n+k} = F_k F_{n+1} + F_{k-1} F_n.

* From the previous equality in k = nfollows:

F_{2n} = F_n (F_{n+1} + F_{n-1}).

* From the previous equality by induction, we find that

F_{nk}

always a multiple of F_n.

* Converse is also true of the previous statement:

if F_mmultiply F_n, Then mmultiply n.

* **Miller Rabin [SPOJ(PON)]**

**#include**<cstdio>

**#include**<iostream>

**#include**<algorithm>

**usingnamespace** std;

**typedeflonglong**int64;

**#define**REP(i, n) **for** (**int** i = 0; i < (**int**)(n); ++i)

int64**modMul**(int64 a, int64 b, int64 mod)

{

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

int64 x = modMul(a, b/2, mod);

x = (2 \* x) % mod;

**if** (b & 1) x = (x + a) % mod;

**return** x;

}

int64**modPow**(int64 a, int64 n, int64 mod)

{

**if** (!n) **return** 1;

int64 x = modPow(a, n/2, mod);

x = modMul(x, x, mod);

**if** (n & 1) x = modMul(x, a, mod);

**return** x;

}

**boolwitness**(int64 a, int64 s, int64 d, int64 n)

{

int64 x = modPow(a, d, n);

**if** (x == 1 || x == n - 1) **return false**; // composite

REP(i, s - 1)

{

x = modMul(x, x, n);

**if** (x == 1) **return true**; // probably prime

**if** (x == n - 1) **return false**; // composite

}

**return true**; // probably prime

}

**boolisPrime**(int64 n)

{

**if** (n < 2) **return false**;

**if** (n == 2) **return true**;

**if** (n % 2 == 0) **return false**;

int64 d = n - 1, s = 0;

**while** (d % 2 == 0) ++s, d /= 2;

int64 test[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 0};

**for** (**int** i = 0; test[i] && test[i] < n; ++i)

**if** (witness(test[i], s, d, n))

**return false**; // composite

**return true**; // probably prime

}

**intmain**()

{

**int** T;

**for** (cin >> T ; T--;)

{

int64 N; cin >> N;

cout<< (isPrime(N) ? "YES" :"NO") << endl;

}

**return** 0;

}

**Math**

* **Karatsuba**

/\*

Karatsuba fast multiplication algorithm

(u. + v) \* (x. + y) (. means shifted; ex 1234 = 12.34 u=12 v=34)

= ux.. + uy. + xv. + vy

= ux.. + (uy + xv). + vy

note that: (u+v)\*(x+y) = (ux + uy + xv + vy)

we only need:

ux

vy

uy + xv wich is obtained from (u+v)\*(x+y) via uy + xv = (u+v)\*(x+y) - ux - vy

only 3 multiplications needed, each n/2 size

+ and - are O(n)

overall time = O(n ^ log2(3))

this implementation assumes a.length = b.length = power of 2.

\*/

**typedef** vector< int64 >big;

big**bruteMul**(**const**big& a, **const**big& b)

{

big ret(SZ(a) + SZ(b));

REP(i, SZ(a)) **if** (a[i])

REP(j, SZ(b))

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

**return** ret;

}

big**fastMul**(**const**big& a, **const**big& b)

{

**int** n = SZ(a);

assert(SZ(a) == SZ(b));

assert(!(n & (n - 1)));

**if** (n < 100)

**return** bruteMul(a, b);

big u(a.begin() + n/2, a.end());

big v(a.begin(), a.begin() + n/2);

big x(b.begin() + n/2, b.end());

big y(b.begin(), b.begin() + n/2);

big ux = fastMul(u, x);

big vy = fastMul(v, y);

REP(i, n/2)

{

u[i] += v[i];

x[i] += y[i];

}

big uvxy = fastMul(u, x);

big ret(SZ(a) + SZ(b));

REP(i, n)

{

ret[i] += vy[i];

ret[i + n/2] += uvxy[i] - ux[i] - vy[i];

ret[i + n] += ux[i];

}

**return** ret;

}

**Flow**

* **Dinic**

**namespace** Dinic

{

**#define** oo 1 << 20

**struct**edge

{

**int**dest;

**int**cap, flow;

**int**next;

**edge**() {}

**edge**(**int** dest, **int** cap, **int** next) :

dest(dest), cap(cap), next(next)

{

flow = 0;

}

};

**const int** MAXV = 1005;

**const int** MAXE = 1000000;

**int** Q[MAXV], lo, hi;

**int** nodes, source, sink;

**int** size;

**int** dst[ MAXV ];

**int** ptr[ MAXV ];

edge g[ MAXE ];

**int** first[ MAXV ];

**voidinitialize**(**int** \_\_nodes, **int** \_\_source, **int** \_\_sink)

{

size = 0;

nodes = \_\_nodes;

source = \_\_source;

sink = \_\_sink;

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

first[ i ] = -1;

}

**voidadd\_edge**(**int** u, **int** v, **int** c)

{

g[ size ] = edge(v, c, first[u]);

first[u] = size++;

g[ size ] = edge(u, 0, first[v]);

first[v] = size++;

}

**boolbfs**()

{

lo = 0;hi = 0;

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

dst[i] = -1;

dst[ source ] = 0;

**for** (Q[ hi++ ] = source; lo < hi; lo++)

{

**int** u = Q[ lo ];

**for** (**int** k = first[ u ]; k != -1; k = g[ k ].next)

{

**int** w = g[ k ].dest;

**if** (g[ k ].cap> g[ k ].flow&& dst[ w ] == -1)

{

dst[ w ] = dst[ u ] + 1;

Q[ hi++ ] = w;

}

}

}

**return** dst[ sink ] != -1;

}

**intdfs**(**int** u, **int** flow)

{

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

**if** (u == sink) **return** flow;

**for** (**int**&k = ptr[ u ]; k != -1; k = g[ k ].next)

{

**int** w = g[ k ].dest;

**int** cf = g[ k ].cap - g[ k ].flow;

**if** (dst[u] + 1 != dst[w]) **continue**;

**int** pushed = dfs(w, min(flow, cf));

**if** (pushed)

{

g[ k ].flow += pushed;

g[ k^1 ].flow -= pushed;

**return** pushed;

}

}

**return** 0;

}

**intdinic**()

{

**int** flow = 0;

**while** (**true**)

{

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

**for** (**int** i = 0; i < nodes; i++)ptr[i] = first[ i ];

**while** (**int** pushed = dfs(source, oo))flow += pushed;

}

**return** flow;

}};

* **Min Cost Flow**

**namespace** MCF

{

**#define** oo 1 << 29

**struct**edge

{

**int**dest;

**int**cap, flow, cost;

**int**next;

**edge**() {}

**edge**(**int** dest, **int** cap, **int** cost, **int** next) :

dest(dest), cap(cap), cost(cost), next(next)

{

flow = 0;

}

};

**const int** MAXV = 1005;

**const int** MAXE = 1000000;

**int** nodes, source, sink;

**int** size;

edge g[ MAXE ];

**int** first[ MAXV ];

**int** nLast[ MAXV ]; //last node

**int** eLast[ MAXV ]; //last edge

**int** cst[ MAXV ];

**voidinitialize**(**int** \_\_nodes, **int** \_\_source, **int** \_\_sink)

{

size = 0;

nodes = \_\_nodes;

source = \_\_source;

sink = \_\_sink;

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

first[ i ] = -1;

}

**voidaddEdge**(**int** u, **int** v, **int** c, **int** cost)

{

g[ size ] = edge(v, c, +cost, first[u]);

first[u] = size++;

g[ size ] = edge(u, 0, -cost, first[v]);

first[v] = size++;

}

pii**maxFlowMinCost**()

{

**int** flow = 0;

**int** flowCost = 0;

**while** (**true**)

{

priority\_queue<pii> Q;

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

cst[i] = oo;

cst[source] = 0;

Q.push(make\_pair(0, source));

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

{

**int** u = Q.top().second;

**int** c = -Q.top().first;

Q.pop();

**if** (u == sink) **break**;

**for** (**int** k = first[u]; k != -1; k = g[k].next)

{

**int** newU = g[k].dest;

**int** newC = g[k].cost + c;

**if** (g[k].cap> g[k].flow&& newC < cst[newU])

{

cst[newU] = newC;

//keep track//

nLast[newU] = u;

eLast[newU] = k;

//keep track//

Q.push(make\_pair(-newC, newU));

}

}

}

**if** (cst[sink] == oo) **break**;

**int** push = oo;

**for** (**int** u = sink; u != source; u = nLast[u])

push = min(push, g[ eLast[u] ].cap - g[ eLast[u] ].flow);

flow += push;

flowCost += cst[sink] \* push;

**for** (**int** u = sink; u != source; u = nLast[u])

{

g[ eLast[u] ].flow += push;

g[ eLast[u]^1 ].flow -= push;

}

}

**return** make\_pair(flow, flowCost);

}

};

* **Kuhn Munkres**

**namespace** KuhnMunkres

{

**const int** MAX\_N = 505;

**const int** INF = 100000000;

**const int** BASE = 1000;

**int** n;

**int** graph[MAX\_N][MAX\_N];

**bool** usedX[MAX\_N];

**bool** usedY[MAX\_N];

**int** linkTo[MAX\_N];

**int** labelX[MAX\_N];

**int** labelY[MAX\_N];

**boolfindPath**(**int** i)

{

usedX[i] = **true**;

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

**if** (!usedY[j] && labelX[i] + labelY[j] == graph[i][j] && linkTo[j] == -1)

{

usedY[j] = **true**;

linkTo[j] = i;

**return true**;

}

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

**if** (!usedY[j] && labelX[i] + labelY[j] == graph[i][j])

{

usedY[j] = **true**;

**if** (linkTo[j] == -1 || findPath(linkTo[j]))

{

linkTo[j] = i;

**return true**;

}

}

**return false**;

}

**intKuhnMunkres**()

{

//for min weighted match

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

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

graph[i][j] = BASE - graph[i][j];

**memset**(linkTo, -1, **sizeof**(linkTo));

**memset**(labelX, 0, **sizeof**(labelX));

**memset**(labelY, 0, **sizeof**(labelY));

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

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

labelX[i] = max(labelX[i], graph[i][j]);

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

**while** (**true**)

{

**memset**(usedX, 0, **sizeof**(usedX));

**memset**(usedY, 0, **sizeof**(usedY));

**if** (findPath(k)) **break**;

**int** delta = INF;

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

**if** (usedX[i])

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

**if** (!usedY[j])

delta = min(delta, labelX[i] + labelY[j] - graph[i][j]);

**if** (delta == 0 || delta == INF) **break**;

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

**if** (usedX[i])

labelX[i] -= delta;

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

**if** (usedY[i])

labelY[i] += delta;

}

//for min weighted match

**int** minTotal = BASE \* n;

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

minTotal -= labelX[i] + labelY[i];

//for max weighted match

**int** maxTotal = 0;

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

maxTotal += labelX[i] + labelY[i];

//return minTotal;

//return maxTotal;

}

};

* **Matching in General Graph**

**namespace** MaxMatching

{

**constint**

MaxV = 1001;

**int** V, E;

**int** match[MaxV];

**int** head, tail, Q[MaxV];

**int** start, finish;

**int** newbase;

**int** father[MaxV], base[MaxV];

**bool** graph[MaxV][MaxV];

**int** queue[MaxV];

**bool** inpath[MaxV];

**bool** inblossom[MaxV];

**bool** inqueue[MaxV];

**voidinitialize**(**int** \_\_nodes)

{

V = \_\_nodes;

**memset**(graph, **false**, **sizeof**(graph));

}

**voidaddEdge**(**int** u, **int** v)

{

graph[u][v] = **true**;

graph[v][u] = **true**;

}

**voidpush**(**int** u)

{

Q[tail++] = u;

inqueue[u] = **true**;

}

**intpop**()

{

**return** Q[head++];

}

**intfindCommonAncestor**(**int** u, **int** v)

{

**memset**(inpath, 0, **sizeof**(inpath));

**while** (**true**)

{

u = base[u];

inpath[u] = **true**;

**if** (u == start)

**break**;

u = father[ match[u] ];

}

**while** (**true**)

{

v = base[v];

**if** (inpath[v])

**break**;

v = father[ match[v] ];

}

**return** v;

}

**voidresetTrace**(**int** u)

{

**while** (base[u] != newbase)

{

**int** v = match[u];

inblossom[ base[u] ]= **true**;

inblossom[ base[v] ]= **true**;

u = father[ v ];

**if** (base[u] != newbase)

father[u] = v;

}

}

**voidblossomContract**(**int** u, **int** v)

{

newbase = findCommonAncestor(u, v);

**memset**(inblossom, **false**, **sizeof**(inblossom));

resetTrace(u);

resetTrace(v);

**if** (base[u] != newbase) father[u]= v;

**if** (base[v] != newbase) father[v]= u;

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

**if** (inblossom[ base[i] ])

{

base[ i ] = newbase;

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

push(i);

}

}

**voidfind\_augmenting\_path**()

{

**memset**(inqueue, **false**, **sizeof**(inqueue));

**memset**(father, 0, **sizeof**(father));

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

base[i] = i;

head = 0;

tail = 0;

push(start);

finish = 0;

**while** (head < tail)

{

**int** u = pop();

**for** (**int** v = 1; v <= V; v++)

**if** (graph[u][v] && (base[u] != base[v]) && (match[u] != v))

{

**if** ((v == start) || ((match[v] > 0) && (father[match[v]] > 0)))

{

blossomContract(u, v);

**continue**;

}

**if** (father[v] == 0)

{

father[v] = u;

**if** (match[v] > 0)

push(match[v]);

**else**

{

finish = v;

**return**;

}

}

}

}

}

**voidaugment\_path**()

{

**int** u = finish;

**while** (u > 0)

{

**int** v = father[u];

**int** w = match[v];

match[v] = u;

match[u] = v;

u = w;

}

}

**voidedmonds**()

{

**memset**(match, 0, **sizeof**(match));

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

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

{

start = i;

find\_augmenting\_path( );

**if** (finish > 0)

augment\_path();

}

}

};

**Miscellanious**

* **Numero de subsecuencias diferentes en W**

dp[0] = 1;

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

**if** (last[W[i] - 'a'])

dp[i] = (dp[i-1] - dp[last[W[i] - 'a'] - 1] + MODULO) % MODULO;

**else** dp[i] = dp[i-1];

last[W[i] - 'a'] = i;dp[i] = (dp[i] + dp[i-1]) % MODULO;

}

W = “abbc”

S = {“”, a, b, c, ab, ac, bb, bc, abb, abc, bbc, abbc}

* **Domino**

**{PROBLEM}**

Size :4xN

**{SOLUTION}**

dp[0] = S[0] = A[0] = 1;

i -> 1 .. N

dp[i] = dp[i - 1] + dp[i - 2] + 2 \* S[i - 2] + A[i - 2];

S[i] = S[i - 1] + dp[i]; A[i] = A[i - 2] + dp[i];

* **Numeracion Romana**

- el símbolo I sólo puede restar a V y a X.

- el símbolo X sólo resta a L y a C.

- el símbolo C sólo resta a D y a M.

1947 is 1000 + 900 + 40 + 7 = "M" + "CM" + "XL" + "VII"

{ **I** 1 }, { **II** 2 }, { **III** 3 }, { **IV** 4 }, { **V** 5 }, { **VI** 6 }, { **VII** 7 }, { **VIII** 8 }, { **IX** 9 }, { **X** 10 },{ **XI** 11 },{ **XII** 12 },{ **XX** 20 }, { **XXX** 30 }, { **XL** 40 }, { **L** 50 }, { **LX** 60 }, { **LXX** 70 },{ **LXXX** 80 },{ **XC** 90 }, { **LXIX** 69 },{ **CDL** 450 }, { **DCLXVI** 666 }, { **CMXCIX** 999 }, { **MCDXLIV** 1.444 }, { **MMVIII** 2.008 }, { **MMIX** 2.009 }, { **MMXII** 2.012 }

* **Bisiesto (Leap Year)**

**boolleapYear**(**int** Y)

{

**return** (Y % 4 == 0) && (!(Y % 100 == 0) || (Y % 400 == 0));

}

* **Gray Number**

cin>> N;

FOR(i, 1 << N)

{

**int** gi = i ^ (i >> 1);

FOR(j, N)

cout<< (**int**)((gi & (1 << j)) != 0);

cout<< endl;

}

* **Stable Marriage Problem**

/\* Solves Stable Marriage Problem [Tested STABLEMP(SPOJ),Block Game(Code Chef)]

\* Woman's preferences

\* pref[i] will contain the preference list of woman i

\* Man's preferences

\* rank[i][j] will contain the preference value of men i for woman j

\* returns the wife for each man\*/

**int**[] stable\_matching(**int**[][] pref, **int**[][] rank, **int** n)

{

**int**[] next = **newint**[n+1], wife = **newint**[n+1];

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

{

**int** w = i;

**while**(**true**)

{

**int** m = pref[w][++next[w]];

**if** (wife[m]==0) {wife[m] = w;**break**;}

**if** ( rank[m][w] < rank[m][ wife[m] ] ) {

**int** tmp = wife[m];

wife[m] = w;

w = tmp;

}

}

}

**return** wife;

}

**Data Structures**

* **Treap**

**#include**<cstdio>

**#include**<cstdlib>

**#include**<ctime>

**struct**treap {

**int**key;

**int**pri;

**int**cnt;

treap \*child[2];

**treap**() { }

**treap**(**const int**&key) :

key(key), pri(**rand**( ) % 100) {

cnt = 0;

child[0] = NULL;

child[1] = NULL;

}

};

**inlineintsize**(treap \*node) {

**return** node != NULL ? node->cnt : 0;

}

treap \***rotate**(treap \*v, **const int**&son) {

treap \*aux = v->child[son];

v->child[son] = aux->child[1-son];

aux->child[1-son] = v;

**return** aux;

}

treap \***insert**(treap \*node, **const int**&key) {

**if** (node == NULL)

**returnnew** treap(key);

**elseif** (node->key == key)

**return** node;

**int** son = key > node->key;

node->child[son] = insert(node->child[son], key);

**if** (node->child[son]->pri< node->pri)

node = rotate(node, son);

**return** node;

}

treap \***erase**(treap \*node, **const int**&key) {

**if** (node == NULL)

**return** NULL;

**if** (node->key == key) {

**if** (node->child[0] == NULL && node->child[1] == NULL)

**return** NULL;

**elseif** (node->child[0] == NULL) node = rotate(node, 1);

**elseif** (node->child[1] == NULL) node = rotate(node, 0);

**else** node = rotate(node, node->child[1] < node->child[0]);

node = erase(node, key);

} **else** {

**int** son = key > node->key;

node->child[son] = erase(node->child[son], key);

}

**return** node;

}

**inttree**(treap \*node) {

**if** (node == NULL)

**return** 0;

node->cnt = tree(node->child[0]) + tree(node->child[1]) + 1;

}

treap \***kth**(treap \*node, **int** k) {

**int** cnt = size(node->child[0]) + 1;

**if** (cnt > k) **return** kth(node->child[0], k);

**if** (cnt < k) **return** kth(node->child[1], k-cnt);

**return** node;

}

**int** Q, k, v;

treap \*root;

**voidtraverse**(treap \*node) {

**printf**("NODE : %d\n", node);

**printf**("KEY : %d\n", node->key);

**printf**("PRI : %d\n", node->pri);

**printf**("CH0 : %d\n", node->child[0]);

**printf**("CH1 : %d\n", node->child[1]);

**if** (node->child[0] != NULL) traverse(node->child[0]);

**if** (node->child[1] != NULL) traverse(node->child[1]);

}

**intmain**() {

/\* tree \*/

root = NULL;

/\* Queries \*/

**for**(**scanf**("%d", &Q); Q; --Q) {

**scanf**("%d%d", &k, &v);

**if** (k == 0) root = insert(root, v);

**if** (k == 1) root = erase(root, v);

**if** (k == 2) **printf**("%d\n", kth(root, k)->key);

**if** (k == 3) traverse(root);

}

**return** 0;}

* **KdTreeRect**

**const int** MAX = 10010;

**const**llint inf = 1LL << 60;

**struct**point {

llintx, y;

**int**cnt;

**point**() {}

**point**( llint \_x, llint \_y ) { x = \_x; y = \_y;cnt = 1; }

**inline**llint**dist**( **const**point&P ) **const**

{

llint dx = P.x - x;

llint dy = P.y - y;

**return** dx \* dx + dy \* dy;

}

};

**int** n,q;

point pts[ MAX ];

point tree[ MAX ];

**struct**rect {

llintx1, x2;

llinty1, y2;

**rect**() {}

**rect**( llint \_x1, llint \_x2, llint \_y1, llint \_y2 ) {

x1 = \_x1; x2 = \_x2; y1 = \_y1; y2 = \_y2;

}

**inlinevoidmerge**( **const**rect&R ) {

x1 = min( x1, R.x1 );

x2 = max( x2, R.x2 );

y1 = min(y1, R.y1 );

y2 = max(y2, R.y2 );

}

**inlineboolinsidecirc**(**const**point&P,llint r)**const**

{

point p1(x1,y1);

point p2(x1,y2);

point p3(x2,y2);

point p4(x2,y1);

**return** p1.dist(P)<=r\*r && p2.dist(P)<=r\*r && p3.dist(P)<=r\*r && p4.dist(P)<=r\*r;

}

**inlinebooloutsidecirc**(**const**point&c,llint r)**const**

{

point p1(x1,y1);

point p2(x1,y2);

point p3(x2,y2);

point p4(x2,y1);

**if**(p1.dist(c)>r\*r && p2.dist(c)>r\*r && p3.dist(c)>r\*r && p4.dist(c)>r\*r)

{

**if**(x1<= c.x&& c.x<= x2)

**return** abs(c.y-y1)> r && abs(c.y-y2) > r;

**if**(y1<= c.y&& c.y<= y2 )

**return** abs(c.x-x1) > r && abs(c.x-x2)> r;

}

**return false**;

}

**boolcontains**(point P){

**return** P.x>=x1&& P.x<=x2&& P.y>=y1&& P.y<=y2;

}

};

**struct**node

{

point \*pt;

node \*left, \*right;

rectrt;

**int**cnt;

};

node mem[ MAX ];

node \*alloc = mem;

**inlineboolcmpfx**( **const**point&A, **const**point&B ){

**if**( A.x != B.x ) **return** A.x< B.x;

**if**( A.y != B.y ) **return** A.y< B.y;

**returnfalse**;

}

**inlineboolcmpfy**( **const**point&A, **const**point&B ){

**if**( A.y != B.y ) **return** A.y< B.y;

**if**( A.x != B.x ) **return** A.x< B.x;

**returnfalse**;

}

node \***construct**( **int** lo, **int** hi, **int** D )

{

**if**( lo >= hi ) **return** 0;

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

nth\_element( tree + lo, tree + mid, tree + hi, D&1 ? cmpfx : cmpfy );

node \*root = alloc++;

root->pt = &tree[mid];

root->left = construct( lo, mid, D^1 );

root->right = construct( mid+1, hi, D^1 );

root->rt = rect( tree[mid].x, tree[mid].x, tree[mid].y, tree[mid].y );

root->cnt = tree[mid].cnt;

**if**( root->left){

root->rt.merge( root->left->rt );

root->cnt += root->left->cnt;

}

**if**( root->right){

root->rt.merge( root->right->rt );

root->cnt += root->right->cnt;

}

**return** root;

}

**intsolve**( node \*pos, **const**point&P, llint r )

{

**if**( pos == 0 ) **return** 0;

**if**(pos->rt.insidecirc(P,r)) **return** pos->cnt;

**elseif**(pos->rt.outsidecirc(P,r) && !pos->rt.contains(P)) **return** 0;

**elsereturn** solve(pos->left,P,r) + solve(pos->right,P,r) + ((pos->pt->dist(P)<=r\*r)?1:0);

}

* **KDtree Nearest Neighbour**

**const**i64 INF = (1LL)<<60;

**constint** maxn = 100005;

**int** tx[maxn], ty[maxn];

**int** n;

pii points[maxn];

**boolcmpX**(**const**pii&a, **const**pii&b) {

**return** a.first< b.first;

}

**boolcmpY**(**const**pii&a, **const**pii&b) {

**return** a.second< b.second;

}

**voidbuild**(**int** low, **int** high, **bool** divX)

{

**if** (low >= high)

**return**;

**int** mid = (low + high) >> 1;

nth\_element(points + low, points+mid, points+high, divX ? cmpX : cmpY);

tx[mid] = points[mid].first;

ty[mid] = points[mid].second;

build(low, mid, !divX);

build(mid + 1, high, !divX);

}

i64 bestDist;

**int** bestNode;

**voidfindNearestNeighbour**(**int** low, **int** high, **int** x, **int** y, **bool** divX)

{

**if** (low >= high)

**return**;

**int** mid = (low + high) >> 1;

i64 dx = x - tx[mid];

i64 dy = y - ty[mid];

i64 d = dx \* dx + dy \* dy;

**if** (bestDist > d) {bestDist = d;bestNode = mid;}

i64 delta = divX ? dx : dy;

i64 delta2 = delta \* delta;

**int** l1 = low;**int** h1 = mid;**int** l2 = mid + 1;**int** h2 = high;

**if** (delta > 0) {

**int** t = l1;

l1 = l2;

l2 = t;

t = h1;

h1 = h2;

h2 = t;

}

findNearestNeighbour(l1, h1, x, y, !divX);

**if** (delta2 < bestDist)

findNearestNeighbour(l2, h2, x, y, !divX);

}

**intfindNearestNeighbour**(**int** x, **int** y)

{

bestDist = INF;

findNearestNeighbour(0, n, x, y, **true**);

**return** bestNode;

}

**Graph**

* **Articulation Points**

**int** N, M;

**int** dtime, root;

**int** num[MAX], low[MAX];

**bool** ans[MAX];

vi G[MAX];

**void** **dfs**(**int** x)

{

num[x] = low[x] = ++dtime;

REPI(k, G[x])

{

**int** v = \*k;

**if** (!num[v])

{

dfs(v);

**if** ((x!=root && low[v]>=num[x])||( x == root && num[v]-num[root] > 1))

ans[x] = **true**;

**else**

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

}

**else**

low[x] = min(low[x], num[v]);

}

}

* **Biconected Components – No Articulation Points**

/\*

Koder : Ramon Alejandro Reyes Fajardo

Task : Biconnected components

>>> Biconnected Graph <<<

- Any graph which contains no articulation points.

- Biconnected graphs have at least two vertex-disjoint paths between any pair of vertices.

- To disconnect a biconnected graph, we must remove at least two vertices.

- We can generalize the above concepts to k-connected graphs (k vertices must be removed to disconnect the graph).

>>> Biconnected components of a graph <<<

- A maximal biconnected subgraph (i.e., not contained in any larger biconnected subgraph).

- Note that biconnected components partition the edges (not the vertices!!) into disjoint sets.

\*/

**#define** edge second

**#define** next first

**typedef** pair<**int**, **int**> Edge;

**typedef** vector<Edge> Edges;

**int** N, M;

**int** u, v, t;

**int** d[MAX], l[MAX];

**bool** pushed[EDG];

Edge tmp, curr;

stack<Edge> S;

vector<Edges> G;

**void** **dfs**(**int** x)

{

d[x] = l[x] = ++t;

**for** (Edges::iterator it = G[x].begin(); it != G[x].end(); it++)

{

**if** (!pushed[it->edge])

{

S.push(make\_pair(x, it->next));

pushed[it->edge] = 1;

}

**if** (!d[it->next])

{

dfs(it->next);

**if** (l[it->next] >= d[x])

{

curr = make\_pair(x, it->next);

**do**{

tmp = S.top(); S.pop();

**printf**("%d %d\n", tmp.first + 1, tmp.second + 1);

}**while** (tmp != curr);

**printf**("\n");

}

**else**

**if** (l[it->next] < l[x]) l[x] = l[it->next];

}

**else**

**if** (d[it->next] < l[x]) l[x] = d[it->next];

}

}

* **Biconected Components – No Bridges**

/\* Partition of the graph into subgraph having at least two edge-disjoint paths between

\* any pair of vertices.

\* \*/

**int** N, M, C;

**int** dtime;

**int** num[MAX], low[MAX], cmp[MAX], par[MAX];

**bool** bio[MAX];

stack<**int**> S;

vi G[MAX];

**void** **dfs**(**int** u)

{

num[u] = low[u] = ++dtime;

S.push(u);

**int** s = G[u].size();

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

{

**int** v = G[u][i];

**if** (par[u] == v) **continue**;

**if** (!num[v])

{

par[v] = u;

dfs(v);

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

}

**else**

**if** (!bio[v])

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

}

**if** (low[u] == num[u])

{

**while** (!bio[u])

{

cmp[S.top()] = C;

bio[S.top()] = **true**;

//cout << S.top()+1 <<' ';

S.pop();

}

//cout << '\n';

C++;

}

}

* **LCA**

**const** **int** MAX = 10010;

**const** **int** LOG = 18;

**int** log2[MAX];

**int** l[MAX];

**int** p[MAX][LOG];

**void** **build**(vi G[MAX], **int** x)

{

queue<**int**> Q;

l[x] = 0;

p[x][0] = -1;

**for** (Q.push(x); !Q.empty(); Q.pop())

{

x = Q.front();

REPI(i, G[x])

{

**int** v = \*i;

**if** (p[x][0] == v) **continue**;

p[v][0] = x;

l[v] = l[x]+1;

FOR(j, log2[ l[v] ])

p[v][j] = p[ p[v][j-1] ][ j-1 ];

Q.push(v);

}

}

}

**int** **query**(**int** u, **int** v)

{

**if** (l[v] > l[u])

swap(u, v);

**if** (l[u] != l[v])

**for** (**int** i = log2[ l[u] ]; i >= 0; --i)

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

u = p[u][i];

**if** (u == v)

**return** u;

**for** (**int** i = log2[ l[u] ]; i >= 0; --i)

**if** ((1 << i) <= l[u] && p[u][i] != p[v][i])

{

u = p[u][i];

v = p[v][i];

}

**return** p[u][0];

}

* **Strongly Connected Components**

**int** N, M, C;

**int** dtime;

**int** num[MAX], low[MAX], cmp[MAX];

**bool** bio[MAX];

vi G[MAX];

stack<**int**> S;

**void** **dfs**(**int** u)

{

num[u] = low[u] = ++dtime;

S.push(u);

**int** s = G[u].size();

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

{

**int** v = G[u][i];

**if** (!num[v])

{

dfs(v);

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

}

**else**

**if** (!bio[v])

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

}

**if** (low[u] == num[u])

{

**while** (!bio[u])

{

cmp[S.top()] = C;

bio[S.top()] = **true**;

//cout << S.top()+1 <<' ';

S.pop();

}

//cout << '\n';

C++;

}

}