1. **Data Structure**
   1. **Size Balanced Tree**

struct SBTTree { int l, r, s, key; };

void leftRotate(int &x) {

int y = a[x].r; a[x].r = a[y].l; a[y].l = x;

a[y].s = a[x].s;

a[x].s = a[a[x].l].s + a[a[x].r].s + 1;

// renew(x); renew(y);

x = y;

}

void righRotate(int &x) {

int y = a[x].l; a[x].l = a[y].r; a[y].r = x;

a[y].s = a[x].s;

a[x].s = a[a[x].l].s + a[a[x].r].s + 1;

// renew(x); renew(y);

x = y;

}

void maintain(int &x, bool fl) {

if (!fl)

if (a[a[a[x].l].l].s > a[a[x].r].s) righRotate(x); else

if (a[a[a[x].l].r].s > a[a[x].r].s) leftRotate(a[x].l), righRotate(x);

else return;

else

if (a[a[a[x].r].r].s > a[a[x].l].s) leftRotate(x); else

if (a[a[a[x].r].l].s > a[a[x].l].s) righRotate(a[x].r), leftRotate(x);

else return;

maintain(a[x].l, 0);

maintain(a[x].r, 1);

maintain(x, 0);

maintain(x, 1);

}

void insert(int &x, int n) {

if (!x) {

a[n].l = a[n].r = 0; a[n].s = 1;

x = n;

return;

}

++ a[x].s;

if (a[n].key < a[x].key) insert(a[x].l, n); else insert(a[x].r, n);

// renew(x);

maintain(x, a[n].key >= a[x].key);

}

void delett(int &x, int n) {

// if (!x) return;

if (x == n) {

if (!a[x].l || !a[x].r) {

x = a[x].l + a[x].r;

return;

}

righRotate(x); -- a[x].s;

delett(a[x].r, n);

// renew(x);

return;

}

-- a[x].s;

if (a[n].key < a[x].key) delett(a[x].l, n); else delett(a[x].r, n);

// renew(x);

}

int findKth(int x, int k) {

if (k < 1 || k > a[x].s) return -1;

for (; ; )

if (k == a[a[x].l].s + 1) return x; else

if (k <= a[a[x].l].s) x = a[x].l; else

k -= a[a[x].l].s + 1, x = a[x].r;

}

* 1. **Splay**

struct SPLTree { int l, r, p, s, key; };

void zig(int x) {

int y = a[x].p, z = a[y].p, w = a[x].l;

a[w].p = y; a[y].r = w;

a[y].p = x; a[x].l = y;

a[x].p = z;

if (y == a[z].l) a[z].l = x;

if (y == a[z].r) a[z].r = x;

// renew(y); renew(x);

}

void zag(int x) {

int y = a[x].p, z = a[y].p, w = a[x].r;

a[w].p = y; a[y].l = w;

a[y].p = x; a[x].r = y;

a[x].p = z;

if (y == a[z].l) a[z].l = x;

if (y == a[z].r) a[z].r = x;

// renew(y); renew(x);

}

void splay(int x) {

// int la = 0;

// for (int i = x; ; i = a[i].p) {

// que[++ la] = i;

// if (!a[i].p) break;

// }

// for (int i = la; i; -- i) updata(que[i]);

for (; a[x].p; ) {

int y = a[x].p, z = a[y].p;

if (!z)

if (x == a[y].l) zag(x); else zig(x);

else

if (x == a[y].l)

if (y == a[z].l) zag(y), zag(x); else zag(x), zig(x);

else

if (y == a[z].r) zig(y), zig(x); else zig(x), zag(x);

}

}

void cut(int x) {

if (!x) return;

int y = a[x].p;

if (x == a[y].l) a[y].l = 0; else a[y].r = 0;

a[x].p = 0;

// renew(y);

}

int join(int p, int q) {

if (!p) return q;

if (!q) return p;

int x = p;

for (; a[x].r; x = a[x].r);// updata(x);

splay(x);

a[q].p = x; a[x].r = q;

// renew(x);

return x;

}

* 1. **Link Cut Tree**

bool isRoot(int x) {

int y = a[x].p;

if (!y) return 1;

if (a[y].l != x && a[y].r != x) return 1;

return 0;

}

void zig(int x);

void zag(int x);

void splay(int x) {

// ……

for (; !isRoot(x); ) {

int y = a[x].p, z = a[y].p;

if (isRoot(y))

if (a[y].l == x) zag(x); else zig(x);

else

if (a[y].l == x)

if (a[z].l == y) zag(y), zag(x); else zag(x), zig(x);

else

if (a[z].r == y) zig(y), zig(x); else zig(x), zag(x);

}

}

void access(int x) {

for (int y = 0; x; x = a[x].p) {

splay(x);

a[x].r = y;

// renew(x);

y = x;

}

}

int getRoot(int x) {

access(x);

splay(x);

for (; ; x = a[x].l) if (!a[x].l) return x;

}

int getLca(int w, int x) {

if (w == x) return w;

if (getRoot(w) != getRoot(x)) return -1;

access(w);

for (int y = 0; x; x = a[x].p) {

splay(x);

a[x].r = y;

// renew(x);

if (!a[x].p) return x;

y = x;

}

}

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

access(v); splay(v);

a[v].p = u;

// renew(v); renew(u);

access(v);

}

void cutt(int u) {

access(u); splay(u);

a[a[u].l].p = 0; a[u].l = 0;

// renew(u);

}

1. **Flow**
   1. **Maxflow**

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

w[++ W].v = v; w[W].c = c; w[W].next = ww[u]; ww[u] = W;

w[++ W].v = u; w[W].c = 0; w[W].next = ww[v]; ww[v] = W;

}

int build() {

int fi, la;

memset(dist, 0, sizeof(dist));

dist[que[la = 1] = s] = 1;

for (fi = 1; fi <= la; ++ fi) {

int u = que[fi];

for (int i = ww[u]; i; i = w[i].next) if (w[i].c) {

int v = w[i].v;

if (dist[v]) continue;

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

que[++ la] = v;

if (v == t) return 1;

}

}

return 0;

}

void dinic(int u) {

if (u == t) {

int flow = 1 << 30;

for (int i = t; i != s; i = w[stq[i] ^ 1].v) flow = min(flow, w[stq[i]].c);

for (int i = t; i != s; i = w[stq[i] ^ 1].v) {

w[stq[i]].c -= flow; w[stq[i] ^ 1].c += flow;

if (!w[stq[i]].c) off = w[stq[i] ^ 1].v;

}

maxflow += flow;

return;

}

for (int i = ww[u]; i; i = w[i].next) if (w[i].c) {

int v = w[i].v;

if (dist[v] != dist[u] + 1) continue;

stq[v] = i; dinic(v);

if (dist[u] > dist[off]) return;

off = t;

}

dist[u] = -1;

}

* 1. **Costflow**

void addedge(int u, int v, int c, int q) {

w[++ W].v = v; w[W].c = c; w[W].q = q; w[W].next = ww[u]; ww[u] = W;

w[++ W].v = u; w[W].c = 0; w[W].q = -q; w[W].next = ww[v]; ww[v] = W;

}

int mcmf() {

int mincost = 0;

for (int la; ; ) {

memset(dist, 60, sizeof(dist));

// memset(dist, -60, sizeof(dist));

memset(visit, 0, sizeof(visit));

dist[s] = 0; visit[s] = 1; que[la = 1] = s;

for (int fi = 1; fi <= la; ++ fi) {

int u = que[fi];

for (int i = ww[u]; i; i = w[i].next) if (w[i].c) {

int v = w[i].v;

if (dist[v] <= dist[u] + w[i].q) continue;

// if (dist[v] >= dist[u] + w[i].q) continue;

dist[v] = dist[u] + w[i].q; stq[v] = i;

if (visit[v]) continue;

visit[v] = 1; que[++ la] = v;

}

visit[u] = 0;

}

if (dist[t] > 1 << 29) break;

// if (dist[t] <= 0) break;

// if (dist[t] < -1 << 29) break;

int flow = 1 << 30;

for (int i = t; i != s; i = w[stq[i] ^ 1].v) flow = min(flow, w[stq[i]].c);

for (int i = t; i != s; i = w[stq[i] ^ 1].v)

w[stq[i]].c -= flow, w[stq[i] ^ 1].c += flow;

mincost += dist[t] \* flow;

}

return mincost;

}

1. **Graph**
   1. **Tarjan for cut point/cut edge**

void Tarjan\_cut(int u) {

s[u].low = s[u].dfn = ++ DFN;

stk[++ STK] = u; s[u].in = 1;

for (int i = ww[u]; i; i = w[i].next) {

int v = w[i].v;

if (!s[v].dfn) {

Tarjan\_cut(v);

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

// if (s[v].low >= s[u].dfn) s[u].cut = 1;

// if (s[v].low > s[u].dfn) w[i].cut = 1;

} else

if (s[v].in) s[u].low = min(s[u].low, s[v].dfn);

}

if (s[u].low == s[u].dfn)

for (++ BLOCK; stk[STK + 1] != u; -- STK) {

int v = stk[STK];

s[v].in = 0;

s[v].block = BLOCK;

}

}

* 1. **Dijkstra optimized by heap**

void heapUp(int u) {

for (int i = p[u], j; i >> 1; i = j) {

j = i >> 1;

if (dist[h[i]] >= dist[h[j]]) return;

swap(p[h[i]], p[h[j]]);

swap(h[i], h[j]);

}

}

void heapDeleteHead() {

h[1] = h[H --]; p[h[1]] = 1;

for (int i = 1, j; i << 1 <= H; i = j) {

j = i << 1;

if (j < H && dist[h[j + 1]] < dist[h[j]]) ++ j;

if (dist[h[i]] <= dist[h[j]]) return;

swap(p[h[i]], p[h[j]]);

swap(h[i], h[j]);

}

}

void dijkstra() {

memset(dist, 60, sizeof(dist)); dist[1] = 0;

for (int i = 1; i <= N; ++ i) h[i] = p[i] = i;

H = N;

for (int times = 0; times < N; ++ times) {

if (!H) break;

int u = h[1];

heapDeleteHead();

for (int i = ww[u]; i; i = w[i].next) {

int v = w[i].v;

if (dist[v] <= dist[u] + w[i].c) continue;

dist[v] = dist[u] + w[i].c;

heapUp(v);

}

}

}

1. **Matching**
   1. **KM**

int hungary(int u) {

X[u] = 1;

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

if(!Y[i] && lx[u] + ly[i] == maq[u][i]) {

Y[i] = 1;

if(linky[i] == 0 || Find(linky[i])) {

linky[i] = u;

return 1;

}

}

return 0;

}

void KM() {

memset(linky, 0, sizeof(linky));

memset(lx, 0, sizeof(lx));

// memset(lx, 60, sizeof(lx));

memset(ly, 0, sizeof(ly));

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

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

lx[i] = max(lx[i], maq[i][j]);

// lx[i] = min(lx[i], maq[i][j]);

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

for (; ; ) {

memset(X, 0, sizeof(X));

memset(Y, 0, sizeof(Y));

if (hungary(k)) break;

d = 1 << 30;

// d = -1 << 30;

for (i = 1; i <= N; ++ i) if (X[i])

for (j = 1; j <= N; ++ j) if (!Y[j])

d = min(d, lx[i] + ly[j] - maq[i][j]);

// d = max(d, lx[i] + ly[j] - maq[i][j]);

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

if (X[i]) lx[i] -= d;

if (Y[i]) ly[i] += d;

}

}

}

1. **String**
   1. **KMP**

next[1] = 0;

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

for (j = next[i - 1]; j > 0 && str[j + 1] != str[i]; j = next[j]);

if (str[j + 1] == str[i]) next[i] = j + 1; else next[i] = 0;

}

* 1. **Extend KMP**

memset(A, 0, sizeof(A));

for (int i = 2, k = 1; i <= N; ++ i) {

int maxLen = k + A[k] - 1, j = i – k + 1;

if (i + A[j] – 1 < maxLen) A[i] = A[j]; else {

int p = max(0, maxLen – i + 1);

for (; str[i + p] == str[1 + p]; ++ p);

A[i] = p; k = i;

}

}

* 1. **Extend KMP for palindrome strings**

memset(A, 0, sizeof(A));

for (int i = 2, k = 1; i <= N; ++ i) {

int maxLen = k + A[k], j = k - (i - k);

if (i <= maxLen) A[i] = min(A[j], maxLen - i); else A[i]=0;

if (i + A[i] >= maxLen) {

for (; str[i - A[i]] == str[i + A[i]]; ) ++ A[i];

-- A[i]; k = i;

}

}

* 1. **Suffix array**

void suffixsort() {

int T = max(N, 26);

memset(top, 0, sizeof(top));

for (int i = 1; i <= N; ++ i) top[rank[i] = a[i] + 1] ++;

for (int i = 1; i <= T; ++ i) top[i] += top[i - 1];

for (int i = 1; i <= N; ++ i) sa[top[rank[i]] --] = i;

for (int L = 1; L < N; L <<= 1) {

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

int j = sa[i] - L;

if (j <= 0) j += N;

nsa[++ top[rank[j]]] = j;

}

top[1] = 0;

for (int i = 2, j = nrank[nsa[1]] = 1; i <= N; ++ i) {

if (rank[nsa[i]] != rank[nsa[i - 1]] ||

rank[nsa[i] + L] != rank[nsa[i - 1] + L]) top[++ j] = i - 1;

nrank[nsa[i]] = j;

}

memcpy(rank, nrank, sizeof(rank));

memcpy(sa, nsa, sizeof(sa));

if (rank[sa[N]] == N) break;

}

for (int i = 1, p = 0; i <= N; ++ i)

if (rank[i] == 1) h[rank[i]] = 0; else {

p = max(0, p - 1);

for (int j = sa[rank[i] - 1]; a[i + p] == a[j + p]; ++ p);

h[rank[i]] = p;

}

}

1. **Mathematic**
   1. **Extend GCD**

*Calculate integer solution for AX+BY=C*

void exGCD(int A, int B, int C, int &X, int &Y) {

if (!B) {

X = C / A; Y = 0;

return;

}

exGCD(B, A % B, C, X, Y);

int S = Y, T = X - A / B \* Y;

X = S; Y = T;

}

* 1. **Extend GCD\***

*Calculate integer solution for*

int exGCD2(int N, int A, int B, int C) {

if (!N || !C) return 0;

int T = A / C \* N + B / C \* (N - 1) \* N / 2;

A %= C; B %= C;

return T + exGCD2((A + B \* N) / C, (A + B \* N) % C, C, B);

}

* 1. **Gauss Elim**

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

for (int j = i; j <= N; ++ j) if (a[j][i] != 0) break;

for (int k = i; k <= N + 1; ++ k) swap(a[i][k], a[j][k]);

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

int t = a[j][i] / a[i][i];

for (int k = i; k <= N + 1; ++ k) a[j][k] -= a[i][k] \* t;

}

}

for (int i = N; i; -- i) {

int t = a[i][N + 1];

for (int j = N; j > i; -- j) t -= ans[j] \* a[i][j];

ans[i] = t / a[i][i];

}

* 1. **Prime algorithm in O(N)**

memset(next, 0, sizeof(next));

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

if (!next [i]) prime[++ PRIME] = i, next[i] = i;

for (int j = 1; j <= PRIME && i \* prime[j] <= N; ++ j) {

next[i \* prime[j]] = prime[j];

if (i % prime[j] == 0) break;

}

}

* 1. **Minimum expression**

int minimumExpression(char st[],int len) {

int i = 0, j = 1, k = 0;

while (k < len)

if (st[(i + k) % len] == st[(j + k) % len]) ++ k; else {

if (st[(i + k) % len] > st[(j + k) % len]) i += k + 1; else j += k + 1;

if (i == j) ++ j;

k = 0;

}

return (i % len < j % len ? i % len : j % len);

}

* 1. **Simpson**

dobule f(l, r) {

double m = (l + r) / 2.0;

return ( f(l) + f(m) \* 4.0 + f(r) ) \* (r - l) / 6.0;

}

double simpson(l, r) {

double m = (l + r) / 2.0;

if (fabs(f(l, m) + f(m, r) - f(l, r)) < eps) return f(l, m) + f(m, r);

return simpson(l, m) + simpson(m, r)

}

* 1. **Inverse element**

void getInv(int N,int mod) {

inv[1] = 1;

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

inv[i] = ((-(int64)mod / i \* inv[mod % i]) % mod + mod) % mod;

}

* 1. **Miller Rabin**

*Judge if N is a prime*

const int prime[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47};

int64 producMult(int64 A, int64 B, int64 C) {

int64 ANS = 0, P = A;

for (; B; B >>= 1) {

if (B & 1) if ((ANS += P) >= C) ANS -= C;

if ((P += P) >= C) P -= C;

}

return ANS;

}

int64 producPower(int64 A, int64 B, int64 C) {

int64 ANS = 1, P = A;

for (; B; B >>= 1) {

if (B & 1) ANS = producMult(ANS, P, C);

P = producMult(P, P, C);

}

return ANS;

}

bool Witness(int64 A, int64 N) {

if (N == 1) return 1;

int64 TIMES = 0, NN = N - 1;

for (; !(NN & 1); NN >>= 1) ++ TIMES;

int64 X, Y = producPower(A, NN, N), Z;

for (; TIMES--; ) {

Z = producMult(Y, Y, N); X = Y; Y = Z;

if (Y == 1 && X != 1 && X != N - 1) return 0;

}

if (Y != 1) return 0;

return 1;

}

bool MillerRabin(int64 N) {

for (int i = 0; i < 15; ++ i) if (N == prime[i]) return 1;

for (int i = 0; i < 15; ++ i) if (!Witness(prime[i], N)) return 0;

return 1;

}

* 1. **Pollard Rho**

*Find factors of N*

int64 PollarRho(int64 N) {

int64 x = rand() % N, y = x;

for (int i = 2, k = 2; ; ++ i) {

x = producMult(x, x, N);

if (!x) x = N - 1; else -- x;

if (x == y) return N;

int64 GCD = gcd(N + y - x, N);

if (GCD > 1 && GCD < N) return GCD;

if (i == k) y = x, k += k;

}

}

void findFactor(int64 N) {

if (N <= smallCase) {

specialJudge(N);

return;

}

if (MillerRabin(N)) { factor[++ FACTOR]=N; return; }

int64 Q = N;

for (; Q == N || Q == 1; ) Q = PollardRho(N);

findFactor(Q);

findFactor(N / Q);

}