/\*\*\*\*\*\*\*\*\*\*\* MANACHER ALGORTIHM O(n)\*\*\*\*\*\*\*\*\*\*\*/

CALCULA EL MAXIMO PALIDROMO CENTRADO EN i (d[i])

centrado en i impar (d1[i])

centrado en i-1 , i par (d2[i])

int n; // tamaÃ±o del string s

string s; // cadena

void rec()

{

vector<int> d1 (n);

int l=0, r=-1;

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

int k = (i>r ? 0 : min (d1[l+r-i], r-i)) + 1;

while (i+k < n && i-k >= 0 && s[i+k] == s[i-k]) ++k;

d1[i] = --k;

if (i+k > r)

l = i-k, r = i+k;

}

vector<int> d2 (n);

l=0, r=-1;

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

int k = (i>r ? 0 : min (d2[l+r-i+1], r-i+1)) + 1;

while (i+k-1 < n && i-k >= 0 && s[i+k-1] == s[i-k]) ++k;

d2[i] = --k;

if (i+k-1 > r)

l = i-k, r = i+k-1;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*LCA (O(nlogn))\*\*\*\*\*\*\*\*\*\*\*\*\*/

//padre[source of the tree] = -1 cuidado!!!!

int dp[N][17] , parent[N] , level[N] , dist[N];

int n;

void init(){

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

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

dp[i][j] = -1;

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

dp[i][0] = parent[i];

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

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

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

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

}

//level de menor a mayor cuidado (level[source] = 0 )

int lca(int a,int b){

if(level[a] < level[b] )

swap(a,b);

int log;

for(log = 1; level[a] - (1<<log) >= 0; log++);

log--;

for(int i = log; i >= 0; i--)

if( level[a] - (1<<i) >= level[b] )

a = dp[a][i];

if(a == b) return a;

for(int i = log; i >=0 ; i--)

if(dp[a][i] != -1 && dp[a][i] != dp[b][i])

a = dp[a][i] , b = dp[b][i];

return parent[a];

}

int path(int a,int b,int num){ // KTH node a->b (0\_idx)

int x = lca(a,b);

if( level[a] - num < level[x] )

{ num = level[a] + level[b] - 2\*level[x] - num ;

a = b;

}

int log;

for(log = 1;level[a] - (1<<log) >= 0; log++);

log--;

num = level[a] - num;

for(int i = log; i>=0 ; i--)

if(level[a] - (1<<i) >= num )

a = dp[a][i];

return a;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*SUFFIX ARRAY O(nlogn) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//lcp[i] = suffix(i,i+1)

#define checkMod(i, n) (((i) < (n)) ? (i) : (i) - (n))

#define ALPH\_SIZE 256

char s[MAXN];

int n;

int SA[MAXN], lcp[MAXN], cnt[MAXN], c[MAXN];

int pn[MAXN], cn[MAXN];

void build\_suffix\_array()

{

memset(cnt, 0, ALPH\_SIZE \* sizeof(int));

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

for(int i=1; i<ALPH\_SIZE; ++i) cnt[i] += cnt[i-1];

for(int i=0; i<n; ++i) SA[--cnt[(int)s[i]]] = i;

c[SA[0]] = 0;

int classes = 1;

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

if(s[SA[i]] != s[SA[i-1]]) ++classes;

c[SA[i]] = classes-1;

}

for(int h=0; (1<<h)<n; ++h){

for(int i=0; i<n; ++i) pn[i] = checkMod(SA[i] - (1<<h) + n, n);

memset(cnt, 0, classes \* sizeof(int));

for(int i=0; i<n; ++i) ++cnt[c[pn[i]]];

for(int i=1; i<classes; ++i) cnt[i] += cnt[i-1];

for(int i=n-1; i>=0; i--) SA[--cnt[c[pn[i]]]] = pn[i];

for(int i=0; i<n; ++i) pn[i] = checkMod(SA[i] + (1<<h), n);

cn[SA[0]] = 0;

classes = 1;

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

if(c[SA[i]] != c[SA[i-1]] || c[pn[i]] != c[pn[i-1]]) ++classes;

cn[SA[i]] = classes-1;

}

memcpy(c, cn, n \* sizeof(int));

}

}

void build\_lcp() {

int k = 0;

for(int i = 0; i < n; i++) if (c[i]) {

int j = SA[c[i] - 1];

while(s[i + k] == s[j + k]) k++;

lcp[c[i] - 1] = k;

if (k) k--;

}

lcp[n - 1] = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*SUFFIX ARRAY O(n) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

inline bool leq(int a1, int a2, int b1, int b2) { // lexicographic order

return(a1 < b1 || a1 == b1 && a2 <= b2); // for pairs

}

inline bool leq(int a1, int a2, int a3, int b1, int b2, int b3) {

return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3)); // and triples

}

// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r

static void radixPass(int\* a, int\* b, int\* r, int n, int K) {

// count occurrences

int\* c = new int[K + 1]; // counter array

for (int i = 0; i <= K; i++) c[i] = 0; // reset counters

for (int i = 0; i < n; i++) c[r[a[i]]]++; // count occurrences

for (int i = 0, sum = 0; i <= K; i++) { // exclusive prefix sums

int t = c[i];

c[i] = sum;

sum += t;

}

for (int i = 0; i < n; i++) b[c[r[a[i]]]++] = a[i]; // sort

delete [] c;

}

// find the suffix array SA of T[0..n-1] in {1..K}^n

// require T[n]=T[n+1]=T[n+2]=0, n>=2

void suffixArray(int\* T, int\* SA, int n, int K) {

int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;

int\* R = new int[n02 + 3];

R[n02]= R[n02+1]= R[n02+2]=0;

int\* SA12 = new int[n02 + 3];

SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;

int\* R0 = new int[n0];

int\* SA0 = new int[n0];

//\*\*\*\*\*\*\* Step 0: Construct sample \*\*\*\*\*\*\*\*

// generate positions of mod 1 and mod 2 suffixes

// the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1

for (int i=0, j=0; i < n+(n0-n1); i++) if (i%3 != 0) R[j++] = i;

//\*\*\*\*\*\*\* Step 1: Sort sample suffixes \*\*\*\*\*\*\*\*

// lsb radix sort the mod 1 and mod 2 triples

radixPass(R , SA12, T+2, n02, K);

radixPass(SA12, R , T+1, n02, K);

radixPass(R , SA12, T , n02, K);

// find lexicographic names of triples and

// write them to correct places in R

int name = 0, c0 = -1, c1 = -1, c2 = -1;

for (int i = 0; i < n02; i++) {

if (T[SA12[i]] != c0 || T[SA12[i]+1] != c1 || T[SA12[i]+2] != c2) {

name++;

c0 = T[SA12[i]];

c1 = T[SA12[i]+1];

c2 = T[SA12[i]+2];

}

if (SA12[i] % 3 == 1) {

R[SA12[i]/3] = name; // write to R1

} else {

R[SA12[i]/3 + n0] = name; // write to R2

}

}

// recurse if names are not yet unique

if (name < n02) {

suffixArray(R, SA12, n02, name);

// store unique names in R using the suffix array

for (int i = 0; i < n02; i++) R[SA12[i]] = i + 1;

} else // generate the suffix array of R directly

for (int i = 0; i < n02; i++) SA12[R[i] - 1] = i;

//\*\*\*\*\*\*\* Step 2: Sort nonsample suffixes \*\*\*\*\*\*\*\*

// stably sort the mod 0 suffixes from SA12 by their first character

for (int i=0, j=0; i < n02; i++) if (SA12[i] < n0) R0[j++] = 3\*SA12[i];

radixPass(R0, SA0, T, n0, K);

//\*\*\*\*\*\*\* Step 3: Merge \*\*\*\*\*\*\*\*

// merge sorted SA0 suffixes and sorted SA12 suffixes

for (int p=0, t=n0-n1, k=0; k < n; k++) {

#define GetI() (SA12[t] < n0 ? SA12[t] \* 3 + 1 : (SA12[t] - n0) \* 3 + 2)

int i = GetI(); // pos of current offset 12 suffix

int j = SA0[p]; // pos of current offset 0 suffix

if (SA12[t] < n0 ? // different compares for mod 1 and mod 2 suffixes

leq(T[i], R[SA12[t] + n0], T[j], R[j/3]) :

leq(T[i],T[i+1],R[SA12[t]-n0+1], T[j],T[j+1],R[j/3+n0])) {

// suffix from SA12 is smaller

SA[k] = i;

t++;

if (t == n02) // done --- only SA0 suffixes left

for (k++; p < n0; p++, k++) SA[k] = SA0[p];

} else { // suffix from SA0 is smaller

SA[k] = j;

p++;

if (p == n0) // done --- only SA12 suffixes left

for (k++; t < n02; t++, k++) SA[k] = GetI();

}

}

delete [] R;

delete [] SA12;

delete [] SA0;

delete [] R0;

}

char T[MAXN];

int n;

int main() {

n = strlen(T);

int\* SA = new int[n + 3];

int\* TT = new int[n + 3];

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

TT[i] = T[i];

TT[n] = TT[n+1] = TT[n+2] = 0;

suffixArray(TT,SA,n,100000);

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MAX FLOW\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//EDMON KARP O(V \* E^2)

vector<vint> adj;

int n;

int source, sink;

int residual[200][200];

int find\_path()

{

bool visited[n];

memset(visited, 0, sizeof(visited));

int from[n];

memset(from, -1, sizeof(from));

queue <int> Q;

Q.push(source);

visited[source] = 1;

int where;

while(!Q.empty())

{

where = Q.front();

Q.pop();

if(where==sink) break;

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

{ int i = adj[where][j];

if(residual[where][i] > 0 && !visited[i])

{

Q.push(i);

visited[i] = 1;

from[i] = where;

}

}

}

int path\_cap = 1<<30;

where = sink;

while(from[where] > -1)

{

int prev = from[where];

path\_cap = min(path\_cap, residual[prev][where]);

where = prev;

}

where = sink;

while(from[where] > -1)

{

int prev = from[where];

residual[prev][where] -= path\_cap;

residual[where][prev] += path\_cap;

where = prev;

}

if(path\_cap == 1<<30) return 0;

else return path\_cap;

}

int max\_flow()

{

int flow = 0;

while(1)

{

int path\_cap = find\_path();

if(!path\_cap) break;

else flow += path\_cap;

}

return flow;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MAX FLOW \*\*\*\*\*\*\*\*\*\*\*\*\*/

// Adjacency list implementation of Dinic's blocking flow algorithm.

// This is very fast in practice, and only loses to push-relabel flow.

//

// Running time:

// O(|V|^2 |E|)

//

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

//

// OUTPUT:

// - maximum flow value

// - To obtain the actual flow values, look at all edges with

// capacity > 0 (zero capacity edges are residual edges).

struct Edge {

int from, to, cap, flow, index;

Edge(int from, int to, int cap, int flow, int index) :

from(from), to(to), cap(cap), flow(flow), index(index) {}

};

struct Dinic {

int N;

vector<vector<Edge> > G;

vector<Edge \*> dad;

vector<int> Q;

Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

void AddEdge(int from, int to, int cap) {

G[from].push\_back(Edge(from, to, cap, 0, G[to].size()));

if (from == to) G[from].back().index++;

G[to].push\_back(Edge(to, from, 0, 0, G[from].size() - 1));

}

long long BlockingFlow(int s, int t) {

fill(dad.begin(), dad.end(), (Edge \*) NULL);

dad[s] = &G[0][0] - 1;

int head = 0, tail = 0;

Q[tail++] = s;

while (head < tail) {

int x = Q[head++];

for (int i = 0; i < (int)G[x].size(); i++) {

Edge &e = G[x][i];

if (!dad[e.to] && e.cap - e.flow > 0) {

dad[e.to] = &G[x][i];

Q[tail++] = e.to;

}

}

}

if (!dad[t]) return 0;

long long totflow = 0;

for (int i = 0; i < (int)G[t].size(); i++) {

Edge \*start = &G[G[t][i].to][G[t][i].index];

int amt = INF;

for (Edge \*e = start; amt && e != dad[s]; e = dad[e->from]) {

if (!e) { amt = 0; break; }

amt = min(amt, e->cap - e->flow);

}

if (amt == 0) continue;

for (Edge \*e = start; amt && e != dad[s]; e = dad[e->from]) {

e->flow += amt;

G[e->to][e->index].flow -= amt;

}

totflow += amt;

}

return totflow;

}

long long GetMaxFlow(int s, int t) {

long long totflow = 0;

while (long long flow = BlockingFlow(s, t))

totflow += flow;

return totflow;

}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*PUENTES Y ARTICULACIONES\*\*\*\*\*\*\*\*///

vector<vpii> adj;

int counter, dfsRoot, rootChildren;

int num[N] , low[N] , parent[N];

int n; // numero de vertices

bool articulation[N];

void dfs(int u) {

low[u] = num[u] = counter++; // dfs\_low[u] <= dfs\_num[u]

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

pii v = adj[u][j];

if (num[v.first] == -1) { // a tree edge

parent[v.first] = u;

if (u == dfsRoot) rootChildren++; // special case, count children of root

dfs(v.first);

if (low[v.first] >= num[u]) // for articulation point

articulation[u] = true; // store this information first

if (low[v.first] > num[u]) // for bridge

printf(" Edge (%d, %d) is a bridge\n", u, v.first);

low[u] = min(low[u], low[v.first]); // update dfs\_low[u]

}

else if (v.first != parent[u]) // a back edge and not direct cycle

low[u] = min(low[u], num[v.first]); // update dfs\_low[u]

} }

int main() {

f(i,0,n)

{ num[i] = -1;

low[i] = 0;

parent[i] = -1;

articulation[i] = false;

}

counter = 0;

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

if (num[i] == -1) {

dfsRoot = i; rootChildren = 0;

dfs(i);

articulation[dfsRoot] = (rootChildren > 1); // special case

}

return 0;

}

/\*\*\*\*\*\* STRONGLY COMPONENTES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

vint S; // additional global variables

bool vis[N] ;

int dfs\_low[N] , dfs\_num[N];

int numSCC ,dfsNumberCounter ;

vector<vpii> adj , G;

int id[N];

void tarjanSCC(int u) {

dfs\_low[u] = dfs\_num[u] = dfsNumberCounter++; // dfs\_low[u] <= dfs\_num[u]

S.pb(u); // stores u in a vector based on order of visitation

vis[u] = true;

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

pii v = adj[u][j];

if (dfs\_num[v.first] == -1)

tarjanSCC(v.first);

if (vis[v.first]) // condition for update

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

}

if (dfs\_low[u] == dfs\_num[u]) { // if this is a root (start) of an SCC

// printf("SCC %d:", numSCC); // this part is done after recursion

while (1) {

int v = S.back(); S.pop\_back(); vis[v] = false;

// printf(" %d", v);

id[v] = numSCC;

if (u == v) break;

}

++numSCC;

//printf("\n");

}

}

/\*\*\*\*\*\*\*\*\*\*\*HASH\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

HASH pot[MAXN];

HASH HA[MAXN];

char a[MAXN];

int tama;

// es el hash de [a,b>

HASH readA(int a,int b){ return HA[b] - HA[a]\*pot[b-a]; }

int main()

{

pot[0] = 1;

f(i,1,MAXN)

pot[i] = pot[i-1] \* B;

f(i,1,tama+1)

HA[i] = HA[i-1] \* B + a[i-1]-'a'+1;

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MATCHING \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

CAMINOS AUMENTATIVOS O (E \* V)

int le , ri; // cantidad nodo left and right

vector<vint> adj;

int dueno[N]; // es de todo le + ri

int vis[N]; // solo es le

int path(int x){

if(vis[x]) return 0;

vis[x] = 1;

f(j,0,adj[x].size())

{ int where = adj[x][j];

if(dueno[where]==-1 || path(dueno[where]))

{ dueno[where] = x;

return 1;

}

}

return 0;

}

int matching(){

int ans = 0;

f(i,0,le + ri) dueno[i] = -1;

f(i,0,le){

f(j,0,le) vis[j] = false;

ans += path(i);

}

return ans;

}

/\*\*\*\*\*\*\*\*\*\*MATCHING \*\*\*\*\*\*\*\*\*\*/

HOPCROFT O (E \* sqrt(V))

vint adj[N + 1]; // (u, v) <=> (v, u)

int n, m, NIL, match[N + 1], dist[N + 1];

// Izquierda; nodos del 0 al n-1

// Derecha: Nodos del n al n+m-1

// NIL: Nodo n+m

bool bfs(){

queue <int> Q;

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

if(match[i] == NIL) {

dist[i] = 0;

Q.push(i);

}

else dist[i] = INF;

}

dist[NIL] = INF;

while(!Q.empty()) {

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

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

int v = adj[u][i];

if(dist[match[v]] == INF) {

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

Q.push(match[v]);

}

}

}

return dist[NIL] != INF;

}

bool dfs(int u) {

if(u != NIL) {

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

int v = adj[u][i];

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

if(dfs(match[v])) {

match[v] = u;

match[u] = v;

return true;

}

}

}

dist[u] = INF;

return false;

}

return true;

}

int hopcroft\_karp()

{

NIL = n + m;

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

match[i] = NIL;

int matching = 0;

//Greedy Step

for(int u=0; u<n; u++)

{

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

{

int v = adj[u][i];

if(match[v] == NIL)

{

matching++;

match[u] = v;

match[v] = u;

break;

}

}

}

while(bfs())

for(int u=0; u<n; u++)

if(match[u] == NIL && dfs(u))

matching++;

return matching;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*KMP\*\*\*\*\*\*\*\*\*\*\*\*/

char T[N], P[N]; // T = text, P = pattern

int b[N], n, m; // b = back table, n = length of T, m = length of P

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

int i = 0, j = -1; b[0] = -1;

while (i < m) {

while (j >= 0 && P[i] != P[j]) j = b[j];

i++; j++;

b[i] = j;

} }

void kmpSearch() {

int i = 0, j = 0;

while (i < n) {

while (j >= 0 && T[i] != P[j]) j = b[j];

i++; j++;

if (j == m) {

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

j = b[j];

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*BIT\*\*\*\*\*\*\*\*\*\*\*\*/

//empieza en idx = 0

// BIT 1D

int n;

int t[N], value[N];

int sum(int x)

{

int result = 0;

for(int i = x; i >= 0; i = (i & (i+1)) - 1)

result += t[i];

return result;

}

void inc(int x, int delta)

{

for(int i = x; i < n; i = (i | (i+1)))

t[i] += delta;

}

// BIT 2D

int n, m;

int t[1024][1024], value[1024][1024];

int sum(int x, int y)

{

int result = 0;

for(int i = x; i >= 0; i = (i & (i+1)) - 1)

for(int j = y; j >= 0; j = (j & (j+1)) - 1)

result += t[i][j];

return result;

}

void inc(int x, int y, int delta)

{

for(int i = x; i < n; i = (i | (i+1)))

for(int j = y; j < m; j = (j | (j+1)))

t[i][j] += delta;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*+BIT\*\*\*\*\*\*\*\*\*\*\*\*/

empieza en idx = 1

//1D

int tree[N], n ;

int read(int idx){

int sum = 0;

for( ; idx > 0 ; idx -= (idx & -idx))

sum += tree[idx];

return sum;

}

void update(int idx ,int val){

for (;idx <= n; idx += (idx & -idx))

tree[idx] += val;

}

//2D

int tree[N][N];

int n;

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

for(int i = x;i <= n; i += i&-i)

for(int j = y; j <= n; j += j&-j)

tree[i][j] += val;

}

int read(int x,int y){

int res = 0;

for(int i = x; i > 0 ; i -= i&-i)

for(int j = y; j > 0; j -= j&-j)

res += tree[i][j];

return res;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MATEMATICAS\*\*\*\*\*\*\*\*\*\*\*\*\*/

CRIBA NORMAL

ll n;

bitset<10000010> bs; // 10^7 should be enough for most cases

vint primes; // compact list of primes in form of vector<int>

// first part

void sieve() { // create list of primes in [0..upperbound]

bs.set(); // set all bits to 1

bs[0] = bs[1] = 0; // except index 0 and 1

for (ll i = 2; i <= n; i++) if (bs[i]) {

// cross out multiples of i starting from i \* i!

for (ll j = i \* i; j <= n; j += i) bs[j] = 0;

primes.push\_back((int)i); // also add this vector containing list of primes

} } // call this method in main method

bool isPrime(ll N) { // a good enough deterministic prime tester

if (N <= n) return bs[N]; // O(1) for small primes

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

if (N % primes[i] == 0) return false;

return true; // it takes longer time if N is a large prime!

} // note: only work for N <= (last prime in vi "primes")^2

// second part

vint primeFactors(ll N) { // remember: vi is vector of integers, ll is long long

vint factors; // vi `primes' (generated by sieve) is optional

ll PF\_idx = 0, PF = primes[PF\_idx]; // using PF = 2, 3, 4, ..., is also ok

while (N != 1 && (PF \* PF <= N)) { // stop at sqrt(N), but N can get smaller

while (N % PF == 0) { N /= PF; factors.push\_back(PF); } // remove this PF

PF = primes[++PF\_idx]; // only consider primes!

}

if (N != 1) factors.push\_back(N); // special case if N is actually a prime

return factors; // if pf exceeds 32-bit integer, you have to change vi

}

// third part

ll numPF(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

while (N != 1 && (PF \* PF <= N)) {

while (N % PF == 0) { N /= PF; ans++; }

PF = primes[++PF\_idx];

}

if (N != 1) ans++;

return ans;

}

ll numDiffPF(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

while (N != 1 && (PF \* PF <= N)) {

if (N % PF == 0) ans++; // count this pf only once

while (N % PF == 0) N /= PF;

PF = primes[++PF\_idx];

}

if (N != 1) ans++;

return ans;

}

ll sumPF(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 0;

while (N != 1 && (PF \* PF <= N)) {

while (N % PF == 0) { N /= PF; ans += PF; }

PF = primes[++PF\_idx];

}

if (N != 1) ans += N;

return ans;

}

ll numDiv(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 1; // start from ans = 1

while (N != 1 && (PF \* PF <= N)) {

ll power = 0; // count the power

while (N % PF == 0) { N /= PF; power++; }

ans \*= (power + 1); // according to the formula

PF = primes[++PF\_idx];

}

if (N != 1) ans \*= 2; // (last factor has pow = 1, we add 1 to it)

return ans;

}

ll sumDiv(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = 1; // start from ans = 1

while (N != 1 && (PF \* PF <= N)) {

ll power = 0;

while (N % PF == 0) { N /= PF; power++; }

ans \*= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1); // formula

PF = primes[++PF\_idx];

}

if (N != 1) ans \*= ((ll)pow((double)N, 2.0) - 1) / (N - 1); // last one

return ans;

}

ll EulerPhi(ll N) {

ll PF\_idx = 0, PF = primes[PF\_idx], ans = N; // start from ans = N

while (N != 1 && (PF \* PF <= N)) {

if (N % PF == 0) ans -= ans / PF; // only count unique factor

while (N % PF == 0) N /= PF;

PF = primes[++PF\_idx];

}

if (N != 1) ans -= ans / N; // last factor

return ans;

}

//CRIBA EN O(n)

int n , lp[N] , primes[N] ,sz;

void sieve(){

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

if (lp[i] == 0) {

lp[i] = i;

primes[sz++] = i;

}

for(int j = 0; j < sz && primes[j] <= lp[i] && (long long)i\*primes[j] <= n; ++j)

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

}

}

// return a % b (positive value)

int mod(int a, int b) {

return ((a%b)+b)%b;

}

// computes gcd(a,b)

int gcd(int a, int b) {

int tmp;

while(b){a%=b; tmp=a; a=b; b=tmp;}

return a;

}

// computes lcm(a,b)

int lcm(int a, int b) {

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

}

// returns d = gcd(a,b); finds x,y such that d = ax + by

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

int xx = y = 0;

int yy = x = 1;

while (b) {

int q = a/b;

int t = b; b = a%b; a = t;

t = xx; xx = x-q\*xx; x = t;

t = yy; yy = y-q\*yy; y = t;

}

return a;

}

// finds all solutions to ax = b (mod n)

vint modular\_linear\_equation\_solver(int a, int b, int n) {

int x, y;

vint solutions;

int d = extended\_euclid(a, n, x, y);

if (!(b%d)) {

x = mod (x\*(b/d), n);

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

solutions.push\_back(mod(x + i\*(n/d), n));

}

return solutions;

}

// computes b such that ab = 1 (mod n), returns -1 on failure

int mod\_inverse(int a, int n) {

int x, y;

int d = extended\_euclid(a, n, x, y);

if (d > 1) return -1;

return mod(x,n);

}

// Chinese remainder theorem (special case): find z such that

// z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).

// Return (z,M). On failure, M = -1.

pii chinese\_remainder\_theorem(int x, int a, int y, int b) {

int s, t;

int d = extended\_euclid(x, y, s, t);

if (a%d != b%d) return make\_pair(0, -1);

return make\_pair(mod(s\*b\*x+t\*a\*y,x\*y)/d, x\*y/d);

}

// Chinese remainder theorem: find z such that

// z % x[i] = a[i] for all i. Note that the solution is

// unique modulo M = lcm\_i (x[i]). Return (z,M). On

// failure, M = -1. Note that we do not require the a[i]'s

// to be relatively prime.

pii chinese\_remainder\_theorem(const vint &x, const vint &a) {

pii ret = make\_pair(a[0], x[0]);

for (int i = 1; i < (int)x.size(); i++) {

ret = chinese\_remainder\_theorem(ret.second, ret.first, x[i], a[i]);

if (ret.second == -1) break;

}

return ret;

}

// computes x and y such that ax + by = c; on failure, x = y =-1

void linear\_diophantine(int a, int b, int c, int &x, int &y) {

int d = gcd(a,b);

if (c%d) {

x = y = -1;

} else {

x = c/d \* mod\_inverse(a/d, b/d);

y = (c-a\*x)/b;

}

}

int main(){

// expected: 2

cout << gcd(14, 30) << endl;

// expected: 2 -2 1

int x, y;

int d = extended\_euclid(14, 30, x, y);

cout << d << " " << x << " " << y << endl;

// expected: 95 45

vint sols = modular\_linear\_equation\_solver(14, 30, 100);

for (int i = 0; i < (int) sols.size(); i++) cout << sols[i] << " ";

cout << endl;

// expected: 8

cout << mod\_inverse(8, 9) << endl;

// expected: 23 56

// 11 12

int xs[] = {3, 5, 7, 4, 6};

int as[] = {2, 3, 2, 3, 5};

pii ret = chinese\_remainder\_theorem(vint (xs, xs+3), vint(as, as+3));

cout << ret.first << " " << ret.second << endl;

ret = chinese\_remainder\_theorem (vint(xs+3, xs+5), vint(as+3, as+5));

cout << ret.first << " " << ret.second << endl;

// expected: 5 -15

linear\_diophantine(7, 2, 5, x, y);

cout << x << " " << y << endl;

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*TEOREMA CHINO DEL RESTO , FAST EXP Y INVERSO MODULAR \*\*\*\*\*\*\*/

int exp(int a,int b,int c){

int ans = 1;

while(b){

if(b&1) ans = ans \* a % c;

a = a \* a % c;

b>>=1;

}

return ans;

}

int inv(int a,int b){

return exp(a,b-2,b);

}

//todo bi es primo (hacer con euler para el otro caso)

//dados bi,coprimos para todo (bi,bj) 1 <= i < j <= k

// dados a = ai % bi

//retorna a % (b1\*b2\*...\*bk)

int teorema\_chino(vint bases,vint ai){

int k = bases.size();

int prod = 1;

f(i,0,k)

prod \*= bases[i];

int b = prod;

int ans = 0;

f(i,0,k){

int m = prod / bases[i];

ans = (ans + ai[i] \* m \* inv(m , bases[i] ) )%b;

}

return ans;

}

int main()

{

vint bases;

bases.pb(2);

bases.pb(5);

vint ai;

ai.pb(0);

ai.pb(2);

int ans = teorema\_chino(bases,ai);

bug1(ans);

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*GAUSS ECHELON \*\*\*\*\*\*\*\*\*\*\*\*\*/

int cmp(double a,double b){

return a - EPS > b ? 1: a + EPS < b ? -1:0;

}

double a[N][N+1] , x[N];

// a[i][n] es la matriz b[i]

//matrix a coeficientes

//[a:b] matriz aumentada

//m = numeros de ecuaciones n = numeros de variables

//matrid de m\*(n+1)

void gauss(int m,int n){

for(int row = 0 , colum = 0; row < m && colum < n; colum++){

bool found = false;

for(int newrow = row; newrow < m; newrow++)

if( cmp(a[newrow][colum],0)!= 0 )

{ swap(a[row],a[newrow]);

found = true;

break;

}

if(found == false) continue;

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

{ double k = a[i][colum] / a[row][colum];

for(int j = colum; j <= n; ++j)

a[i][j] -= a[row][j] \* k;

}

row++;

}

int ra = 0;

f(i,0,m)

{ bool ok = false;

f(j,0,n)

if( cmp(a[i][j],0)!=0)

ok = true;

ra += ok;

}

int rab = 0;

f(i,0,m)

{ bool ok = false;

f(j,0,n+1)

if(cmp(a[i][j],0)!=0)

ok = true;

rab += ok;

}

if(ra != rab){}// puts("no solution")};

else

if( ra == n){

fd(i,n-1,0)

{ double aux = 0;

fd(j,n-1,i+1)

aux += x[j]\* a[i][j];

x[i] = (a[i][n] - aux) / a[i][i];

}

}

else {}; //puts("infinitas soluciones");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*AHO CORASICK\*\*\*\*/

bool ispal[MAXN];

int tree[MAXN][26] , T[MAXN] , padre[MAXN];

int node;

void add(string a){

int n = a.size() , p = 0;

f(i,0,n){

if( tree[p][a[i]-'a'] == 0) tree[p][a[i]-'a'] = node++;

padre[tree[p][a[i]-'a']] = p;

p = tree[p][a[i]-'a'];

}

ispal[p] = true;

}

void aho(){

queue<int> Q;

int where = 0;

f(i,0,26)

if(tree[where][i]){

Q.push(tree[where][i]); // tamaÃ±o 1

T[tree[where][i]] = 0; // apunta al vacio

}

while(!Q.empty()){

where = Q.front(); Q.pop();

f(i,0,26)

if(tree[where][i]){

int v = tree[where][i];

int p = T[where]; // link de where

while(p && tree[p][i]==0)

p = T[p];

T[v] = tree[p][i];

Q.push(v);

}

}

}

int dp[MAXN] , c[MAXN] , caso;

int solve(int p){

if(p==0) return 0;

int &ans = dp[p] , &aux = c[p];

if(aux==caso) return dp[p];

aux = caso;

return ans = ispal[p] + max(solve(padre[p]),solve(T[p]));

}

int solve2(int p){

int ans = 0;

if(ispal[p])

ans = solve(p);

f(i,0,26)

if(tree[p][i])

ans = max( ans , solve2(tree[p][i]));

return ans;

}

int main(){

freopen("in.c","r",stdin);

int n;

string a;

while(cin >> n , n ){

node = 1;

clr(ispal,0);

clr(tree,0);

f(i,0,n)

{ cin >> a;

add(a);

}

aho();

caso++;

cout << solve2(0) << endl;

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* F \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

vint S; // additional global variables

bool vis[N] ;

int dfs\_low[N] , dfs\_num[N];

int numSCC ,dfsNumberCounter ;

vector<vpii> adj , G;

int id[N];

void tarjanSCC(int u) {

dfs\_low[u] = dfs\_num[u] = dfsNumberCounter++; // dfs\_low[u] <= dfs\_num[u]

S.pb(u); // stores u in a vector based on order of visitation

vis[u] = true;

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

pii v = adj[u][j];

if (dfs\_num[v.first] == -1)

tarjanSCC(v.first);

if (vis[v.first]) // condition for update

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

}

if (dfs\_low[u] == dfs\_num[u]) { // if this is a root (start) of an SCC

// printf("SCC %d:", numSCC); // this part is done after recursion

while (1) {

int v = S.back(); S.pop\_back(); vis[v] = false;

// printf(" %d", v);

id[v] = numSCC;

if (u == v) break;

}

++numSCC;

//printf("\n");

} }

void build(int x){

vis[x] = true;

f(j,0,adj[x].size()){

pii where = adj[x][j];

if(id[x] != id[where.fst])

G[id[x]].pb(pii(id[where.fst],where.snd));

if(!vis[where.fst])

build(where.fst);

}

}

void dfs2(int x){

vis[x] = true;

f(j,0,G[x].size()){

pii where = G[x][j];

if(!vis[where.fst])

dfs2(where.fst);

}

S.pb(x);

}

ll dp[N];

int main(){

int tc , n , m , w ,x ,y;

sc(tc);

while(tc--){

scanf("%d %d",&n,&m);

adj.assign(n,vpii());

f(i,0,m){

scanf("%d %d %d",&x,&y,&w); x--; y--;

adj[x].pb(pii(y,w));

}

f(i,0,n)

{ dfs\_num[i] = -1;

dfs\_low[i] = 0;

vis[i] = false;

}

dfsNumberCounter = numSCC = 0;

f(i,0,n)

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

tarjanSCC(i);

G.assign(numSCC,vpii());

f(i,0,n)

vis[i] = false;

f(i,0,n)

if(!vis[i])

build(i);

/\*\*\*\*\*\*\*\*\*SCC\*///

n = numSCC;

f(i,0,n)

vis[i] = false;

S.clear();

f(i,0,n)

if(!vis[i])

dfs2(i);

reverse(all(S));

f(i,0,n)

dp[i] = 0;

f(i,0,n)

{ int v = S[i];

f(j,0,G[v].size()){

pii where = G[v][j];

dp[where.fst] = max(dp[where.fst] , where.snd + dp[v]);

}

}

ll ans = 0;

f(i,0,n)

ans = max(ans,dp[i]);

cout << ans << endl;

}

return 0;

}