UNAL ICPC Team Notebook (2024)

Contents

hola mUNdo

1 Data Structures

1.1 DSU

```
const int N = 1e5+5;
int dsu[N];
int cc;

int find (int node){
    if(dsu[node] == -1) return node;
    return dsu[node] = find(dsu[node]);
}

bool connected(int A, int B){
    return find(A)==find(B);
}

void join (int A, int B){
    A = find(A);
    B = find(B);
    dsu[A] = B;
    cc--;
}

memset(dsu, -1, sizeof dsu);
```

1.2 DSU Pesos

```
int parent[MAX];
int rango[MAX];
int n;
void Init( int _n ){
   n = _n;
   for( int i = 0 ; i < n ; ++i ){
       parent[i] = i;
       rango[i] = 0;
   }
}

int Find( int x ){
   if( x == parent[ x ] )
       return x;
   else
       return parent[ x ] = Find( parent[ x ] );
}
void Union( int x , int y ){
   int xRoot = Find( x );
   int yRoot = Find( y );
   if( rango[ xRoot ] > rango[ yRoot ] )
       parent[ yRoot ] = xRoot;
   else{
       parent[ xRoot ] = yRoot;
       if( rango[ xRoot ] == rango[ yRoot ] )
           rango[ yRoot ]++;
   }
}
int countComponents(){
   int c = 0;
```

```
    for( int i=0; i<n; i++ )
        if( parent[i] == i )
            c++;
    return c++;
}
vector<int> getRoots(){
    vector<int> v;
    for( int i=0; i<n; i++ )
        if( i == parent[i] )
            v.push_back(i);
    return v;
}

int countNodesInComponent( int root ){
    int c = 0;
    for( int i=0; i<n; i++)
        if( Find(i) == root )
            c++;
    return c++;
}
bool sameComponent( int x, int y ){
    return Find(x) == Find(y);
}
```

2 Graphs

2.1 Strongest Connected components

```
vector<bool> visited; // keeps track of which vertices are
    already visited

// runs depth first search starting at vertex v.
// each visited vertex is appended to the output vector when
    dfs leaves it.
void dfs(int v, vector<vector<int>> const& adj, vector<int> &
    output) {
    visited[v] = true;
    for (auto u : adj[v])
        if (!visited[u])
            dfs(u, adj, output);
    output.push_back(v);
}

// input: adj -- adjacency list of G
// output: components -- the strongy connected components in G
// output: adj_cond -- adjacency list of G^SCC (by root
    vertices)
void strongly_connected_components(vector<vector<int>> const&
    adj,
                                   vector<vector<int>> &
                                       components,
                                   vector<vector<int>> &
                                       adj_cond) {
    int n = adj.size();
    components.clear(), adj_cond.clear();

    vector<int> order; // will be a sorted list of G's
        vertices by exit time

    visited.assign(n, false);

    // first series of depth first searches
    for (int i = 0; i < n; i++)
        if (!visited[i])
            dfs(i, adj, order);

    // create adjacency list of G^T
    vector<vector<int>> adj_rev(n);
    for (int v = 0; v < n; v++)
        for (int u : adj[v])
            adj_rev[u].push_back(v);

    visited.assign(n, false);
    reverse(order.begin(), order.end());

    vector<int> roots(n, 0); // gives the root vertex of a
        vertex's SCC

    // second series of depth first searches
    for (auto v : order)
        if (!visited[v]) {
            std::vector<int> component;
            dfs(v, adj_rev, component);
            components.push_back(component);
            int root = *min_element(begin(component), end(
                component));
            for (auto u : component)
                roots[u] = root;
        }

    // add edges to condensation graph
    adj_cond.assign(n, {});
    for (int v = 0; v < n; v++)
        for (auto u : adj[v])
            if (roots[v] != roots[u])
                adj_cond[roots[v]].push_back(roots[u]);
}
```

2.2 SCC Tarjan

```
struct TarjanScc{
    vector<bool> marked;
    vector<int> id;
    vector<int> low;
    int pre;
    int count;
    stack<int> stck;
    vector<vector<int> >G;

    TarjanScc( vector<vector<int> >g, int V ){
        G=g;
        marked = vector<bool>(V, false);
        stck = stack<int>();
        id= low = vector<int>(V, 0);
        pre=count=0;
        for(int u=0; u<V; u++)
            if( !marked[u] ) dfs(u);
    }

    void dfs( int u ){
        marked[ u ] = true;
        low[ u ] = pre++;
        int min = low[ u ];

        stck.push( u );
        for( int w=0; w<G[u].size(); w++){
            if(  !marked[G[u][w]] ) dfs( G[u][w] );
```

```
            if( low[ G[u][w] ] < min ) min = low[ G[u][w] ];
        }
        if( min<low[u] ){
            low[u] = min;
            return;
        }
        int w;
        do{
            w = stck.top();stck.pop();
            id[ w ] = count;
            low[ w ]= G.size();
        }while( w != u );
        count++;
    }

    int getCount() { return count; }

    // are v and w strongly connected?
    bool stronglyConnected(int v, int w) {
        return id[v] == id[w];
    }

    // in which strongly connected component is vertex v?
    int getId(int v) { return id[v]; }
};

//Ejemplo de Uso
int main( ){
    int u, v, N, M, cas, k=0;
    for(cin>>cas; k<cas; k++){
        scanf("%d %d", &N, &M);
        //cin>>N>>M;
        vector<vector<int> >G(N);

        for(int i=0; i < M; i++){
            scanf("%d %d", &u, &v);
            //cin>>u>>v;
            u--;v--;
            G[u].PB(v);
        }

        TarjanScc tscc(G, N);
        //Encontrar cuantos nodos tienen grado de entrada 0
        vector<int>indegree(tscc.getCount(), 0);
        int idu, idv;
        for( u = 0; u < N; u++){
            idu = tscc.getId( u );
            for( v = 0; v < G[u].size(); v++){
                idv = tscc.getId( G[u][v] );

                if( idu!=idv ){
                    indegree[idv]++;
                }
            }
        }
        int res=0;
        for(int i=0; i<indegree.size(); i++){
            if(indegree[i]==0)res++;
        }
        printf("Case %d: %d\n",k+1,res);
    }
    return 0;
}
```

2.3 Topological sort

```
int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
    ans.push_back(v);
}

void topological_sort() {
    visited.assign(n, false);
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    reverse(ans.begin(), ans.end());
}
```

2.4 Floyd-Warshall

```
//O(n^3)

//inicializar todo en INF previo a la lectura

for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}

//Si se tienen pesos negativos:
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (d[i][k] < INF && d[k][j] < INF)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}

//Pesos reales
if (d[i][k] + d[k][j] < d[i][j] - EPS)
    d[i][j] = d[i][k] + d[k][j];

/*Identificar ciclos negativos:
Si al final del algoritmo d[i][i] es negativo.*/
```

2.5 Dijkstra

```
//O(n^2+m)
```

```cpp
    for (int i = 1; i <= n; i++) distance[i] = INF;
    distance[x] = 0;
    q.push({0,x});
    while (!q.empty()) {
        int a = q.top().second; q.pop();
        if (processed[a]) continue;
        processed[a] = true;
        for (auto u : adj[a]) {
            int b = u.first, w = u.second;
            if (distance[a]+w < distance[b]) {
                distance[b] = distance[a]+w;
                q.push({-distance[b],b});
            }
        }
    }
```

2.6 Shortest Path Fast algorithm

```cpp
//O(nm)
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

bool spfa(int s, vector<int>& d) {
    int n = adj.size();
    d.assign(n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inqueue(n, false);
    queue<int> q;

    d[s] = 0;
    q.push(s);
    inqueue[s] = true;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        inqueue[v] = false;

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                if (!inqueue[to]) {
                    q.push(to);
                    inqueue[to] = true;
                    cnt[to]++;
                    if (cnt[to] > n)
                        return false;  // negative cycle
                }
            }
        }
    }
    return true;
}
```

3 Dynamic Programming

3.1 Coin Exchange Problem

```cpp
#include <bits/stdc++.h>

using namespace std;

// Returns total distinct ways to make sum using n coins of
// different denominations
int count(vector<int>& coins, int n, int sum)
{
    // 2d dp array where n is the number of coin
    // denominations and sum is the target sum
    vector<vector<int> > dp(n + 1, vector<int>(sum + 1, 0));

    // Represents the base case where the target sum is 0,
    // and there is only one way to make change: by not
    // selecting any coin
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= sum; j++) {

            // Add the number of ways to make change without
            // using the current coin,
            dp[i][j] += dp[i - 1][j];

            if ((j - coins[i - 1]) >= 0) {

                // Add the number of ways to make change
                // using the current coin
                dp[i][j] += dp[i][j - coins[i - 1]];
            }
        }
    }
    return dp[n][sum];
}

// Driver Code
int main()
{
    vector<int> coins{ 1, 2, 3 };
    int n = 3;
    int sum = 5;
    cout << count(coins, n, sum);
    return 0;
}
```

4 Flows

4.1 Dinic

```cpp
struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(
        cap) {}
};

struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;
```

```
    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1)
                    continue;
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }

    long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid
            ++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap -
                edges[id].flow < 1)
                continue;
            long long tr = dfs(u, min(pushed, edges[id].cap -
                edges[id].flow));
            if (tr == 0)
                continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
```

4.2 MinCost Flow

```
struct Edge
{
    int from, to, capacity, cost;
};

vector<vector<int>> adj, cost, capacity;

const int INF = 1e9;

void shortest_paths(int n, int v0, vector<int>& d, vector<int
    >& p) {
    d.assign(n, INF);
    d[v0] = 0;
    vector<bool> inq(n, false);
    queue<int> q;
    q.push(v0);
    p.assign(n, -1);

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inq[u] = false;
        for (int v : adj[u]) {
            if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v
                ]) {
                d[v] = d[u] + cost[u][v];
                p[v] = u;
                if (!inq[v]) {
                    inq[v] = true;
                    q.push(v);
                }
            }
        }
    }
}

int min_cost_flow(int N, vector<Edge> edges, int K, int s, int
     t) {
    adj.assign(N, vector<int>());
    cost.assign(N, vector<int>(N, 0));
    capacity.assign(N, vector<int>(N, 0));
    for (Edge e : edges) {
        adj[e.from].push_back(e.to);
        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;
        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity;
    }

    int flow = 0;
    int cost = 0;
    vector<int> d, p;
    while (flow < K) {
        shortest_paths(N, s, d, p);
        if (d[t] == INF)
            break;

        // find max flow on that path
```

```
        int f = K - flow;
        int cur = t;
        while (cur != s) {
            f = min(f, capacity[p[cur]][cur]);
            cur = p[cur];
        }

        // apply flow
        flow += f;
        cost += f * d[t];
        cur = t;
        while (cur != s) {
            capacity[p[cur]][cur] -= f;
            capacity[cur][p[cur]] += f;
            cur = p[cur];
        }
    }

    if (flow < K)
        return -1;
    else
        return cost;
}
```

5 Math

5.1 Primes

```
/*
2        3        5        7        11       13       17       19
          23       29       31       37       41       43       47
          53       59       61       67       71       73       79
          83       89
97       101      103      107      109      113      127      131
          137      139      149      151      157      163      167
          173      179      181      191      193      197      199
          211      223
227      229      233      239      241      251      257      263
          269      271      277      281      283      293      307
          311      313      317      331      337      347      349
          353      359
367      373      379      383      389      397      401      409
          419      421      431      433      439      443      449
          457      461      463      467      479      487      491
          499      503
509      521      523      541      547      557      563      569
          571      577      587      593      599      601      607
          613      617      619      631      641      643      647
          653      659
661      673      677      683      691      701      709      719
          727      733      739      743      751      757      761
          769      773      787      797      809      811      821
          823      827
829      839      853      857      859      863      877      881
          883      887      907      911      919      929      937
          941      947      953      967      971      977      983
          991      997

1009     1013     1019     1021     1031     1033     1039     1049
         1051     1061     1063     1069     1087     1091     1093
         1097     1103     1109     1117     1123
1129     1151     1153     1163     1171     1181     1187     1193
         1201     1213     1217     1223     1229     1231     1237
         1249     1259     1277     1279     1283
1289     1291     1297     1301     1303     1307     1319     1321
         1327     1361     1367     1373     1381     1399     1409
         1423     1427     1429     1433     1439
1447     1451     1453     1459     1471     1481     1483     1487
         1489     1493     1499     1511     1523     1531     1543
         1549     1553     1559     1567     1571
1579     1583     1597     1601     1607     1609     1613     1619
         1621     1627     1637     1657     1663     1667     1669
         1693     1697     1699     1709     1721
1723     1733     1741     1747     1753     1759     1777     1783
         1787     1789     1801     1811     1823     1831     1847
         1861     1867     1871     1873     1877
1879     1889     1901     1907     1913     1931     1933     1949
         1951     1973     1979     1987     1993     1997     1999

*/
```

5.2 Log Utils

```
ln: log()
log base 10: log10()
e: exp()
primos aproximados hasta x: x/ln(x) o x/(ln(x)-1.08366)
```

5.3 Modular Operations

```
const int MOD = 998244353;

int add ( int A, int B ) { return A+B<MOD? A+B: A+B-MOD; }
int mul ( int A, int B ) { return ll(A)*B % ll(MOD); }
int sub ( int A, int B ) { return add ( A, MOD-B ); }
```

5.4 Line Representation

```
//si b=0: es como si fuera oo o -oo
//si a=0: la fraccion es 0

struct frac{
    ll a,b;
    frac(ll_a, ll_b): a(_a), b(_b){
        if(b<0) a*=-1, b *=-1;
        if(b==0) a = 1;
    }

    bool operator < (frac other){
        return a * other.b < b * other.a;
    }
};

map<frac,frac> mp;
```

5.5 Lines Intersection

```
typedef complex<double> point;

/*Line Segment Intersection*/

double dot(const point &a, const point &b) { return real(conj(
    a) * b); }
```

```
double cross(const point &a, const point &b) { return imag(
    conj(a) * b); }

// returns intersection of infinite lines ab and pq (undefined
     if they are parallel)
point intersect(const point &a, const point &b, const point &p
    , const point &q)
{
    double d1 = cross(p - a, b - a);
    double d2 = cross(q - a, b - a);
    return (d1 * q - d2 * p) / (d1 - d2);
}

int main(){
    vector<int> p(8);
    for(int i=0; i < 8; i++){
        cin >> p[i];
    }

    point a(p[0],p[1]), b(p[2],p[3]), c(p[4],p[5]), d(p[6],p
        [7]);

    point ans = intersect(a,b,c,d);
    cout << fixed << setprecision(2) << real(ans) << " " <<
        imag(ans) << endl;
}
```

5.6 Cribe

```
int cribe[N];

for(int i=2; i < N; i++){
    if(!cribe[i]){
        for(int k=i+i; k<N; k+=i){
            cribe[k] = i;
        }
    }
}
```

5.7 FastCribe

```
#define forr(i,a,b) for(int i=(a); i<(b); i++)
typedef long long ll;
typedef pair<int,int> ii;

#define MAXP 100000     //no necesariamente primo
int criba[MAXP+1];
void crearcriba(){
        int w[] = {4,2,4,2,4,6,2,6};
        for(int p=25;p<=MAXP;p+=10) criba[p]=5;
        for(int p=9;p<=MAXP;p+=6) criba[p]=3;
        for(int p=4;p<=MAXP;p+=2) criba[p]=2;
        for(int p=7,cur=0;p*p<=MAXP;p+=w[cur++&7]) if (!criba[
            p])
                for(int j=p*p;j<=MAXP;j+=(p<<1)) if(!criba[j])
                        criba[j]=p;
}
vector<int> primos;
void buscarprimos(){
        crearcriba();
        forr (i,2,MAXP+1) if (!criba[i]) primos.push_back(i);
}
//~ Useful for bit trick: #define SET(i) ( criba[(i)
    >>5]|=1<<((i)&31) ), #define INDEX(i) ( (criba[i>>5]>>((i)
    &31))&1 ), unsigned int criba[MAXP/32+1];

int main() {
        freopen("primos", "w", stdout);
        buscarprimos();
        cout << '{';
        bool first=true;
        forall(it, primos){
                if(first) first=false;
                else cout << ',';
                cout << *it;
        }
        cout << "};\n";
        return 0;
}
```

5.8 Fast Exp.

```
ll binpow(ll a, ll b) {
    /*si se necesita la potencia modulo m: aplicar el modulo a
         todas
    las multiplicaciones y a 'a' al antes del loop*/
    ll res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

5.9 Matrix Power

```
ll MOD;
const int MAX_N = 2;

struct Matrix {ll mat[MAX_N][MAX_N];};

ll mod(ll a, ll m) {return ((a%m)+m)%m;}

Matrix matMul(Matrix a, Matrix b){
    Matrix ans;
    rep(i,MAX_N){
        rep(j, MAX_N){
            ans.mat[i][j] = 0;
        }
    }
    rep(i,MAX_N){
        rep(k,MAX_N){
            if(a.mat[i][k] == 0) continue;
            rep(j, MAX_N){
                ans.mat[i][j] += mod(a.mat[i][k], MOD) * mod(b
                    .mat[k][j], MOD);
                ans.mat[i][j] = mod(ans.mat[i][j], MOD);
            }
        }
    }
    return ans;
}
```

```
Matrix matPow(Matrix base, ll p){
    Matrix ans;
    rep(i,MAX_N)
        rep(j,MAX_N)
            ans.mat[i][j] = (i==j);

    while(p){
        if(p&1) ans = matMul(ans,base);
        base = matMul(base,base);
        p >>= 1;
    }
    return ans;
}

int main(){
    /*Fib(n) mod 2 ^ m*/
    ll n, m;
    MOD = binpow(2,m);
    Matrix mat;
    mat.mat[0][0] = 1;
    mat.mat[0][1] = 1;
    mat.mat[1][0] = 1;
    mat.mat[1][1] = 0;

    Matrix aa = matPow(mat, n);

    cout << aa.mat[0][1] << endl;

}
```

5.10 Fast Matrix Exp.

```
#define forn(i,n) forr(i,0,n)
#define SIZE 350
int NN;
double tmp[SIZE][SIZE];
void mul(double a[SIZE][SIZE], double b[SIZE][SIZE]){ zero(tmp
    );
    forn(i, NN) forn(j, NN) forn(k, NN) res[i][j]+=a[i][k]*b[k
        ][j];
    forn(i, NN) forn(j, NN) a[i][j]=res[i][j];
}
void powmat(double a[SIZE][SIZE], int n, double res[SIZE][SIZE
    ]){
    forn(i, NN) forn(j, NN) res[i][j]=(i==j);
    while(n){
        if(n&1) mul(res, a), n--;
        else mul(a, a), n/=2;
    } }
```

5.11 Euclidean algorithm

```
//Iterative
int gcd(int a, int b, int& x, int& y) {
    x = 1, y = 0;
    int x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        int q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a1, b1) = make_tuple(b1, a1 - q * b1);
    }
    return a1;
}
```

5.12 GaussJordan

```
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be infinity
    or a big number

int gauss (vector < vector<double> > a, vector<double> & ans)
{
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }

    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)
            return 0;
    }

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```

5.13 Chinese Remainder Theorem

```
struct Congruence {
    long long a, m;
};

long long chinese_remainder_theorem(vector<Congruence> const&
    congruences) {
```

```
    long long M = 1;
    for (auto const& congruence : congruences) {
        M *= congruence.m;
    }

    long long solution = 0;
    for (auto const& congruence : congruences) {
        long long a_i = congruence.a;
        long long M_i = M / congruence.m;
        long long N_i = mod_inv(M_i, congruence.m);
        solution = (solution + a_i * M_i % M * N_i) % M;
    }
    return solution;
}
```

6 Geometry

6.1 Poligon

```
#include <bits/stdc++.h>
using namespace std;

const double EPS = 1e-9;

double DEG_to_RAD(double d) { return d*M_PI / 180.0; }

double RAD_to_DEG(double r) { return r*180.0 / M_PI; }

struct point { double x, y;   // only used if more precision
    is needed
  point() { x = y = 0.0; }                        // default
     constructor
  point(double _x, double _y) : x(_x), y(_y) {}         // user
     -defined
  bool operator == (point other) const {
   return (fabs(x-other.x) < EPS && (fabs(y-other.y) < EPS));
      }
  bool operator <(const point &p) const {
   return x < p.x || (abs(x-p.x) < EPS && y < p.y); } };

struct vec { double x, y;  // name: 'vec' is different from
    STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to
    vector a->b
  return vec(b.x-a.x, b.y-a.y); }

double dist(point p1, point p2) {                // Euclidean
    distance
  return hypot(p1.x-p2.x, p1.y-p2.y); }                 //
     return double

// returns the perimeter of polygon P, which is the sum of
// Euclidian distances of consecutive line segments (polygon
    edges)
double perimeter(const vector<point> &P) {       // by ref for
     efficiency
  double ans = 0.0;
  for (int i = 0; i < (int)P.size()-1; ++i)      // note: P[n
     -1] = P[0]
    ans += dist(P[i], P[i+1]);                   // as we
       duplicate P[0]
  return ans;
}

// returns the area of polygon P
double area(const vector<point> &P) {
  double ans = 0.0;
  for (int i = 0; i < (int)P.size()-1; ++i)      // Shoelace
     formula
    ans += (P[i].x*P[i+1].y - P[i+1].x*P[i].y);
  return fabs(ans)/2.0;                          // only do /
     2.0 here
}

double dot(vec a, vec b) { return (a.x*b.x + a.y*b.y); }

double norm_sq(vec v) { return v.x*v.x + v.y*v.y; }

double angle(point a, point o, point b) {  // returns angle
    aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
      }

double cross(vec a, vec b) { return a.x*b.y - a.y*b.x; }

// returns the area of polygon P, which is half the cross
    products
// of vectors defined by edge endpoints
double area_alternative(const vector<point> &P) {
  double ans = 0.0; point O(0.0, 0.0);           // O = the
     Origin
  for (int i = 0; i < (int)P.size()-1; ++i)      // sum of
     signed areas
    ans += cross(toVec(O, P[i]), toVec(O, P[i+1]));
  return fabs(ans)/2.0;
}

// note: to accept collinear points, we have to change the '>
    0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0;
}

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
}

// returns true if we always make the same turn
// while examining all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
  int n = (int)P.size();
  // a point/sz=2 or a line/sz=3 is not convex
  if (n <= 3) return false;
  bool firstTurn = ccw(P[0], P[1], P[2]);        // remember
     one result,
  for (int i = 1; i < n-1; ++i)                 // compare
     with the others
    if (ccw(P[i], P[i+1], P[(i+2) == n ? 1 : i+2]) !=
       firstTurn)
      return false;                              // different
        -> concave
  return true;                                   // otherwise
     -> convex
}
```

```cpp
// returns 1/0/-1 if point p is inside/on (vertex/edge)/outside of
// either convex/concave polygon P
int insidePolygon(point pt, const vector<point> &P) {
  int n = (int)P.size();
  if (n <= 3) return -1;                         // avoid point or line
  bool on_polygon = false;
  for (int i = 0; i < n-1; ++i)                  // on vertex/edge?
    if (fabs(dist(P[i], pt) + dist(pt, P[i+1]) - dist(P[i], P[i+1])) < EPS)
      on_polygon = true;
  if (on_polygon) return 0;                      // pt is on polygon
  double sum = 0.0;                              // first = last point
  for (int i = 0; i < n-1; ++i) {
    if (ccw(pt, P[i], P[i+1]))
      sum += angle(P[i], pt, P[i+1]);            // left turn/ccw
    else
      sum -= angle(P[i], pt, P[i+1]);            // right turn/cw
  }
  return fabs(sum) > M_PI ? 1 : -1;              // 360d->in, 0d->out
}

// compute the intersection point between line segment p-q and line A-B
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y-A.y, b = A.x-B.x, c = B.x*A.y - A.x*B.y;
  double u = fabs(a*p.x + b*p.y + c);
  double v = fabs(a*q.x + b*q.y + c);
  return point((p.x*v + q.x*u) / (u+v), (p.y*v + q.y*u) / (u+v));
}

// cuts polygon Q along the line formed by point A->point B (order matters)
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point A, point B, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); ++i) {
    double left1 = cross(toVec(A, B), toVec(A, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(A, B), toVec(A, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);         // Q[i] is on the left
    if (left1*left2 < -EPS)                      // crosses line AB
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], A, B));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());                      // wrap around
  return P;
}

vector<point> CH_Graham(vector<point> &Pts) {    // overall O(n log n)
  vector<point> P(Pts);                          // copy all points
  int n = (int)P.size();
  if (n <= 3) {                                  // point/line/triangle
    if (!(P[0] == P[n-1])) P.push_back(P[0]);    // corner case
    return P;                                    // the CH is P itself
  }

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = min_element(P.begin(), P.end())-P.begin();
  swap(P[0], P[P0]);                             // swap P[P0] with P[0]

  // second, sort points by angle around P0, O(n log n) for this sort
  sort(++P.begin(), P.end(), [&](point a, point b) {
    return ccw(P[0], a, b);                      // use P[0] as the pivot
  });

  // third, the ccw tests, although complex, it is just O(n)
  vector<point> S({P[n-1], P[0], P[1]});         // initial S
  int i = 2;                                     // then, we check the rest
  while (i < n) {                                // n > 3, O(n)
    int j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i]))                 // CCW turn
      S.push_back(P[i++]);                       // accept this point
    else                                         // CW turn
      S.pop_back();                              // pop until a CCW turn
  }
  return S;                                      // return the result
}

vector<point> CH_Andrew(vector<point> &Pts) {    // overall O(n log n)
  int n = Pts.size(), k = 0;
  vector<point> H(2*n);
  sort(Pts.begin(), Pts.end());                  // sort the points by x/y
  for (int i = 0; i < n; ++i) {                  // build lower hull
    while ((k >= 2) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
    H[k++] = Pts[i];
  }
  for (int i = n-2, t = k+1; i >= 0; --i) {       // build upper hull
    while ((k >= t) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
    H[k++] = Pts[i];
  }
  H.resize(k);
  return H;
}

int main() {
  // 6(+1) points, entered in counter clockwise order, 0-based indexing
  vector<point> P;
  P.emplace_back(1, 1);                          // P0
```

```
P.emplace_back(3, 3);                          // P1
P.emplace_back(9, 1);                          // P2
P.emplace_back(12, 4);                         // P3
P.emplace_back(9, 7);                          // P4
P.emplace_back(1, 7);                          // P5
P.push_back(P[0]);                             // loop back,
    P6 = P0

printf("Perimeter = %.2lf\n", perimeter(P));   // 31.64
printf("Area = %.2lf\n", area(P));             // 49.00
printf("Area = %.2lf\n", area_alternative(P)); // also 49.00
printf("Is convex = %d\n", isConvex(P));       // 0 (false)

//// the positions of P_out, P_on, P_in with respect to the
    polygon
//7 P5------P_on----P4
//6 |                  \
//5 |                    \
//4 |   P_in              P3
//3 |   P1___            /
//2 | / P_out \ ___     /
//1 P0                P2
//0 1 2 3 4 5 6 7 8 9 101112

point p_out(3, 2); // outside this (concave) polygon
printf("P_out is inside = %d\n", insidePolygon(p_out, P));
    // -1
printf("P1 is inside = %d\n", insidePolygon(P[1], P)); // 0
point p_on(5, 7); // on this (concave) polygon
printf("P_on is inside = %d\n", insidePolygon(p_on, P)); //
    0
point p_in(3, 4); // inside this (concave) polygon
printf("P_in is inside = %d\n", insidePolygon(p_in, P)); //
    1

// cutting the original polygon based on line P[2] -> P[4] (
    get the left side)
//7 P5---------------P4
//6 |               |  \
//5 |               |    \
//4 |               |     P3
//3 |   P1___       |    /
//2 | /       \ ___ |  /
//1 P0                P2
//0 1 2 3 4 5 6 7 8 9 101112
// new polygon (notice the index are different now):
//7 P4---------------P3
//6 |               |
//5 |               |
//4 |               |
//3 |   P1___       |
//2 | /       \ ___ |
//1 P0                P2
//0 1 2 3 4 5 6 7 8 9

P = cutPolygon(P[2], P[4], P);
printf("Perimeter = %.2lf\n", perimeter(P));   // smaller
    now, 29.15
printf("Area = %.2lf\n", area(P));             // 40.00

// running convex hull of the resulting polygon (index
    changes again)
//7 P3---------------P2
//6 |               |
//5 |               |
//4 |   P_out       |
//3 |               |
//2 |   P_in        |
//1 P0---------------P1
//0 1 2 3 4 5 6 7 8 9

P = CH_Graham(P);                              // now this
    is a rectangle
printf("Perimeter = %.2lf\n", perimeter(P));   // precisely
    28.00
printf("Area = %.2lf\n", area(P));             // precisely
    48.00
printf("Is convex = %d\n", isConvex(P));       // true
printf("P_out is inside = %d\n", insidePolygon(p_out, P));
    // 1
printf("P_in is inside = %d\n", insidePolygon(p_in, P)); //
    1

return 0;
}
```

7 Range Queries

7.1 BIT

```
const int N = 200005;
int BIT[N];

void update(int idx, int val){
    for(; idx < N; idx += idx&(-idx)){
        BIT[idx]+=val;
    }
}

int query (int idx){
    ll ret = 0;
    for(; idx > 0; idx-=idx&(-idx)){
        ret += BIT[idx];
    }
    return ret;
}

int query (int left, int right){
    return query(right) - query(left-1);
}

int lower_find(int val){
    int id = 0;
    for(int i = 31-__builtin_clz(n); i >= 0; --i){
        int nid = id | ( 1 << i);
        if(nid <= n && BIT[nid] <= val){
            val -= BIT[nid];
            id = nid;
        }
    }
    return id;
}


iota(idx+1, idx+n+1,1);
```

```
sort(idx+1, idx+n+1, [](int i_a, int i_b) { return arr[i_a] >
    arr[i_b];});

//Update range [l,r] to v
update(l,v);
update(r+1,-v);

//Update specific value at pos k to u
ll prev = query(k)-query(k-1);
update(k,u);
update(k, -prev);

//Inversions
for(int i=1; i <=n; i++){
    forward[i] = query(values[i]);
    update(1,1);
    update(values[i],-1);
}
memset(BIT, 0, sizeof BIT);

for(int i=n; i >0 ; i--){
    backward[i] = query(values[i]);
    update(values[i]+1, 1);
}


//Dimension change
sort(difval, difval+ind);
map<int,int> idx;
int cnt = 0;
idx[difval[0]] = cnt;
cnt++;
for(int i=1; i < ind;  i++){
    if(difval[i] != difval[i-1]){
        idx[difval[i]] = cnt;
        cnt++;
    }
}
```

7.2 Basic Segment Tree

```
const int N = 1e6+5;

struct SegTree {
    SegTree *L, *R;
    int fr, to;

    SegTree (int fr, int to):
        fr(fr), to(to){
            if(fr == to){
                //Calc base values
                L = R = NULL;
            }else if (fr < to){
                L = new SegTree(fr, (fr+to)/2);
                R = new SegTree((L->to)+1, to);
                //Calc value: L->value + R->value
            }
        }

    void propagate (){
        //Propagation operation
    }

    void update(int l, int r){
        propagate();
        if(l == fr && r == to){
            //Change before prop? ex: change ^= true;
            propagate();
            return;
        }
        if(r < R->fr){
            L->update(l,r);
            R->propagate();
        }else if(l > L->to){
            R->update(l,r);
            L->propagate();
        }else{
            L->update(l, L->to);
            R->update(R->fr, r);
        }
        //Calc values, ex: num_on = L->num_on + R->num_on;
    }
};
```

7.3 Segment Tree Range Query

```
const int N = 1e5;  // limit for array size
int n;  // array size
int t[2 * N];

void build() {  // build the tree
  for (int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}//O(n)

void modify(int p, int value) {  // set value at position p
  for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p
      ^1];
}//O(log(n))

int query(int l, int r) {  // sum on interval [l, r)
  int res = 0;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) res += t[l++];
    if (r&1) res += t[--r];
  }
  return res;
}//O(log(n))

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) scanf("%d", t + n + i);
  build();
  modify(0, 1);
  printf("%d\n", query(3, 11));
  return 0;
}
```

7.4 Segment Tree Range Update

```
void modify(int l, int r, int value) {
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) t[l++] += value;
    if (r&1) t[--r] += value;
  }
}

int query(int p) {
```

```
    int res = 0;
    for (p += n; p > 0; p >>= 1) res += t[p];
    return res;
  }

  /*Push to inspect modifications*/

  void push() {
    for (int i = 1; i < n; ++i) {
      t[i<<1] += t[i];
      t[i<<1|1] += t[i];
      t[i] = 0;
    }
  } //
```

7.5 Segment Tree Lazy Propagation

```
const int N = 500005;
const int MOD = 998244353;

int add ( int A, int B ) { return A+B<MOD? A+B: A+B-MOD; }
int mul ( int A, int B ) { return ll(A)*B % ll(MOD); }
int sub ( int A, int B ) { return add ( A, MOD-B ); }

int n, q, h;

int sum[2*N];
pii lazy[2*N];
int lengths[2*N];

pii combine ( pii A, pii B ) {
    return {mul(A.ff, B.ff), add(mul(A.ss, B.ff), B.ss)};
}

void apply(int p, pii value) {
    sum[p] = add(mul(sum[p], value.ff), mul(lengths[p], value.
        ss));
    if (p < n) lazy[p] = combine(lazy[p], value);
}

void build_t() {  // build the tree
  for (int i = n - 1; i > 0; --i) {
    sum[i] = add(sum[i<<1], sum[i<<1|1]);
    lengths[i] = lengths[i<<1]+lengths[i<<1|1];
    lazy[i] = {1,0};
  }
}

void build(int p) {
  while (p > 1){
    p >>= 1;
    if(lazy[p] == pii(1,0)) sum[p] = add(sum[p<<1], sum[p
        <<1|1]);

  }
}

void push(int p) {
  for (int s = h-1; s > 0; --s) {
    int i = p >> s;
    if (lazy[i] != pii(1,0)) {
      apply(i<<1, lazy[i]);
      apply(i<<1|1, lazy[i]);
      lazy[i] = {1,0};
    }
  }
}

void modify(int l, int r, pii value) {
  l += n, r += n;
  int l0 = l, r0 = r;
  push(l0);
  push(r0 - 1);
  for (; l < r; l >>= 1, r >>= 1) {
    if (l&1) apply(l++, value);
    if (r&1) apply(--r, value);
  }
  build(l0);
  build(r0 - 1);
}

int query(int l, int r) {
  l += n, r += n;
  push(l);
  push(r - 1);
  int res = 0;
  for (; l < r; l >>= 1, r >>= 1) {
    if (l&1)res = add(res, sum[l++]);
    if (r&1)res = add(sum[--r], res);
  }
  return res;
}

//Initialization:
scanf ( "%d%d", &n, &q );
h = sizeof(int) * 8 - __builtin_clz(n);
//cout << "h: " << h << endl;

for ( int i = n; i < 2*n; ++i ){
    scanf ( "%d", &sum[i] );
    lengths[i] = 1;
}

build_t();
```

7.6 Sparse Table

```
#include <bits/stdc++.h>
using namespace std;

typedef vector<int> vi;

class SparseTable {                              // OOP style
private:
  vi A, P2, L2;
  vector<vi> SpT;                                // the Sparse
      Table
public:
  SparseTable() {}                               // default
      constructor

  SparseTable(vi &initialA) {                    // pre-
      processing routine
    A = initialA;
    int n = (int)A.size();
    int L2_n = (int)log2(n)+1;
    P2.assign(L2_n+1, 0);
    L2.assign((1<<L2_n)+1, 0);
    for (int i = 0; i <= L2_n; ++i) {
```

```
      P2[i] = (1<<i);                       // to speed
        up 2^i
      L2[(1<<i)] = i;                       // to speed
        up log_2(i)
    }
    for (int i = 2; i < P2[L2_n]; ++i)
      if (L2[i] == 0)
        L2[i] = L2[i-1];                    // to fill in
          the blanks

    // the initialization phase
    SpT = vector<vi>(L2[n]+1, vi(n));
    for (int j = 0; j < n; ++j)
      SpT[0][j] = j;                        // RMQ of sub
         array [j..j]

    // the two nested loops below have overall time complexity
         = O(n log n)
    for (int i = 1; P2[i] <= n; ++i)        // for all i
       s.t. 2^i <= n
      for (int j = 0; j+P2[i]-1 < n; ++j) { // for all
         valid j
        int x = SpT[i-1][j];                // [j..j+2^(i
          -1)-1]
        int y = SpT[i-1][j+P2[i-1]];        // [j+2^(i-1)
          ..j+2^i-1]
        SpT[i][j] = A[x] <= A[y] ? x : y;
      }
  }

  int RMQ(int i, int j) {
    int k = L2[j-i+1];                      // 2^k <= (j-
       i+1)
    int x = SpT[k][i];                      // covers [i
       ..i+2^k-1]
    int y = SpT[k][j-P2[k]+1];              // covers [j
       -2^k+1..j]
    return A[x] <= A[y] ? x : y;
  }
};

int main() {
  // same example as in Chapter 2: Segment Tree
  vi A = {18, 17, 13, 19, 15, 11, 20};
  SparseTable SpT(A);
  int n = (int)A.size();
  for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j)
      printf("RMQ(%d, %d) = %d\n", i, j, SpT.RMQ(i, j));
  return 0;
}
```

8 Strings

8.1 Borders

```
const int N = 1e6+5;
int b[N];
int sz;

void borders(string p){
    b[0] = -1;
    p = '#'+p;
    for(int i=1; i <= sz; i++){
        int j=b[i-1];
        while(j>=0 && p[i] != p[j+1]) j = b[j];
        b[i] = j+1;
    }
}

//Encontrar periodos:
sz = s.size();
int aux = sz;
borders(s);

while(aux){
    cout << sz-b[aux] << " ";
    aux = b[aux];
}
```

8.2 Hashing

```
const int p = 283;
const int M = 1e9+7;
const int N = 1e6+1;

int P[N], h[N];

ll binpow(ll a, ll b) {
    ll res = 1;
    a %= M;
    while (b > 0) {
        if (b & 1)
            res = (res * a)%M;
        a = (a * a)%M;
        b >>= 1;
    }
    return res;
}

void prepareP(int n){
    P[0] = 1;
    for(int i =1; i < n; ++i){
        P[i] = ((ll)P[i-1]*p) % M;
    }
}

void computeRollingHash(string T){
    for(int i=0; i < (int)T.size(); ++i){
        if(i!=0) h[i] = h[i-1];
        h[i] = (h[i]+((ll)(T[i]-'a'+1)*P[i])%M)%M;
    }
}

int hash_fast(int L, int R){
    if(L==0) return h[R];
    int ans = 0;
    ans = ((h[R]-h[L-1]) %M +M) %M;
    ans = ((ll)ans*binpow(P[L],M-2)) %M;
    return ans;
}
```

8.3 Manacher

```
//Find palindromes
vector<int> manacher_odd(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    int l = 1, r = 1;
    for(int i = 1; i <= n; i++) {
        p[i] = max(0, min(r - i, p[l + (r - i)]));
        while(s[i - p[i]] == s[i + p[i]]) {
            p[i]++;
        }
        if(i + p[i] > r) {
            l = i - p[i], r = i + p[i];
        }
    }
    return vector<int>(begin(p) + 1, end(p) - 1);
}
```

8.4 Z-Algorithm

```
vector<int> z(string s) {
    int n = s.size();
    vector<int> z(n);
    int x = 0, y = 0;
    for (int i = 1; i < n; i++) {
        z[i] = max(0,min(z[i-x],y-i+1));
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
            x = i; y = i+z[i]; z[i]++;
        }
    }
    return z;
}
```

8.5 Prefix Function (KMP)

```
/*KMP
long max del prefijo que tambien es sufijo del substring s
    [0...i]*/
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

8.6 Trie

```
const int N = 500'005; //Number of nodes
const int ALPHA = 26; // Number of characters
int trie[N][ALPHA];
set<int> final_s;
int last_n = 0;

bool is_already(string &s){
    int curr_node = 0;
    for(char it: s){
        if(trie[curr_node][it-'a'] == 0){
            trie[curr_node][it-'a'] = ++last_n;
        }
        curr_node = trie[curr_node][it-'a'];
    }
    bool is_alr = final_s.count(curr_node);
    final_s.insert(curr_node);
    return is_alr;
}
```

8.7 Aho Corasick

```
/*sufijo mas grande en el trie?*/
const int K = 26;
int last_n = 0;

struct Node {
    char c;
    int next[K], go[K]; //next: trie, go: automata
    bool terminal = false;
    int patt = -1;
    int p = -1, link = -1;
    set<int> has_terminals;

    Node(int p=-1, char c = '$') : p(p), c(c) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Node> trie(1);

void insert(string &s, int idx){
    int curr_node = 0;
    for(char ch: s){
        int c = ch-'a';
        if(trie[curr_node].next[c] == -1){
            trie[curr_node].next[c] = ++last_n;
            trie.emplace_back(curr_node, ch);
        }
        curr_node = trie[curr_node].next[c];
    }
    trie[curr_node].terminal = true;
    trie[curr_node].patt = idx;;
}

int go(int v, char ch);

int get_link(int v){
    if (trie[v].link == -1){
        if(v == 0 || trie[v].p == 0){
            trie[v].link = 0;
        }else{
            trie[v].link = go(get_link(trie[v].p), trie[v].c);
        }
    }
    return trie[v].link;
```

```cpp
}

int go(int v, char ch){
    int c = ch - 'a';
    if(trie[v].go[c] == -1){
        if(trie[v].next[c] != -1){
            trie[v].go[c] = trie[v].next[c];
        }else{
            trie[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
        }
    }
    return trie[v].go[c];
}
```

8.8 Strings Matching

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAX_N = 200010;

char T[MAX_N], P[MAX_N];                           // T = text, P = pattern
int n, m;                                          // n = |T|, m = |P|

// Knuth-Morris-Pratt's algorithm specific code
int b[MAX_N];                                      // b = back table

int naiveMatching() {
  int freq = 0;
  for (int i = 0; i < n; ++i) {                    // try all starting index
    bool found = true;
    for (int j = 0; (j < m) && found; ++j)
      if ((i+j >= n) || (P[j] != T[i+j]))          // if mismatch found
        found = false;                             // abort this, try i+1
    if (found) {                                   // T[i..i+m-1] = P[0..m-1]
      ++freq;
      // printf("P is found at index %d in T\n", i);
    }
  }
  return freq;
}

void kmpPreprocess() {                             // call this first
  int i = 0, j = -1; b[0] = -1;                    // starting values
  while (i < m) {                                  // pre-process P
    while ((j >= 0) && (P[i] != P[j])) j = b[j]; // different, reset j
    ++i; ++j;                                      // same, advance both
    b[i] = j;
  }
}

int kmpSearch() {                                  // similar as above
  int freq = 0;
  int i = 0, j = 0;                                // starting values
  while (i < n) {                                  // search through T
    while ((j >= 0) && (T[i] != P[j])) j = b[j]; // if different, reset j
    ++i; ++j;                                      // if same, advance both
    if (j == m) {                                  // a match is found
      ++freq;
      // printf("P is found at index %d in T\n", i-j);
      j = b[j];                                    // prepare j for the next
    }
  }
  return freq;
}

// Rabin-Karp's algorithm specific code
typedef long long ll;

const int p = 131;                                 // p and M are
const int M = 1e9+7;                               // relatively prime

int Pow[MAX_N];                                    // to store p^i % M
int h[MAX_N];                                      // to store prefix hashes

void computeRollingHash() {                        // Overall: O(n)
  Pow[0] = 1;                                      // compute p^i % M
  for (int i = 1; i < n; ++i)                      // O(n)
    Pow[i] = ((ll)Pow[i-1]*p) % M;
  h[0] = 0;
  for (int i = 0; i < n; ++i) {                    // O(n)
    if (i != 0) h[i] = h[i-1];                     // rolling hash
    h[i] = (h[i] + ((ll)T[i]*Pow[i]) % M) % M;
  }
}

int extEuclid(int a, int b, int &x, int &y) {      // pass x and y by ref
  int xx = y = 0;
  int yy = x = 1;
  while (b) {                                      // repeats until b == 0
    int q = a/b;
    tie(a, b) = tuple(b, a%b);
    tie(x, xx) = tuple(xx, x-q*xx);
    tie(y, yy) = tuple(yy, y-q*yy);
  }
  return a;                                        // returns gcd(a, b)
}

int modInverse(int b, int m) {                     // returns b^(-1) (mod m)
```

```
  int x, y;
  int d = extEuclid(b, m, x, y);                    // to get b*x
     + m*y == d
  if (d != 1) return -1;                            // to
     indicate failure
  // b*x + m*y == 1, now apply (mod m) to get b*x == 1 (mod m)
  return (x+m)%m;                                   // this is
     the answer
}

int hash_fast(int L, int R) {                       // O(1) hash
   of any substr
  if (L == 0) return h[R];                          // h is the
     prefix hashes
  int ans = 0;
  ans = ((h[R] - h[L-1]) % M + M) % M;              // compute
     differences
  ans = ((ll)ans * modInverse(Pow[L], M)) % M;      // remove P[L
     ]^-1 (mod M)
  return ans;
}

int main() {
  // strcpy(T, "I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY
     SEVEN");
  // strcpy(P, "SEVENTY SEVEN");
  int extreme_limit = 100000; // experiment time is about 10s+
      total
  for (int i = 0; i < extreme_limit-1; ++i) T[i] = 'A'+rand()
     %2;
  T[extreme_limit-2] = 'B';
  T[extreme_limit-1] = 0;
  for (int i = 0; i < 100; ++i) P[i] = 'A'+rand()%2;
  P[10] = 0;
  n = (int)strlen(T);
  m = (int)strlen(P);

  //if the end of line character is read too, uncomment the
     line below
  //T[n-1] = 0; n--; P[m-1] = 0; m--;

  // printf("T = '%s'\n", T);
  // printf("P = '%s'\n", P);
  // printf("\n");

  clock_t t0 = clock();
  printf("String Library, #match = ");
  char *pos = strstr(T, P);
  int freq = 0;
  while (pos != NULL) {
    ++freq;
    // printf("P is found at index %d in T\n", pos-T);
    pos = strstr(pos+1, P);
  }
  printf("%d\n", freq);
  clock_t t1 = clock();
  printf("Runtime = %.10lf s\n\n", (t1-t0) / (double)
     CLOCKS_PER_SEC);

  printf("Naive Matching, #match = ");
  printf("%d\n", naiveMatching());
  clock_t t2 = clock();
  printf("Runtime = %.10lf s\n\n", (t2-t1) / (double)
     CLOCKS_PER_SEC);

  printf("Rabin-Karp, #match = ");
  computeRollingHash();                               // use
     Rolling Hash
  int hP = 0;
  for (int i = 0; i < m; ++i)                         // O(n)
    hP = (hP + (ll)P[i]*Pow[i]) % M;
  freq = 0;
  for (int i = 0; i <= n-m; ++i)                      // try all
     starting pos
    if (hash_fast(i, i+m-1) == hP) {                  // a possible
        match
      ++freq;
      // printf("P is found at index %d in T\n", i);
    }
  printf("%d\n", freq);
  clock_t t3 = clock();
  printf("Runtime = %.10lf s\n\n", (t3-t2) / (double)
     CLOCKS_PER_SEC);

  printf("Knuth-Morris-Pratt, #match = ");
  kmpPreprocess();
  printf("%d\n", kmpSearch());
  clock_t t4 = clock();
  printf("Runtime = %.10lf s\n\n", (t4-t3) / (double)
     CLOCKS_PER_SEC);

  return 0;
}
```

8.9 Suffix Array + LCP

```
#include <bits/stdc++.h>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;

class SuffixArray {
private:
  vi RA;                                              // rank array

  void countingSort(int k) {                          // O(n)
    int maxi = max(300, n);                           // up to 255
       ASCII chars
    vi c(maxi, 0);                                    // clear
       frequency table
    for (int i = 0; i < n; ++i)                       // count the
       frequency
      ++c[i+k < n ? RA[i+k] : 0];                     // of each
         integer rank
    for (int i = 0, sum = 0; i < maxi; ++i) {
      int t = c[i]; c[i] = sum; sum += t;
    }
    vi tempSA(n);
    for (int i = 0; i < n; ++i)                       // sort SA
      tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
    swap(SA, tempSA);                                 // update SA
  }

  void constructSA() {                                // can go up
     to 400K chars
    SA.resize(n);
    iota(SA.begin(), SA.end(), 0);                    // the
       initial SA
    RA.resize(n);
```

```cpp
    for (int i = 0; i < n; ++i) RA[i] = T[i];    // initial rankings
    for (int k = 1; k < n; k <<= 1) {            // repeat log_2 n times
      // this is actually radix sort
      countingSort(k);                           // sort by 2nd item
      countingSort(0);                           // stable-sort by 1st item
      vi tempRA(n);
      int r = 0;
      tempRA[SA[0]] = r;                         // re-ranking process
      for (int i = 1; i < n; ++i)                // compare adj suffixes
        tempRA[SA[i]] = // same pair => same rank r; otherwise, increase r
          ((RA[SA[i]] == RA[SA[i-1]]) && (RA[SA[i]+k] == RA[SA[i-1]+k])) ?
            r : ++r;
      swap(RA, tempRA);                          // update RA
      if (RA[SA[n-1]] == n-1) break;             // nice optimization
    }
  }

  void computeLCP() {
    vi Phi(n);
    vi PLCP(n);
    PLCP.resize(n);
    Phi[SA[0]] = -1;                             // default value
    for (int i = 1; i < n; ++i)                  // compute Phi in O(n)
      Phi[SA[i]] = SA[i-1];                      // remember prev suffix
    for (int i = 0, L = 0; i < n; ++i) {         // compute PLCP in O(n)
      if (Phi[i] == -1) { PLCP[i] = 0; continue; } // special case
      while ((i+L < n) && (Phi[i]+L < n) && (T[i+L] == T[Phi[i]+L]))
        ++L;                                     // L incr max n times
      PLCP[i] = L;
      L = max(L-1, 0);                           // L dec max n times
    }
    LCP.resize(n);
    for (int i = 0; i < n; ++i)                  // compute LCP in O(n)
      LCP[i] = PLCP[SA[i]];                      // restore PLCP
  }

public:
  const char* T;                                 // the input string
  const int n;                                   // the length of T
  vi SA;                                         // Suffix Array
  vi LCP;                                        // of adj sorted suffixes

  SuffixArray(const char* initialT, const int _n) : T(initialT), n(_n) {
    constructSA();                               // O(n log n)
    computeLCP();                                // O(n)
  }

  ii stringMatching(const char *P) {             // in O(m log n)
    int m = (int)strlen(P);                      // usually, m < n
    int lo = 0, hi = n-1;                        // range = [0..n-1]
    while (lo < hi) {                            // find lower bound
      int mid = (lo+hi) / 2;                     // this is round down
      int res = strncmp(T+SA[mid], P, m);        // P in suffix SA[mid]?
      (res >= 0) ? hi = mid : lo = mid+1;        // notice the >= sign
    }
    if (strncmp(T+SA[lo], P, m) != 0) return {-1, -1}; // if not found
    ii ans; ans.first = lo;
    hi = n-1;                                    // range = [lo..n-1]
    while (lo < hi) {                            // now find upper bound
      int mid = (lo+hi) / 2;
      int res = strncmp(T+SA[mid], P, m);
      (res > 0) ? hi = mid : lo = mid+1;         // notice the > sign
    }
    if (strncmp(T+SA[hi], P, m) != 0) --hi;      // special case
    ans.second = hi;
    return ans;                                  // returns (lb, ub)
  }                                              // where P is found

  ii LRS() {                                     // (LRS length, index)
    int idx = 0, maxLCP = -1;
    for (int i = 1; i < n; ++i)                  // O(n), start from i = 1
      if (LCP[i] > maxLCP)
        maxLCP = LCP[i], idx = i;
    return {maxLCP, idx};
  }

  ii LCS(int split_idx) {                        // (LCS length, index)
    int idx = 0, maxLCP = -1;
    for (int i = 1; i < n; ++i) {                // O(n), start from i = 1
      // if suffix SA[i] and suffix SA[i-1] came from the same string, skip
      if ((SA[i] < split_idx) == (SA[i-1] < split_idx))
        continue;
      if (LCP[i] > maxLCP)
        maxLCP = LCP[i], idx = i;
    }
    return {maxLCP, idx};
  }
```

```cpp
};

const int MAX_N = 450010;                    // can go up to 450K chars

char T[MAX_N];
char P[MAX_N];
char LRS_ans[MAX_N];
char LCS_ans[MAX_N];

int main() {
  freopen("sa_lcp_in.txt", "r", stdin);
  scanf("%s", &T);                           // read T
  int n = (int)strlen(T);                    // count n
  T[n++] = '$';                              // add terminating symbol
  SuffixArray S(T, n);                       // construct SA+LCP

  printf("T = '%s'\n", T);
  printf(" i SA[i] LCP[i]   Suffix SA[i]\n");
  for (int i = 0; i < n; ++i)
    printf("%2d    %2d    %2d    %s\n", i, S.SA[i], S.LCP[i],
        T+S.SA[i]);

  // String Matching demo, we will try to find P in T
  strcpy(P, "A");
  auto [lb, ub] = S.stringMatching(P);
  if ((lb != -1) && (ub != -1)) {
    printf("P = '%s' is found SA[%d..%d] of T = '%s'\n", P, lb
        , ub, T);
    printf("They are:\n");
    for (int i = lb; i <= ub; ++i)
      printf("  %s\n", T+S.SA[i]);
  }
  else
    printf("P = '%s' is not found in T = '%s'\n", P, T);

  // LRS demo, find the LRS of T
  auto [LRS_len, LRS_idx] = S.LRS();
  strncpy(LRS_ans, T+S.SA[LRS_idx], LRS_len);
  printf("The LRS is '%s' with length = %d\n", LRS_ans,
      LRS_len);

  // LCS demo, find the LCS of (T, P)
  strcpy(P, "CATA");
  int m = (int)strlen(P);
  strcat(T, P);                              // append P to T
  strcat(T, "#");                            // add '#' at the back
  n = (int)strlen(T);                        // update n

  // reconstruct SA of the combined strings
  SuffixArray S2(T, n);                      // reconstruct SA+LCP
  int split_idx = n-m-1;
  printf("T+P = '%s'\n", T);
  printf(" i SA[i] LCP[i] From  Suffix SA[i]\n");
  for (int i = 0; i < n; ++i)
    printf("%2d    %2d    %2d    %2d    %s\n",
      i, S2.SA[i], S2.LCP[i], S2.SA[i] < split_idx ? 1 : 2, T+
        S2.SA[i]);

  auto [LCS_len, LCS_idx] = S2.LCS(split_idx);
  strncpy(LCS_ans, T+S2.SA[LCS_idx], LCS_len);
  printf("The LCS is '%s' with length = %d\n", LCS_ans,
      LCS_len);

  return 0;
}
```

9 Trees

9.1 LCA

```cpp
/*
Binary lifting:
O(nlogn) para preprocesamiento
O(logn) para cada query
*/
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];

    for (int u : adj[v]) {
        if (u != p)
            dfs(u, v);
    }

    tout[v] = ++timer;
}

bool is_ancestor(int u, int v)
{
    return tin[u] <= tin[v] && tout[u] >= tout[v];
}

int lca(int u, int v)
{
    if (is_ancestor(u, v))
        return u;
    if (is_ancestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v))
            u = up[u][i];
    }
    return up[u][0];
}

void preprocess(int root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<int>(l + 1));
    dfs(root, root);
}
```

10 algorithm

#include <algorithm> #include <numeric>

Algo	**Params**	**Funcion**
sort, stable_sort	f, l	ordena el intervalo
nth_element	f, nth, l	*void* ordena el n-esimo, y particiona el resto
fill, fill_n	f, l / n, elem	*void* llena [f, l) o [f, f+n) con elem
lower_bound, upper_bound	f, l, elem	*it* al primer / ultimo donde se puede insertar elem para que quede ordenada
binary_search	f, l, elem	*bool* esta elem en [f, l)
copy	f, l, resul	hace resul+i=f+i $\forall i$
find, find_if, find_first_of	f, l, elem / pred / f2, l2	*it* encuentra i ∈[f,l) tq. i=elem, pred(i), i∈[f2,l2)
count, count_if	f, l, elem/pred	cuenta elem, pred(i)
search	f, l, f2, l2	busca [f2,l2) ∈ [f,l)
replace, replace_if	f, l, old / pred, new	cambia old / pred(i) por new
reverse	f, l	da vuelta
partition, stable_partition	f, l, pred	pred(i) ad, !pred(i) atras
min_element, max_element	f, l, [comp]	*it* min, max de [f,l]
lexicographical_compare	f1,l1,f2,l2	*bool* con [f1,l1]¡[f2,l2]
next/prev_permutation	f,l	deja en [f,l) la perm sig, ant
set_intersection, set_difference, set_union, set_symmetric_difference,	f1, l1, f2, l2, res	[res, ...) la op. de conj
push_heap, pop_heap, make_heap	f, l, e / e /	mete/saca e en heap [f,l), hace un heap de [f,l)
is_heap	f,l	*bool* es [f,l) un heap
accumulate	f,l,i,[op]	$T = \sum$/oper de [f,l)
inner_product	f1, l1, f2, i	T = i + [f1, l1) . [f2, ...)
partial_sum	f, l, r, [op]	r+i = $\sum$/oper de [f,f+i] $\forall i$ ∈[f,l)
__builtin_ffs	unsigned int	Pos. del primer 1 desde la derecha
__builtin_clz	unsigned int	Cant. de ceros desde la izquierda.
__builtin_ctz	unsigned int	Cant. de ceros desde la derecha.
__builtin_popcount	unsigned int	Cant. de 1's en x.
__builtin_parity	unsigned int	1 si x es par, 0 si es impar.
__builtin_XXXXXXll	unsigned ll	= pero para long long's.

11 Math

11.1 Identidades

$\sum_{i=0}^{n} \binom{n}{i} = 2^n$

$\sum_{i=0}^{n} i\binom{n}{i} = n * 2^{n-1}$

$\sum_{i=m}^{n} i = \frac{n(n+1)}{2} - \frac{m(m-1)}{2} = \frac{(n+1-m)(n+m)}{2}$

$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$

$\sum_{i=0}^{n} i(i-1) = \frac{8}{6}(\frac{n}{2})(\frac{n}{2}+1)(n+1)$ (doubles) $\rightarrow$ Sino ver caso impar y par

$\sum_{i=0}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} = \left[\sum_{i=1}^{n} i\right]^2$

$\sum_{i=0}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n}{30}$

$\sum_{i=0}^{n} i^p = \frac{(n+1)^{p+1}}{p+1} + \sum_{k=1}^{p} \frac{B_k}{p-k+1}\binom{p}{k}(n+1)^{p-k+1}$

$r = e - v + k + 1$

Teorema de Pick: (Area, puntos interiores y puntos en el borde)

$A = I + \frac{B}{2} - 1$

11.2 Ec. Caracteristica

$a_0T(n) + a_1T(n-1) + ... + a_kT(n-k) = 0$

$p(x) = a_0x^k + a_1x^{k-1} + ... + a_k$

Sean $r_1, r_2, ..., r_q$ las raíces distintas, de mult. $m_1, m_2, ..., m_q$

$T(n) = \sum_{i=1}^{q} \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$

11.3 Tablas y cotas (Primos, Divisores, Factoriales, etc)

Factoriales

0! = 1	11! = 39.916.800
1! = 1	12! = 479.001.600 ($\in$ `int`)
2! = 2	13! = 6.227.020.800
3! = 6	14! = 87.178.291.200
4! = 24	15! = 1.307.674.368.000
5! = 120	16! = 20.922.789.888.000
6! = 720	17! = 355.687.428.096.000
7! = 5.040	18! = 6.402.373.705.728.000
8! = 40.320	19! = 121.645.100.408.832.000
9! = 362.880	20! = 2.432.902.008.176.640.000 ($\in$ `tint`)
10! = 3.628.800	21! = 51.090.942.171.709.400.000

max signed tint = 9.223.372.036.854.775.807
max unsigned tint = 18.446.744.073.709.551.615

Primos cercanos a 10^n

9941 9949 9967 9973 10007 10009 10037 10039 10061 10067 10069 10079
99961 99971 99989 99991 100003 100019 100043 100049 100057 100069
999959 999961 999979 999983 1000003 1000033 1000037 1000039
9999943 9999971 9999973 9999991 10000019 10000079 10000103 10000121
99999941 99999959 99999971 99999989 100000007 100000037 100000039 100000049
999999893 999999929 999999937 1000000007 1000000009 1000000021 1000000033

Cantidad de primos menores que 10^n

$\pi(10^1) = 4$; $\pi(10^2) = 25$; $\pi(10^3) = 168$; $\pi(10^4) = 1229$; $\pi(10^5) = 9592$
$\pi(10^6) = 78.498$; $\pi(10^7) = 664.579$; $\pi(10^8) = 5.761.455$; $\pi(10^9) = 50.847.534$
$\pi(10^{10}) = 455.052{,}511$; $\pi(10^{11}) = 4.118.054.813$; $\pi(10^{12}) = 37.607.912.018$

Divisores

Cantidad de divisores (σ_0) para *algunos* $n / \neg\exists n' < n, \sigma_0(n') \geqslant \sigma_0(n)$

$\sigma_0(60) = 12$; $\sigma_0(120) = 16$; $\sigma_0(180) = 18$; $\sigma_0(240) = 20$; $\sigma_0(360) = 24$
$\sigma_0(720) = 30$; $\sigma_0(840) = 32$; $\sigma_0(1260) = 36$; $\sigma_0(1680) = 40$; $\sigma_0(10080) = 72$
$\sigma_0(15120) = 80$; $\sigma_0(50400) = 108$; $\sigma_0(83160) = 128$; $\sigma_0(110880) = 144$
$\sigma_0(498960) = 200$; $\sigma_0(554400) = 216$; $\sigma_0(1081080) = 256$; $\sigma_0(1441440) = 288$
$\sigma_0(4324320) = 384$; $\sigma_0(8648640) = 448$

Suma de divisores (σ_1) para *algunos* $n / \neg\exists n' < n, \sigma_1(n') \geqslant \sigma_1(n)$

$\sigma_1(96) = 252$; $\sigma_1(108) = 280$; $\sigma_1(120) = 360$; $\sigma_1(144) = 403$; $\sigma_1(168) = 480$
$\sigma_1(960) = 3048$; $\sigma_1(1008) = 3224$; $\sigma_1(1080) = 3600$; $\sigma_1(1200) = 3844$
$\sigma_1(4620) = 16128$; $\sigma_1(4680) = 16380$; $\sigma_1(5040) = 19344$; $\sigma_1(5760) = 19890$
$\sigma_1(8820) = 31122$; $\sigma_1(9240) = 34560$; $\sigma_1(10080) = 39312$; $\sigma_1(10920) = 40320$
$\sigma_1(32760) = 131040$; $\sigma_1(35280) = 137826$; $\sigma_1(36960) = 145152$; $\sigma_1(37800) = 148800$
$\sigma_1(60480) = 243840$; $\sigma_1(64680) = 246240$; $\sigma_1(65520) = 270816$; $\sigma_1(70560) = 280098$
$\sigma_1(95760) = 386880$; $\sigma_1(98280) = 403200$; $\sigma_1(100800) = 409448$
$\sigma_1(491400) = 2083200$; $\sigma_1(498960) = 2160576$; $\sigma_1(514080) = 2177280$
$\sigma_1(982800) = 4305280$; $\sigma_1(997920) = 4390848$; $\sigma_1(1048320) = 4464096$
$\sigma_1(4979520) = 22189440$; $\sigma_1(4989600) = 22686048$; $\sigma_1(5045040) = 23154768$
$\sigma_1(9896040) = 44323200$; $\sigma_1(9959040) = 44553600$; $\sigma_1(9979200) = 45732192$