**Number Theory**

1. Bigmod [Repeated Square Method]

int bigmod (int b, int p, int m) {

int res = 1, x = b % m;

while (p > 0) {

if (p & 1) res = (res \* x) % m;

x = (x \* x) % m;

p >>= 1;

}

return res;

}

1. Extended GCD

/\* gcd(a, b) = gcd(b%a, a)

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

x’ = x+(k\*b/gcd), y’ = y-(k\*a/gcd); infinite soln \*/

int extended\_gcd(int a, int b, int & x, int & y) {

if (a == 0) {

x = 0;

y = 1;

return b;

}

int x1, y1;

int gcd = extended\_gcd(b % a, a, x1, y1);

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

y = x1;

return gcd;

}

1. Modulo Inverse[Ext. GCD]

int modInv(int a, int m) {

int x, y, g;

g = extended\_gcd(a, m, x, y);

if (g != 1) return -1; //no solution

else {

int mod\_inv = (x % m + m) % m;

return mod\_inv;

}

}

1. Sieve of Eratosthenes

void sieve() {

for(int i = 3; i < MX; i += 2) {

if(!mark[i])

for(int j = i\*i; j < MX; j += 2\*i)

mark[j] = true;

}

primes.pb(2);

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

if(!mark[i]) primes.pb(i);

}

1. Bitwise Sieve

#define MX 100000008

vector <llu> primes;

int status[(MX/32)+2];

bool Check(llu N, llu pos) {

return (bool)(N & (1<<pos));

}

int Set(llu N, llu pos) {

return N = N | (1<<pos);

}

//Complexity: O(NloglogN)

void bit\_sieve(llu N) {

llu sqrtN, i, j;

sqrtN = sqrt(N);

// j>>5 == j/32 , j & 31 == j % 32

for(i = 3; i <= sqrtN; i += 2) {

if(Check(status[i>>5],i&31) == 0) {

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

status[j>>5] = Set(status[j>>5], j&31);

}

}

}

primes.pb (2);

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

if( Check(status[i>>5],i&31)==0) {

primes.pb (i);

}

}

}

1. Phi Sieve

llu tot[MX];

void phi\_sieve() {

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

tot[i] = i;

if(i%2==0) tot[i] -= tot[i]/2;

}

for(int i = 3; i < MX; i+=2) {

if(tot[i] == i) {

for(int j = i; j < MX; j+=i) {

tot[j] -= tot[j]/i;

}

}

}

}

1. Segmented Sieve

bool mark[MX];

vector <int> sgPrimes;

int segmentedSieve (int a, int b) {

if(a == 1) a++;

int i, j, p, sqrtN = sqrt(b);

memset(mark, false, sizeof mark);

for(i = 0; i < primes.size() &&

primes[i] <= sqrtN; i++) {

p = primes[i];

j = p \* p;

if(j < a) j = ( ( a + p - 1 ) / p ) \* p;

for ( ; j <= b; j += p) {

mark[j-a] = true;

}

}

int cnt = 0;

for (i = a; i <= b; i++) {

if (!mark[i-a]) cnt++;

sgPrimes.pb(i);

}

return cnt;

}

1. Linear Diophatine Eqn

/\*x' = x + (k\*B/g), y' = y - (k\*A/g); infinite soln

if A=B=0, C must equal 0 and any x, y is solution; if A|B=0, (x, y) = (C/A, k) | (k, C/B)\*/

bool LDE(int A, int B, int C, int \*x, int \*y) {

int g = gcd(A, B);

if (C%g != 0) return false; //No Solution

int a = A/g, b = B/g, c = C/g;

extended\_gcd(a, b, x, y); //Solve ax + by = 1

if (g < 0) { //Making Sure gcd(a,b) = 1

a \*= -1; b \*= -1; c \*= -1;

}

\*x \*= c; \*y \*= c; //ax + by = c

return true; //Solution Exists

}

1. Modulo Inverse from 1 to N

int inv[MX];

inv[1] = 1;

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

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

inv[i] = inv[i] + m;

}

1. Bigmod [Repeated Squaring Method]

int bigmod (int b, int p, int m) {

int res = 1, x = b % m;

while (p > 0) {

if (p & 1) res = (res \* x) % m;

x = (x \* x) % m;

p >>= 1;

}

return res;

}

1. Chinese Remainder Theorem

ll CRT(vector<ll>&mod, vector<ll>&rem, ll n) {

ll prod = 1;

for (ll i = 0; i < n; i++) prod \*= mod[i];

ll result = 0;

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

ll pp = prod / mod[i];

result += rem[i] \* modInv(pp, mod[i]) \* pp;

}

return result % prod;

}

**/\*** *GCD SUM, g(n) =*

*LCM SUM, g(n) =*

*SUM OF CO-PRIMES, g(n) =*

***MULTIPLICATION OF DIVISORS, f(n) =***

**\*/**

**Data Structure**

1. Binary Index Tree

int n, bit[MX];

/\* bit[n], bit is like cumulative array

but contains partial sums. \*/

int query(int indx) {

int sum = 0;

for(indx = indx+1; indx > 0; indx -= indx&-indx)

sum += bit[indx];

return sum;

}

void update(int indx, int val) {

for(indx = indx+1; indx <= n; indx += indx&-indx)

bit[indx] += val;

}

1. Sparse Table

//Complexity- built: O(NlogN), query: O(1);

int spt[MX][MX], arr[MX];

void build(int n) {

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

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

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

if(arr[spt[i][j-1]] < arr[spt[i + (1<<(j-1))][j-1]])

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

else

spt[i][j] = spt[i+(1<<(j-1))][j-1];

}}

int query(int L, int R) {

int j = log2(R - L + 1);

if(arr[spt[L][j]] < arr[spt[R-(1<<j)+1][j]])

return arr[spt[L][j]];

else

return arr[spt[R-(1<<j)+1][j]];

}

1. Merge Sort Tree

//Space & Time Complexity: O(N\*logN)

int arr[MX];

vector <int> tree[5\*MX];

void build(int pos, int tl, int tr)

{

if(tl == tr) {

tree[pos].pb(arr[tl]);

return ;

}

int mid = (tl+tr)/2;

build(2\*pos, tl, mid);

build(2\*pos+1, mid+1, tr);

merge( tree[2\*pos].begin(), tree[pos\*2].end(),

tree[2\*pos+1].begin(), tree[2\*pos+1].end(),

back\_inserter(tree[pos]) );

}

int query(int pos, int tl, int tr, int l, int r, int k) {

if(tl > r || tr < l) return 0;

if(tl >= l && tr <= r) {

//binary search over the current sorted vector to

find elements smaller than K or equal to K

return upper\_bound(tree[pos].begin(),

tree[pos].end(), k) - tree[pos].begin();

}

int mid = (tl+tr)/2;

return query(2\*pos, tl, mid, l, r, k) +

query(2\*pos+1, mid+1, tr, l, r, k);

}

* *Find k-th number in a range with O(log2N)*

***Sol n :***  take input as pairs <ff, ss> & push them into a vector. ff = value & ss = index. sort those pairs according to values. then make a MST on it. now try to find the leftmost k-th index of the given query range in the sorted vector. answer will the stored value at that index.

1. Segment Tree [plus lazy propagation]

int arr[MX], tree[4\*MX];

void build(int pos, int tl, int tr) //Complexity: O(N)

{

if(tl == tr) {

tree[pos] = arr[tl];

return;

}

int mid = (tl+tr)/2;

build(pos\*2, tl, mid);

build(pos\*2+1, mid+1, tr);

tree[pos] = tree[pos\*2] + tree[pos\*2+1];

}

void push\_down(int pos, int tl, int tr)

{

tree[pos] += (tr-tl+1)\*prop[pos];

if(tl != tr) {

prop[pos\*2] += prop[pos];

prop[pos\*2+1] += prop[pos];

}

prop[pos] = 0;

}

void update(int pos, int tl, int tr, int indx, int nval) //Complexity: O(logN)

{

if(indx < tl || indx > tr) return;

if(tl == tr) {

tree[pos] = nval;

return;

}

int mid = (tl+tr)/2;

update(pos\*2, tl, mid, indx, nval);

update(pos\*2+1, mid+1, tr, indx, nval);

tree[pos] = tree[pos\*2] + tree[pos\*2+1];

}

void range\_update(int pos, int tl, int tr, int l, int r, int x) //Complexity: O(logN)

{

if(prop[pos]) push\_down(pos, tl, tr);

if(tl > r || tr < l) return;

if(l <= tl && tr <= r) {

tree[pos] += (tr-tl+1)\*x;

if(tl != tr) {

prop[pos\*2] += x;

prop[pos\*2+1] += x;

}

return;

}

int mid = (tl+tr)/2;

range\_update(pos\*2, tl, mid, l, r, x);

range\_update(pos\*2+1, mid+1, tr, l, r, x);

tree[pos] = tree[pos\*2] + tree[pos\*2+1];

}

int query(int pos, int tl, int tr, int l, int r)

//Complexity: O(logN)

{

/\*if(prop[pos]) push\_down(pos, tl, tr);\*/

if(l > tr || r < tl) return 0;

if(tl >= l && tr <= r) return tree[pos];

int mid = (tl+tr)/2;

int Lch = query(pos\*2, tl, mid, l, r);

int Rch = query(pos\*2+1, mid+1, tr, l, r);

return Lch + Rch;

}

1. Persistent Segment Tree

int arr[MX];

struct node {

node \*left, \*right;

int val;

node (int a = 0, node \*b = NULL, node \*c = NULL) :

val(a), left(b), right(c) {}

void build(int tl, int tr) {

if(tl == tr) {

val = arr[tl];

return;

}

left = new node();

right = new node();

int mid = (tl+tr)/2;

left -> build(tl, mid);

right-> build(mid+1, tr);

val = left -> val + right -> val;

}

node\* update(int tl, int tr, int indx, int v) {

if(tl > indx || tr < indx) return this;

if(tl == tr) {

node \*ret = new node(val, left, right);

ret -> val += v;

return ret;

}

int mid = (tl+tr)/2;

node \*ret = new node();

ret -> left = left -> update(tl, mid, indx, v);

ret -> right= right-> update(mid+1, tr, indx, v);

ret -> val = ret -> left -> val + ret -> right -> val;

return ret;

}

int query(int tl, int tr, int l, int r) {

if(tl > r || tr < l) return 0;

if(tl >= l && tr <= r) return val;

int mid = (L+R)/2;

int Lch = left -> query(L, mid, l, r);

int Rch = right-> query(mid+1, R, l, r);

return Lch+Rch;

}} \*root[100005]; //total different versions of ST

int main()

{

arr[1] = 2, arr[2] = 7, arr[3] = 3, arr[4] = 5, arr[5] = 1;

root[0] = new node();

root[0] -> build(1, 5);

root[1] = root[0] -> update(1, 5, 2, 2);

}

1. 2D Binary Index Tree [used just for range-sum]

int n, m, bit[MX][MX]; // bit[n][m]

/\* for a specific rectangle:

query(rx, ry) - query(lx-1, ry) - query(rx, ly-1) + query(lx-1, ly-1), where lx <= ly & rx <= ry \*/

int query(int x, int y) //Complexity: O(log(N) \* log(M))

{

int res = 0;

for(x = x+1; x > 0; x -= x&-x) { //log(N)

for(int py = y+1; py > 0; py -= py&-py) { //log(M)

res += bit[x][py];

}

}

return res;

}

void update(int x, int y, int val)

//Complexity: O(log(N) \* log(M))

{

for(x = x+1; x <= n; x += x&-x) { //log(N)

for(int py = y+1; py <= m; py += py&-py) { // log(M)

bit[x][py] += val;

}

}

}

1. 2D Segment Tree

#define MX 1003

int n, m;

int mat[MX][MX], tree[4\*MX][4\*MX];

//mat[n][m], 64MB memory needed for 1003\*1003

void build\_y(int vx, int lx, int rx, int vy, int ly, int ry)

{

if(ly == ry) {

if(lx == rx) tree[vx][vy] = mat[lx][ly];

else tree[vx][vy] = tree[vx\*2][vy] + tree[vx\*2+1][vy];

return;

}

int mid = (ly+ry) / 2;

build\_y(vx, lx, rx, vy\*2, ly, mid);

build\_y(vx, lx, rx, vy\*2+1, mid+1, ry);

tree[vx][vy] = tree[vx][vy\*2] + tree[vx][vy\*2+1];

}

void build\_x(int vx, int lx, int rx)

//Total Build Complexity: O(N\*M)

{

if(lx == rx) {

build\_y(vx, lx, rx, 1, 1, m);

return;

}

int mid = (lx+rx) / 2;

build\_x(vx\*2, lx, mid);

build\_x(vx\*2+1, mid+1, rx);

build\_y(vx, lx, rx, 1, 1, m);

}

int query\_y(int vx, int vy, int Ly, int Ry, int ly, int ry)

{

if(Ly > ry || Ry < ly) return 0;

if(Ly >= ly && Ry <= ry) return tree[vx][vy];

int mid = (Ly+Ry) / 2;

int Lch = query\_y(vx, vy\*2, Ly, mid, ly, ry);

int Rch = query\_y(vx, vy\*2+1, mid+1, Ry, ly, ry);

return Lch + Rch;

}

*int query\_x(int vx, int Lx, int Rx, int lx, int rx, int ly, int ry) //Total Query Complexity: O(logN\*logM)*

{

if(Lx > rx || Rx < lx) return 0;

if(Lx >= lx && Rx <= rx)

return query\_y(vx, 1, 1, m, ly, ry);

int mid = (Lx+Rx) / 2;

int Lch = query\_x(vx\*2, Lx, mid, lx, rx, ly, ry);

int Rch = query\_x(vx\*2+1, mid+1, Rx, lx, rx, ly, ry);

return Lch + Rch;

}

***void update\_y(int vx, int Lx, int Rx, int vy, int Ly, int Ry, int ry, int val)***

{

if(Ly > ry || Ry < ry) return;

if(Ly >= ry && Ry <= ry) {

if(Lx == Rx) {

tree[vx][vy] = val;

}

else {

tree[vx][vy] = tree[vx\*2][vy] + tree[vx\*2+1][vy];

}

return;

}

int mid = (Ly+Ry) / 2;

update\_y(vx, Lx, Rx, vy\*2, Ly, mid, ry, val);

update\_y(vx, Lx, Rx, vy\*2+1, mid+1, Ry, ry, val);

tree[vx][vy] = tree[vx][vy\*2] + tree[vx][vy\*2+1];

}

***void update\_x(int vx, int Lx, int Rx, int lx, int ry, int val) //Total Update Complexity: O(logN\*logM)***

{

if(Lx > lx || Rx < lx) return;

if(Lx >= lx && Rx <= lx) {

update\_y(vx, Lx, Rx, 1, 1, m, ry, val);

return;

}

int mid = (Lx+Rx) / 2;

update\_x(vx\*2, Lx, mid, lx, ry, val);

update\_x(vx\*2+1, mid+1, Rx, lx, ry, val);

update\_y(vx, Lx, Rx, 1, 1, m, ry, val);

}

1. Policy Based Data Structure

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

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

***using namespace \_\_gnu\_pbds;***

***template <typename T> using orderedSet =***

***tree<T, null\_type, less<T>,***

***rb\_tree\_tag, tree\_order\_statistics\_node\_update>;***

***template <typename T> using orderedMultiSet =***

***tree<T, null\_type, less\_equal<T>,***

***rb\_tree\_tag, tree\_order\_statistics\_node\_update>;***

***orderedSet <int> os;***

***orderedMultiSet <int> oms;***

/\*Alternative of unordered\_map is gp\_hash\_table

(x6 times faster):

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

***using namespace \_\_gnu\_pbds;***

***gp\_hash\_table <int, int> table;***

***gp\_hash\_table <pair<ll, pll>, ll, hash\_pair> dp;***

***//how to use 'pairs' as key:***

***struct hash\_pair {***

***ll operator() (pair<ll, pll> x) const {***

***return x.ff\*63 + x.ss.ff\*31 + x.ss.ss;***

***}***

***};***

\*/

/\*

PBDS performs all the operations as performed by the set data structure in STL in log(n) complexity.

Two additional operations:

1) order\_of\_key: The number of items in a set that are strictly smaller than k; Complexity: O(LogN)

2) find\_by\_order: It returns an iterator to the ith largest element; Complexity: O(LogN)

After using less\_equal instead of less for multiset, lower\_bound works like upper\_bound function

and upper\_bound works like lower\_bound.

\*/

//syntax

int main()

{

cout<<"Set:"<<endl;

os.insert(2);

os.insert(5);

os.insert(2);

/\* [2, 5] \*/

cout<<os.size()<<endl;

cout<<"0: "<<\*os.find\_by\_order(0)<<endl; //2

cout<<"1: "<<\*os.find\_by\_order(1)<<endl; //5

cout<<endl;

cout<<"5: "<<os.order\_of\_key(5) << endl; //1

cout<<"2: "<<os.order\_of\_key(2) << endl; //0

cout<<"Multiset: "<<endl;

oms.insert(2);

oms.insert(5);

oms.insert(2);

/\* [2, 2, 5] \*/

cout<<oms.size()<<endl;

cout<<"0: "<<\*oms.find\_by\_order(0)<<endl; //2

cout<<"1: "<<\*oms.find\_by\_order(1)<<endl; //2

cout<<"2: "<<\*oms.find\_by\_order(2)<<endl; //5

cout<<endl;

cout <<"5: "<<oms.order\_of\_key(5) << endl; //2

cout <<"2: "<<oms.order\_of\_key(2) << endl; //0

}

1. Heavy Light Decomposition

//Heavy Light Decomposition

#define MX 100005

int cur\_pos;

vector <int> adj[MX];

int parent[MX], depth[MX], heavy[MX], head[MX],

pos[MX];

int dfs(int u) { //Complexity: O(V+E)

int sz = 1, mx = 0;

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

int v = adj[u][i];

if (v != parent[u]) {

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

int subtree\_sz = dfs(v);

sz += subtree\_sz;

if (subtree\_sz > mx) {

mx = subtree\_sz, heavy[u] = v;

}

}

}

return sz;

}

void decompose(int u, int h) { //Complexity: O(V+E)

head[u] = h, pos[u] = cur\_pos++;

if (heavy[u] != -1) {

decompose(heavy[u], h);

}

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

int v = adj[u][i];

if (v != parent[u] && v != heavy[u])

decompose(v, v);

}

}

void gen(int n) { //Complexity: O(V+E)

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

cur\_pos = 0;

dfs(0);

decompose(0, 0);

}

/\* following query is about to find the maximum value between two nodes in tree \*/

int query(int a, int b) { //Complexity: O(logN)

int res = 0;

for ( ; head[a] != head[b]; b = parent[head[b]] ) {

if (depth[head[a]] > depth[head[b]]) {

swap(a, b);

}

int cur\_heavy\_path\_max =

ST\_query(pos[head[b]], pos[b]);

res = max(res, cur\_heavy\_path\_max);

}

if (depth[a] > depth[b]) {

swap(a, b);

}

int last\_heavy\_path\_max = ST\_query(pos[a], pos[b]);

res = max(res, last\_heavy\_path\_max);

return res;

}

1. Centroid Decomposition

int subtree[N], parentcentroid[N];

set<int> adj[N];

void dfs(int k, int par) {

nodes++;

subtree[k] = 1;

for(auto it : adj[k]) {

if(it == par) continue;

dfs(it, k);

subtree[k] += subtree[it];

}

}

int centroid(int k, int par) {

for(auto it : adj[k]) {

if(it == par) continue;

if(subtree[it] >(nodes>>1)) return centroid(it, k);

}

return k;

}

void decompose(int k, int par) {

nodes=0;

dfs(k, k);

int node = centroid(k, k);

parentcentroid[node] = par;

for(auto it : adj[node]) {

adj[it].erase(node);

decompose(it, node);

}

}

1. Lowest Common Ancestor [RMQ]

bool vis[MX];

int dfs\_counter;

int dis\_time[MX], euler[MX\*2], depth[MX\*2];

vector <int> adj[MX];

void dfs(int u, int h) {

euler[dfs\_counter] = u;

depth[dfs\_counter] = h;

dis\_time[u] = dfs\_counter++;

vis[u] = true;

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

int v = adj[u][i];

if(!vis[v]) {

dfs(v, h+1);

euler[dfs\_counter] = u;

depth[dfs\_counter++] = h;

}

}

}

/\* sparse table, built: O(NlogN), query: O(1);

if build by ST, each query will be O(logN) \*/

int spt[MX][LOG2(MX)];

void build(int n) {

for(int i = 0; i < n; i++) //0-based index

spt[i][0] = i;

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

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

if(depth[spt[i][j-1]] < depth[spt[i + (1<<(j-1))][j-1]])

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

else

spt[i][j] = spt[i+(1<<(j-1))][j-1];

}

}

}

int query(int L, int R)

{

int j = log2(R - L + 1);

if(depth[spt[L][j]] < depth[spt[R-(1<<j)+1][j]])

return spt[L][j];

else return spt[R-(1<<j)+1][j];

}

void lca\_preprocess()

{

dfs\_counter = 0;

dfs(0, 0);

build(dfs\_counter-1);

}

int lca(int a, int b) {

int x, y;

x = dis\_time[a];

y = dis\_time[b];

if(x < y) return euler[query(x, y)];

else return euler[query(y, x)];

}

1. Lowest Common Ancestor [Binary Lifting]

//LCA using sparse table; Complexity: O(NlgN,lgN)

int depth[MX], par[MX], lca[MX][18];

void dfs(int from, int u, int h)

{

par[u] = from;

depth[u] = h;

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

int v = adj[u][i];

if(v != from) {

dfs(u, v, h+1);

}

}

}

void lca\_init(int N) //Complexity: O(NlogN)

{

dfs(0, 0, 0);

memset(lca, -1, sizeof lca);

int i, j;

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

lca[i][0] = par[i];

for (j = 1; (1 << j) <= N; j++)

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

if(lca[i][j-1] != -1)

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

}

int lca\_query(int p, int q) { //Complexity: O(logN)

int i, pow2;

if (depth[p] < depth[q]) swap(p, q);

pow2 = 1;

while(true) {

int next = pow2+1;

if((1<<next) > depth[p]) break;

else pow2++;

}

for (i = pow2; i >= 0; i--) {

if ((depth[p] - (1 << i)) >= depth[q])

p = lca[p][i];

}

if (p == q) return p;

for (i = pow2; i >= 0; i--)

if (lca[p][i] != -1 && lca[p][i] != lca[q][i])

p = lca[p][i], q = lca[q][i];

return par[p];

}

1. MST – Kruskal Algorithm

ll dsu[MX], mst;

vector <pair<ll, pair<ll, ll> > > edgeslist;

ll fnd(ll x) {

if(x == dsu[x]) return x;

return dsu[x] = fnd(dsu[x]);

}

//Complexity: O(ElogV), better for sparse graph.

void kruskal\_algo() {

for(int i = 0; i < MX; i++) dsu[i] = i;

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

mst = 0;

for(ll i = 0; i < edgeslist.size(); i++) {

ll w = edgeslist[i].ff, u = edgeslist[i].ss.ff,

v = edgeslist[i].ss.ss;

ll pu = fnd(u), pv = fnd(v);

if(pu != pv) {

dsu[pu] = pv;

mst += edgeslist[i].ff;

}

}

}

1. MST – Prim’s Algorithm

ll mst;

bool taken[MX];

vector <pll> adj[MX];

//Complexity: O(ElogV), better for dense graph

void prims\_algo(ll start) {

fill(taken, taken+MX, false);

priority\_queue <pll, vector<pll>, greater<pll> > pq;

pq.push(mk(0, start));

mst = 0;

while(!pq.empty())

{

ll u = pq.top().ss, w = pq.top().ff;

pq.pop();

*if(!taken[u]) {*

*taken[u] = true;*

*mst += w;*

*for(ll i = 0; i < adj[u].size(); i++) {*

*ll v = adj[u][i].ff, \_w = adj[u][i].ss;*

*if(!taken[v])*

*pq.push(mk( \_w, v ) );*

*}*

*}*

}

}

Graph Theory

1. Breadth First Search

void bfs(int s) { //Complexity: O(V+E)

memset(vis, false, sizeof vis);

memset(dist, inf, sizeof dist);

queue <int> q;

dist[s] = 0;

q.push(s);

while(!q.empty()) {

int u = q.top();

q.pop();

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

int v = adj[x][i];

if(!vis[v]) {

vis[v] = true;

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

q.push(v);

}

}

}

}

1. Depth First Search

void dfs(int u) { //Complexity : O(V+E)

vis[u] = true;

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

int v = adj[u][i];

if(!vis[v]) {

dfs(v);

}

}

}

1. Topological Sort

void dfs(int u) { //Complexity: O(V+E)

vis[u] = true;

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

int v = adj[u][i];

if(!vis[v]) {

dfs(v);

}

}

topsort.push\_back(u);

}

1. Topological Sort – Kahn’s Algorithm

void bfs() { //Complexity : O(V+E)

queue <int> pq;

//priority\_queue <int, vector <int>, greater<int> > pq;

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

if(in\_degree[i] == 0)

pq.push(i), level[i] = 0;

while(!pq.empty()) {

int u = pq.front(); //pq.top();

pq.pop();

topsort.pb(v);

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

int v = adj[u][i];

if(--in\_degree[v] == 0) q.push(v);

}

}

}

1. Bipartite Graph Check

/\* Bipartite graph has no odd cycle.

Bipartite graph can have at most V^2/4 edges. \*/

bool bipartite\_check(int src) { //Complexity : O(V+E)

memset(color, -1, sizeof color);

queue <int> q;

q.push(src);

color[src] = 0;

bool isBipartite = true;

while(!q.empty()) {

int u = q.front();

q.pop();

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

int v = adj[u][i];

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

color[v] = 1-color[u];

q.push(v);

}

else if(color[u] == color[v]) {

isBipartite = false;

break;

}

}

}

return isBipartite;

}

1. Articulation Point and Bridge

void APB(int u) {

dfs\_low[u] = dfs\_no[u] = dfs\_counter++;

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

int v = adj[u][i];

if(dfs\_no[v] == -1) {

dfs\_parent[v] = u;

if(u == dfsRoot) rootChildren++;

APB(v);

if(dfs\_low[v] >= dfs\_no[u]) //Articulation Points

articulation\_vertex[u] = true;

if(dfs\_low[v] > dfs\_no[u]) //Articulation Bridges

articulation\_bridge.pb(mk(u, v));

dfs\_low[u] = min(dfs\_low[u], dfs\_low[v]);

}

else if(v != dfs\_parent[u]) {

dfs\_low[u] = min(dfs\_low[u], dfs\_low[v]);

}

}

}

//Complexity: O(V+E)

void articulationPointandBridges(int n)

{

dfs\_counter = 0;

for(int i = 0; i < n; i++) { //0-based index

articulation\_vertex[i] = 0;

dfs\_low[i] = dfs\_parent[i] = 0;

dfs\_no[i] = -1;

}

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

if(dfs\_no[i] == -1) {

dfsRoot = i;

rootChildren = 0;

APB(i);

articulation\_vertex[dfsRoot] = (rootchildren>1);

}

}

//now check articulation\_vertex & articulation\_bridge

}

1. Strongly Connected Component

/\* SCC of undirected graph : Just use an extra track of parent in tarjanSCC function, don't do any operation whenever you go to the parent of a node. \*/

bool vis[MX];

vector <int> stck, adj[MX];

int numSCC, dfs\_low[MX], dfs\_no[MX], dfs\_counter;

void tarjanSCC(int u) {

vis[u] = true;

dfs\_low[u] = dfs\_no[u] = dfs\_counter++;

stck.pb(u);

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

int v = adj[u][i];

if(dfs\_no[v] == -1) tarjanSCC(v);

if(vis[v]) dfs\_low[u] = min(dfs\_low[u], dfs\_low[v]);

}

if(dfs\_low[u] == dfs\_no[u]) {

printf("SCC %d:", ++numSCC);

while(true) {

int v = stck.back();

stck.pop\_back();

vis[v] = 0;

printf(" %d", v);

if(u == v) break;

}

printf("\n");

}

}

//Complexity: O(V+E)

void tarjan(int n) { //n = Vertex Number

dfs\_counter = numSCC = 0;

for(int i = 0; i < n; i++) { //0-based index

dfs\_low[i] = vis[i] = 0;

dfs\_no[i] = -1;

}

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

if(dfs\_no[i] == -1)

tarjanSCC(i);

}

}

1. Tree Diameter

int mx, node;

bool vis[MX];

vector <int> adj[MX];

void dfs(int u, int h) {

vis[u] = true;

if(h > mx) mx = h, node = u;

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

int v = adj[u][i];

if(!vis[v])

dfs(v, h+1);

}

}

int main()

{

mx = 0;

memset(vis, 0, sizeof vis);

dfs(1, 0);

memset(vis, 0, sizeof vis);

dfs(node, 0);

DIAMETER = mx;

}

1. M-Coloring

bool isSafe(int u, int c) {

for(auto v : adj[u]) {

if(color[v] == c) return false;

}

return true;

}

bool M\_Coloring(int v, int m) {

if(v == n+1) return true;

//m = maximum color to be used

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

if(isSafe(v, i)) {

color[v] = i;

if(M\_Coloring(v+1, m)) return true;

color[v] = 0;

}

}

return false;

}

1. Dijkstra

void djk(int srcx)

{

fill(dist, dist+MX, inf);

fill(vis, vis+MX, false);

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

pq.push(mk(0, srcx));

dist[srcx] = 0;

while(!pq.empty()) {

auto u = pq.top();

pq.pop();

if(vis[u.ss]) continue;

vis[u.ss] = true;

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

auto v = adj[u.ss][i];

if(dist[v.ff] > dist[u.ss] + v.ss) {

dist[v.ff] = dist[u.ss] + v.ss;

pq.push(mk(dist[v.ff], v.ff));

}

}

}

}

1. Cycle finding in a graph

void dfs(int u) {

vis[u] = 1;

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

int v = adj[u][i];

if(vis[v] == 0) dfs(v);

else if(vis[v] == 1) return true;

}

vis[u] = 2;

return false;

}

1. Euler Tour [Euler Path & Euler Circuit]

Euler Circuit : starts from one node and visits every edge once and ends at the same starting node.

in undirected graph, every node must have even number of degrees. And in directed graph, number of indegree and outdegree of every node must be equal.

Euler Path : starts from one node and visits every edge once and ends in another node.

in undirected graph, every node (except start and finish node) must have even number of degrees. And in directed graph, every node (except start & finish node) must have equal number of in/out-degree. for start node, outdegree - indegree = 1 and for finish node, indegree – outdegree = 1.

Hierholzer’s algorithm for printing euler circuit / path:

Pseudocode:

tour\_stack = empty stack

find\_circuit(u):

for all edges u->v in G.adjacentEdges(v) do:

remove u->v

find\_circuit(v)

end for

tour\_stack.add(u)

return

Implementation:

void printEulerCircuit()

{

unordered\_map<int,int> edge\_count;

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

edge\_count[i] = adj[i].size();

}

vector<int> circuit;

stack<int> curr\_path;

int curr\_v = 0;

curr\_path.push(0);

while (!curr\_path.empty()) {

if (edge\_count[curr\_v]) {

curr\_path.push(curr\_v);

int next\_v = adj[curr\_v].back();

edge\_count[curr\_v]--;

adj[curr\_v].pop\_back();

curr\_v = next\_v;

}

else {

circuit.push\_back(curr\_v);

curr\_v = curr\_path.top();

curr\_path.pop();

}

}

for (int i=circuit.size()-1; i>=0; i--) {

cout << circuit[i];

if (i) cout<<" -> ";

}

}

1. Chinese Postman Problem

int perfect\_matching(int mask, int prev)

{

if(mask == (1<<odd\_vertices.size())-1) return 0;

int &ret = dp[mask][prev];

if(ret != -1) return ret;

int ans;

ret = inf;

for(int i = 0; i < odd\_vertices.size(); i++) {

if((mask & (1<<i)) == 0) {

ans = *perfect\_matching(mask | (1<<i),*

*odd\_vertices[i]);*

if(\_\_builtin\_popcount(mask) % 2)

ans += mat[prev][odd\_vertices[i]];

ret = min(ret, ans);

}

}

return ret;

}

//Complexity: floyd\_warshall[O(n^3)] + perfect\_matching[O(mask\*prev)]

void chinese\_postman\_problem(int sum)

{

odd\_vertices.clear();

find\_odd(); // all odd nodes in odd\_vertices vector

floyd\_warshall();

memset(dp, -1, sizeof dp);

ans = sum + perfect\_matching(0, 0);

//sum = 'sum of the all the edges in the graph'

}

String Processing

1. String Hashing

struct simplehash {

int len;

long long base, mod;

vector <int> P, H, R;

/\* P = Powers of the base, H = Hash value, R = Reverse hash value. hash = str[0]\*P[n-1] + str[1]\*P[n-2] + .... + str[n-1]\*P[0] \*/

simplehash() {}

simplehash(const char\* str, ll b, ll m) {

base = b, mod = m, len = strlen(str);

P.resize(len + 3, 1), H.resize(len + 3, 0),

R.resize(len + 3, 0);

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

P[i] = (P[i - 1] \* base) % mod;

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

H[i] = (H[i - 1] \* base + str[i - 1] + 1007) % mod;

for (int i = len; i >= 1; i--)

R[i] = (R[i + 1] \* base + str[i - 1] + 1007) % mod;

}

inline int range\_hash(int l, int r) {

int hashval = H[r + 1] - ((long long)P[r - l + 1] \*

H[l] % mod);

return (hashval < 0 ? hashval + mod : hashval);

}

inline int reverse\_hash(int l, int r){;

int hashval = R[l + 1] - ((long long)P[r - l + 1] \*

R[r+ 2] % mod);

return (hashval < 0 ? hashval + mod : hashval);

}

};

struct stringhash {

simplehash sh1, sh2;

stringhash () {}

stringhash (const char\* str) {

sh1 = simplehash (str, 1949313259, 2091573227);

sh2 = simplehash (str, 1997293877, 2117566807);

}

inline long long range\_hash(int l, int r) {

return ( (long long) sh1.range\_hash(l, r) << 32) ^

sh2.range\_hash(l, r);

}

inline long long reverse\_hash(int l, int r){

return ( (long long) sh1.reverse\_hash(l, r) << 32) ^

sh2.reverse\_hash(l, r);

}

};

1. Knuth-Morris-Pratt (KMP) Algorithm

int lps[MX];

//Longest Prefix Subarray

void failure(string &pat, int M) {

int i = 1, len = 0;

lps[0] = 0;

while (i < M) {

if (pat[i] == pat[len]) {

len++;

lps[i] = len;

i++;

}

else {

if (len != 0) {

len = lps[len - 1];

}

else {

lps[i] = 0;

i++;

}

}

}

}

void KMPSearch(string &pat, string &txt)

{

int M = pat.size();

int N = txt.size();

int i = 0, j = 0;

while (i < N) {

if (pat[j] == txt[i]) {

j++;

i++;

}

if (j == M) {

printf("Found pattern at index %d ", i - j);

j = lps[j - 1];

}

else if (i < N && pat[j] != txt[i]) {

if (j != 0) j = lps[j - 1];

else i++;

}

}

}

1. TRIE [pointer implementation]

const int ALPHABET\_SIZE = 26;

typedef struct TrieNode {

struct TrieNode \*children[ALPHABET\_SIZE];

bool isEndOfWord;

} TrieNode;

struct TrieNode \*getNode(void) {

TrieNode \*pNode = new TrieNode;

pNode->isEndOfWord = false;

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

pNode->children[i] = NULL;

return pNode;

}

void insert(TrieNode \*root, string &key) {

TrieNode \*pCrawl = root;

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

int index = key[i] - 'a';

if (!pCrawl->children[index])

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

pCrawl = pCrawl->children[index];

}

pCrawl->isEndOfWord = true;

}

bool search(TrieNode \*root, string &key) {

TrieNode \*pCrawl = root;

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

int index = key[i] - 'a';

if (!pCrawl->children[index])

return false;

pCrawl = pCrawl->children[index];

}

return (pCrawl != NULL && pCrawl->isEndOfWord);

}

void del(TrieNode \*cur) //for destruction of whole trie

{

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

if (cur->children[i])

del(cur->children[i]);

delete (cur);

}

1. TRIE [2D-array implementation]

#define MX\_LEN 100

#define MX\_NODE 100000

#define alphabet\_size 26

char S[MX\_LEN];

int root, nnode;

int isWord[MX\_NODE];

int node[MX\_NODE][alphabet\_size];

void initailize() {

root = 0;

nnode = 0;

for(int i = 0; i < alphabet\_size; i++)

node[root][i] = -1;

}

void insert() {

scanf("%s", S);

int now, len, index;

len = strlen(S);

now = root;

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

index = s[i]-'a';

if(node[now][index] == -1) {

node[now][index] = ++nnode;

for(int j = 0; j < alphabet\_size; j++)

node[nnode][j] = -1;

}

now = node[now][index];

}

isWord[now] = 1;

}

Algebra

1. Matrix Exponent

class matrix {

public:

int mat[2][2];

int row, col;

matrix() {

memset(mat, 0, sizeof mat);

}

matrix(int r, int c) {

row = r, col = c;

memset(mat, 0, sizeof mat);

}

//Complexity: O(N^3)

matrix operator\* (matrix &p) {

matrix temp;

temp.row = row;

temp.col = p.col;

int sum;

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

for(int j = 0; j < temp.col; j++) {

sum = 0;

for(int k = 0; k < col; k++) {

sum += mat[i][k] \* p.mat[k][j];

//sum = (sum + (((ll) mat[i][k] \*

p.mat[k][j]) % mod)) % mod;

}

temp.mat[i][j] = sum;

}

}

return temp;

}

//Complexity: O(N^2)

matrix operator+ (matrix &p) {

matrix temp;

temp.row = row;

temp.col = col;

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

for(int j = 0; j < temp.col; j++) {

temp.mat[i][j] = mat[i][j] + p.mat[i][j];

}

}

return temp;

}

//square matrix with 1s in LR-diagonal

matrix identity() {

matrix temp;

temp.row = row;

temp.col = col;

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

temp.mat[i][i] = 1;

return temp;

}

//Complexity: O(N^3 \* logPow)

matrix pow(int pow) {

matrix temp = (\*this);

matrix ret = (\*this).identity();

while(pow > 0) {

if(pow%2 == 1)

ret = ret \* temp;

temp = temp \* temp;

pow /= 2;

}

return ret;

}

void show() {

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

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

printf("%d ", mat[i][j]);

}

printf("\n");

}

}

};

1. Discrete Logarithm, Primitive Root,

Discrete Root

ll primitive\_root(ll p) { //Complexity: O((logP)^6)

vector<int> prime\_factors;

ll phi = p-1, n = phi;

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

if (n % i == 0) {

prime\_factors.push\_back (i);

while (n % i == 0)

n /= i;

}

}

if (n > 1) prime\_factors.push\_back (n);

//Complexity: O(p\*sqrt(phi(p)))

for (ll res = 2; res <= p; res++) {

bool ok = true;

for (ll i = 0; i<prime\_factors.size() && ok; i++) {

ll x = bigmod(res, phi / prime\_factors[i], p);

if(x == 1) ok = false;

}

if (ok) return res;

}

return -1;

}

//Complexity: O(sqrt(m)\*log(m))

ll discrete\_log(ll a, ll b, ll m) {

ll n = sqrt (m) + 1;

ll an = 1;

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

an = (an \* a) % m;

map<int, int> vals; //or use ‘hash\_table’

for (ll p = 1, cur = an; p <= n; ++p) {

if (!vals.count(cur)) vals[cur] = p;

cur = (cur \* an) % m;

}

for (ll q = 0, cur = b; q <= n; ++q) {

if (vals.count(cur)) {

ll ans = vals[cur] \* n - q;

return ans;

}

cur = (cur \* a) % m;

}

return -1;

}

ll discrete\_root(ll a, ll b, ll m) {

ll g = primitive\_root(m);

ll x = discrete\_log(bigmod(g, a, mod), b, m);

if(x != -1) {

x = bigmod(g, x, m);

}

return x;

/\* Print all possible answers

ll delta = (m-1) / \_\_gcd(k, m-1);

vector<ll> ans;

for (ll cur = any\_ans%delta; cur < m-1; cur+=delta)

ans.push\_back(bigmod(g, cur, m));

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

printf("%d\n", ans.size());

for (int answer : ans)

printf("%lld ", answer); \*/

}

Miscellaneous

1. Longest Increasing Subsequence [N\*logN]

int lis(int n) {

int tail[n];

tail[0] = a[0];

int length = 1;

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

if(a[i] > tail[length-1]) { //strictly greater

tail[length++] = a[i];

}

else {

auto it = lower\_bound(tail, tail+length, a[i]);

\*it = a[i];

}

}

return length;

}

1. String Multiply

string multiply(string s, ll a) {

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

ll carry = 0;

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

carry += a \* (s[i]-48);

s[i] = (carry % 10 + 48);

carry /= 10;

}

while(carry) {

s += (carry % 10 + 48);

carry /= 10;

}

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

return s;

}

1. LIS [N2]

#define MX 100005

int n, arr[MX], dp[MX];

/\* int dir[MX] ; to print solution \*/

int dfs(int pos) {

if(dp[pos] != -1) return dp[pos];

dp[pos] = 0;

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

if(arr[i] >= arr[pos]) {

dp[pos] = max(dp[pos], dfs(i));

/\* when maximum, dir[pos] = i \*/

}

}

return dp[pos] += 1;

}

int lcs() {

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

int longest = 0;

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

longest = max(longest, dfs(i));

}

return longest;

}