struct lex\_compare {

bool operator() (const int64\_t& lhs, const int64\_t& rhs) const{

stringstream s1,s2;

s1 << lhs;

s2 << rhs;

return s1.str() < s2.str();

}

};

You then use the class name as the type parameter

set<int64\_t, lex\_compare> s;

#include <iostream>

#include <set>

struct C

{

bool operator()(const int &a, const int &b) const

{

return a % 10 < b % 10;

}

};

std::set<int, C> y(x.begin(), x.end());

-----#include <bits/stdc++.h> #define Max 100050 #define cons 1000000000+7 #define mp make\_pair #define pb push\_back #define INF 1e16 #define INF2 1e9+9

#define pi 3.141592653589 #define x first #define y second----

-------priority\_queue<pair<int,long>,vector<pair<int,long> >,COMPARE> pq;-----

For set: erase,insert

Pq: push, pop, top.

bool operator()(pair<int,long> a,pair<int,long> b)

{return a.second>b.second;}

printf("Character is %c \n", ch);

   printf("String is %s \n" , str);

   printf("Float value is %f \n", flt);

   printf("Integer value is %d\n" , no);

   printf("Double value is %lf \n", dbl);

   printf("Octal value is %o \n", no);

long long int x = 0;

unsigned long long int y = 0;

printf("%lld\n", x);

printf("%llu\n", y);

printf("When this number: %f is assigned to 2 dp, it will be: %.2f ", 94.9456, 94.9456);

scanf("%d",&testInteger);

scanf("%c",&chr);

#include <iostream>

using namespace std;

int main()

{

char items[20];

cin.getline(items, 20);

cout << items;

}

Not 100% sure if this works on a string array, but try this -  
getline(std::cin, items[0]);

#include <string>

std::string s = std::to\_string(42);

int a = 10;

stringstream ss;

ss << a;

string str = ss.str();

An array "decays" into a pointer to its first element, so scanf("%s", string) is equivalent to scanf("%s", &string[0]). On the other hand, scanf("%s", &string) passes a pointer-to-char[256], but it points to the same place.

* struct Point
* {long long int x, y,price;};
* Point p0;
* Point nextToTop(stack<Point> &S)
* {
* Point p = S.top();
* S.pop();
* Point res = S.top();
* S.push(p);
* return res;
* }
* long long int swap(Point &p1, Point &p2)
* {
* Point temp = p1;
* p1 = p2;
* p2 = temp;
* }
* long long int distSq(Point p1, Point p2)
* {
* return (p1.x - p2.x)\*(p1.x - p2.x) +
* (p1.y - p2.y)\*(p1.y - p2.y);
* }
* long long int orientation(Point p, Point q, Point r)
* {
* long long int val = (q.y - p.y) \* (r.x - q.x) -
* (q.x - p.x) \* (r.y - q.y);
* if (val == 0) return 0; // colinear
* return (val > 0)? 1: 2; // clock or counterclock wise
* }
* int compare(const void \*vp1, const void \*vp2)
* {
* Point \*p1 = (Point \*)vp1;
* Point \*p2 = (Point \*)vp2;
* // Find orientation
* long long int o = orientation(p0, \*p1, \*p2);
* if (o == 0)
* return (distSq(p0, \*p2) >= distSq(p0, \*p1))? -1 : 1;
* return (o == 2)? -1: 1;
* }
* long long int convexHull(Point points[], long long int n)
* {
* long long int answer=0;
* // Find the bottommost polong long int
* long long int ymin = points[0].y, min = 0;
* for (long long int i = 1; i < n; i++)
* {
* long long int y = points[i].y;
* // Pick the bottom-most or chose the left
* // most point in case of tie
* if ((y < ymin) || (ymin == y &&
* points[i].x < points[min].x))
* ymin = points[i].y, min = i;
* }
* // Place the bottom-most point at first position
* swap(points[0], points[min]);

* p0 = points[0];
* [qsort](http://www.opengroup.org/onlinepubs/009695399/functions/qsort.html)(&points[1], n-1, [sizeof](http://www.opengroup.org/onlinepubs/009695399/functions/sizeof.html)(Point), compare);
* long long int m=1;
* for (long long int i=1; i<n; i++)
* {
* while (i < n-1 && orientation(p0, points[i],
* points[i+1]) == 0)
* i++;
* points[m] = points[i];
* m++; // Update size of modified array
* }
* if (m < 3) return -1;
* stack<Point> S;
* S.push(points[0]);
* S.push(points[1]);
* S.push(points[2]);
* for (long long int i = 3; i < m; i++)
* {
* while (orientation(nextToTop(S), S.top(), points[i]) != 2)
* S.pop();
* S.push(points[i]);
* }
* while (!S.empty())
* {
* Point p = S.top();
* answer+=p.price;
* S.pop();
* }
* return answer;
* }

**Union find**

int find(struct subset subsets[], int i)

{

    // find root and make root as parent of i (path compression)

    if (subsets[i].parent != i)

        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;

}

// A function that does union of two sets of x and y

// (uses union by rank)

void Union(struct subset subsets[], int x, int y)

{

    int xroot = find(subsets, x);

    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree

    // (Union by Rank)

    if (subsets[xroot].rank < subsets[yroot].rank)

        subsets[xroot].parent = yroot;

    else if (subsets[xroot].rank > subsets[yroot].rank)

        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment

    // its rank by one

    else

    {

        subsets[yroot].parent = xroot;

        subsets[xroot].rank++;

    }

}

**PRIMALITY**

private static boolean checkPrimality(long p) {

int iteration = 30;

if(p<2)

return false;

if(p != 2 && p%2==0)

return false;

long s = p-1;

while(s%2==0)

s/=2;

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

{

long a = ((long)(Math.random()\*10000000000000000L))%(p-1) +1 , temp = s;

long mod = modPow(a, temp, p);

while(temp!=p-1 && mod!=1 && mod!=p-1){

mod=(mod \* mod)%p;

temp \*= 2;

}

if(mod!=p-1 && temp%2==0){

return false;

}

}

return true;

}

**int lca(int u,int v){**

**if(level[u]>level[v])return lca(v,u);**

**for(int i=19;i>=0 && level[v]!=level[u];i--)**

**{ if(level[v]>=level[u]+(1<<i))**

**v=pa[i][v]; }**

**if(u==v)return u;**

**for(int i=19;i>=0;i--)**

**{ if(pa[i][u]!=pa[i][v])**

**{ u=pa[i][u];v=pa[i][v];**

**}Q}**

**return pa[0][u];**

**}LCA**

**FENWICK TREE---------------------------------------------------**

int read(int idx){

int sum = 0;

while (idx > 0){

sum += tree[idx];

idx -= (idx & -idx);

}

return sum;

}

void update(int idx ,int val){

while (idx <= MaxVal){

tree[idx] += val;

idx += (idx & -idx);

}

}

**2D**

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

while (x <= max\_x){

updatey(x , y , val);

*// this function should update array tree[x]*

x += (x & -x);

}

}

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

while (y <= max\_y){

tree[x][y] += val;

y += (y & -y);

}

}

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

int y1;

while (x <= max\_x){

y1 = y;

while (y1 <= max\_y){

tree[x][y1] += val;

y1 += (y1 & -y1);

}

x += (x & -x);

}

}

void BellmanFord(struct Graph\* graph, int src)

{

    int V = graph->V;

    int E = graph->E;

    int dist[V];

    // Step 1: Initialize distances from src to all other vertices

    // as INFINITE

    for (int i = 0; i < V; i++)

        dist[i]   = INT\_MAX;

    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest

    // path from src to any other vertex can have at-most |V| - 1

    // edges

    for (int i = 1; i <= V-1; i++)

    {

        for (int j = 0; j < E; j++)

        {

            int u = graph->edge[j].src;

            int v = graph->edge[j].dest;

            int weight = graph->edge[j].weight;

            if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

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

        }

    }

    // Step 3: check for negative-weight cycles.  The above step

    // guarantees shortest distances if graph doesn't contain

    // negative weight cycle.  If we get a shorter path, then there

    // is a cycle.

    for (int i = 0; i < E; i++)

    {

        int u = graph->edge[i].src;

        int v = graph->edge[i].dest;

        int weight = graph->edge[i].weight;

        if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

            printf("Graph contains negative weight cycle");

    }

    printArr(dist, V);

    return;

}

--------------**PRIM MST-------------------------------**

void PrimMST(struct Graph\* graph)

{

    int V = graph->V;// Get the number of vertices in graph

    int parent[V];   // Array to store constructed MST

    int key[V];      // Key values used to pick minimum weight edge in cut

    // minHeap represents set E

    struct MinHeap\* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. Key value of

    // all vertices (except 0th vertex) is initially infinite

    for (int v = 1; v < V; ++v)

    {

        parent[v] = -1;

        key[v] = INT\_MAX;

        minHeap->array[v] = newMinHeapNode(v, key[v]);

        minHeap->pos[v] = v;

    }

    // Make key value of 0th vertex as 0 so that it

    // is extracted first

    key[0] = 0;

    minHeap->array[0] = newMinHeapNode(0, key[0]);

    minHeap->pos[0]   = 0;

    // Initially size of min heap is equal to V

    minHeap->size = V;

    // In the followin loop, min heap contains all nodes

    // not yet added to MST.

    while (!isEmpty(minHeap))

    {

        // Extract the vertex with minimum key value

        struct MinHeapNode\* minHeapNode = extractMin(minHeap);

        int u = minHeapNode->v; // Store the extracted vertex number

        // Traverse through all adjacent vertices of u (the extracted

        // vertex) and update their key values

        struct AdjListNode\* pCrawl = graph->array[u].head;

        while (pCrawl != NULL)

        {

            int v = pCrawl->dest;

            // If v is not yet included in MST and weight of u-v is

            // less than key value of v, then update key value and

            // parent of v

            if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v])

            {

                key[v] = pCrawl->weight;

                parent[v] = u;

                decreaseKey(minHeap, v, key[v]);

            }

            pCrawl = pCrawl->next;

        }

    }

    // print edges of MST

    printArr(parent, V);

}

---------------------------------FFT-------------------------------------

static long[] FFT(long L[],int pow2,char c){

L = BitReaarange(L,pow2) ;

int N = L.length ;

if(c=='I')

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

L[i]=opr.D(L[i],N) ;

for(int s=1 ; s<=pow2 ; s++){

int m = 1<<s ;

long wm = (c=='F') ? prt[s] : prti[s] ;

long w=1 ;

int m2 = (m>>1) ;

for(int i=0 ; i<m2 ; i++){

for(int k=i ; k<N ; k+=m){

long t = (w\*L[k+m2])%r ;

long u = L[k] ;

L[k]=(u+t)%r ;

L[k+m2]=(u-t+r)%r ;

}

w = (w\*wm)%r ;

}

}

return L ;

}

static long[] BitReaarange(long L[],int pow2){

int size = L.length ;

long Ans[] = new long[size] ;

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

Ans[BRev(pow2,i)]=L[i] ;

}

return Ans ;

}

static int BRev(int l,int bit){

int ans=0 ;

while(l--!=0){

ans = (ans<<1)+(bit&1) ;

bit>>=1 ;

}

return ans ;

}

------------------------BIPARTITE MATCHING-------------------------------------------

boolean bfs(){

Queue<Integer> Q = new LinkedList<Integer>() ;

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

if(ngU[i]==N){

depth[i]=0 ;

Q.add(i) ;

}else

depth[i]=Integer.MAX\_VALUE ;

}

depth[N]=Integer.MAX\_VALUE ;

while(!Q.isEmpty()){

int node = Q.poll() ;

if(depth[node]<depth[N])

for(int nnode=0 ; nnode<N ; nnode++)

if(edge[node][nnode])

if(depth[ngV[nnode]]==Integer.MAX\_VALUE){

depth[ngV[nnode]]=depth[node]+1 ;

Q.add(ngV[nnode]) ;

}

}

return depth[N]!=Integer.MAX\_VALUE ;

}

boolean dfs(int node){

if(node==N)

return true ;

for(int nnode=0 ; nnode<N ; nnode++)

if(edge[node][nnode] && depth[ngV[nnode]]==depth[node]+1 && dfs(ngV[nnode])){

ngU[node]=nnode ;

ngV[nnode]=node ;

return true ;

}

depth[node]=Integer.MAX\_VALUE ;

return false ;

}

ArtPt(v) {

color[v] = gray;

// v initially can only climb up to itself

Low[v] = d[v] = ++time;

for all w in Adj(v) do {

if (color[w] == white) {

pred[w] = v;

ArtPt(w);

// When ArtPt(w) is completed, Low[w] stores the

// lowest value it can climb up for a subtree

// rooted at w.

// Recall v is the parent of w.

if (pred [v] == NULL) {

// v has no predecessor , so v is the root.

// apply observation 1.

if ('w' is v's second child) output v;

}

else if (Low[w] >= d[v]) output v;

// subtree rooted at w can't climb higher than v

// apply observation 3.

// update Low[v] if a children subtree can

// climb higher

Low[v] = min(Low[v], Low[w]);

}

else if (w != pred[v]) { // (v, w) is a back edge

// update Low[v] if a back edge climbs higher

Low[v] = min(Low[v], d[w]);

}

}

color[v] = black;

}

Recall that DFS is a recursive algorithm, we make use

of a *stack* to trace back the recursive calls. When we

process an edge \_\_ \_\_􀀀 (either by a recursive call on

vertex 􀀀 from vertex \_ , or \_\_ \_\_􀀀 is back edge), we

push that edge to a stack. Later, if we identify \_ as an

articulation point (where the subtree rooted at 􀀀 can’t

climb higher than \_ ), then all the edges from the top

of the stack down to \_\_ \_\_􀀀 are the edges of one biconnected

component. (Observe how a stack is used

to trace the recursive calls). So we pop edges out of

the stack until \_\_ \_\_􀀀 (also pop \_\_ \_\_􀀀 ), those edges

belong to a biconnected component

void Graph::bridgeUtil(int u, bool visited[], int disc[],

                                  int low[], int parent[])

{

    // A static variable is used for simplicity, we can

    // avoid use of static variable by passing a pointer.

    static int time = 0;

    // Mark the current node as visited

    visited[u] = true;

    // Initialize discovery time and low value

    disc[u] = low[u] = ++time;

    // Go through all vertices aadjacent to this

    list<int>::iterator i;

    for (i = adj[u].begin(); i != adj[u].end(); ++i)

    {

        int v = \*i;  // v is current adjacent of u

        // If v is not visited yet, then recur for it

        if (!visited[v])

        {

            parent[v] = u;

            bridgeUtil(v, visited, disc, low, parent);

            // Check if the subtree rooted with v has a

            // connection to one of the ancestors of u

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

            // If the lowest vertex reachable from subtree

            // under v is  below u in DFS tree, then u-v

            // is a bridge

            if (low[v] > disc[u])

              cout << u <<" " << v << endl;

        }

        // Update low value of u for parent function calls.

        else if (v != parent[u])

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

    }

}

// DFS based function to find all bridges. It uses recursive

// function bridgeUtil()

void Graph::bridge()

{

    // Mark all the vertices as not visited

    bool \*visited = new bool[V];

    int \*disc = new int[V];

    int \*low = new int[V];

    int \*parent = new int[V];

    // Initialize parent and visited arrays

    for (int i = 0; i < V; i++)

    {

        parent[i] = NIL;

        visited[i] = false;

    }

    // Call the recursive helper function to find Bridges

    // in DFS tree rooted with vertex 'i'

    for (int i = 0; i < V; i++)

        if (visited[i] == false)

            bridgeUtil(i, visited, disc, low, parent);

}

SEGMENTS-INTERSECT.p1; p2; p3; p4/

1 d1 D DIRECTION.p3; p4; p1/

2 d2 D DIRECTION.p3; p4; p2/

3 d3 D DIRECTION.p1; p2; p3/

4 d4 D DIRECTION.p1; p2; p4/

5 **if** ..d1 > 0 and d2 < 0/ or .d1 < 0 and d2 > 0// and

..d3 > 0 and d4 < 0/ or .d3 < 0 and d4 > 0//

6 **return** TRUE

7 **elseif** d1 == 0 and ON-SEGMENT.p3; p4; p1/

8 **return** TRUE

9 **elseif** d2 == 0 and ON-SEGMENT.p3; p4; p2/

10 **return** TRUE

11 **elseif** d3 == 0 and ON-SEGMENT.p1; p2; p3/

12 **return** TRUE

13 **elseif** d4 == 0 and ON-SEGMENT.p1; p2; p4/

14 **return** TRUE

15 **else return** FALSE

DIRECTION.pi; pj; pk/

1 **return** .pk \_ pi / .pj \_ pi /

ON-SEGMENT.pi; pj; pk/

1 **if** min.xi; xj / \_ xk \_ max.xi; xj / and min.yi; yj / \_ yk \_ max.yi; yj /

2 **return** TRUE

3 **else return** FALSE

int chainNo=0,chainHead[N],chainPos[N],chainInd[N],chainSize[N];

void hld(int cur) {

if(chainHead[chainNo] == -1) chainHead[chainNo]=cur;

chainInd[cur] = chainNo;

chainPos[cur] = chainSize[chainNo];

chainSize[chainNo]++;

int ind = -1,mai = -1;

for(int i = 0; i < adj[cur].sz; i++) { if(subsize[ adj[cur][i] ] > mai) {

mai = subsize[ adj[cur][i] ];

ind = i;

}

}

if(ind >= 0) hld( adj[cur][ind] );

for(int i = 0; i < adj[cur].sz; i++) {

if(i != ind) {

chainNo++;

hld( adj[cur][i] );

}

}

}

add(position):

count[array[position]]++

if count[array[position]] == 3:

answer++

remove(position):

count[array[position]]--

if count[array[position]] == 2:

answer--

currentL = 0

currentR = 0

answer = 0

count[] = 0

for each query:

// currentL should go to L, currentR should go to R

while currentL &amp;lt; L:

remove(currentL)

currentL++

while currentL &amp;gt; L:

add(currentL)

currentL--

while currentR &amp;lt; R:

add(currentR)

currentR++

while currentR &amp;gt; R:

remove(currentR)

currentR--

output answer

Initially we always looped from L to R, but now we are changing the positions from previous query to adjust to current query.  
If previous query was L=3, R=10, then we will have currentL=3 and currentR=10 by the end of that query. Now if the next query is L=5, R=7, then we move the currentL to 5 and currentR to 7.  
add function means we are adding the element at position to our current set. And updating answer accordingly.  
remove function means we are deleting the element at position from our current set. And updating answer accordingly.  
**Edit**: Have a look a Shubajit Saha’s comment. And also [this](http://codeforces.com/blog/entry/46399?#comment-308528).

**Explain an algorithm to solve above problem and state its correctness**

MO’s algorithm is just an order in which we process the queries. We were given M queries, we will re-order the queries in a particular order and then process them. Clearly, this is an offline algorithm. Each query has L and R, we will call them opening and closing. Let us divide the given input array into Sqrt(N) blocks. Each block will be N / Sqrt(N) = Sqrt(N) size. Each opening has to fall in one of these blocks. Each closing has to fall in one of these blocks.

A query belongs to P’th block if the opening of that query fall in P’th block. In this algorithm we will process the queries of 1st block. Then we process the queries of 2nd block and so on.. finally Sqrt(N)’th block. We already have an ordering, queries are ordered in the ascending order of its block. There can be many queries that belong to the same block.

From now, I will ignore about all the blocks and only focus on how we query and answer block 1. We will similarly do for all blocks. All of these queries have their opening in block 1, but their closing can be in any block including block 1. Now let us reorder these queries in ascending order of their R value. We do this for all the blocks.

How does the final order look like?  
All the queries are first ordered in ascending order of their block number (block number is the block in which its opening falls). Ties are ordered in ascending order of their R value.  
For example consider following queries and assume we have 3 blocks each of size 3.  
{0, 3} {1, 7} {2, 8} {7, 8} {4, 8} {4, 4} {1, 2}  
Let us re-order them based on their block number.  
{0, 3} {1, 7} {2, 8} {1, 2} {4, 8} {4, 4} {7, 8}  
Now let us re-order ties based on their R value.  
{1, 2} {0, 3} {1, 7} {2, 8} {4, 4} {4, 8} {7, 8}

Now we use the same code stated in previous section and solve the problem. Above algorithm is correct as we did not do any changes but just reordered the queries.

**Proof for complexity of above algorithm – O(Sqrt(N) \* N)**

We are done with MO’s algorithm, it is just an ordering. Awesome part is its runtime analysis. It turns out that the O(N^2) code we wrote works in O(Sqrt(N) \* N) time if we follow the order i specified above. Thats awesome right, with just reordering the queries we reduced the complexity from O(N^2) to O(Sqrt(N) \* N), and that too with out any further modification to code. Hurray! we will get AC with O(Sqrt(N) \* N).

Have a look at our code above, the complexity over all queries is determined by the 4 while loops. First 2 while loops can be stated as “Amount moved by left pointer in total”, second 2 while loops can be stated as “Amount moved by right pointer”. Sum of these two will be the over all complexity.

Most important. Let us talk about the right pointer first. For each block, the queries are sorted in increasing order, so clearly the right pointer (currentR) moves in increasing order. During the start of next block the pointer possibly at extreme end will move to least R in next block. That means for a given block, the amount moved by right pointer is O(N). We have O(Sqrt(N)) blocks, so the total is O(N \* Sqrt(N)). Great!

Let us see how the left pointer moves. For each block, the left pointer of all the queries fall in the same block, as we move from query to query the left pointer might move but as previous L and current L fall in the same block, the moment is O(Sqrt(N)) (Size of the block). In each block the amount left pointer movies is O(Q \* Sqrt(N)) where Q is number of queries falling in that block. Total complexity is O(M \* Sqrt(N)) for all blocks.

There you go, total complexity is O( (N + M) \* Sqrt(N)) = O( N \* Sqrt(N))

|  |
| --- |
| #include<cstdio> |
|  | #include <map> |
|  | #include <vector> |
|  | #include <cstring> |
|  | using namespace std; |
|  |  |
|  | #define sz size() |
|  | #define pb push\_back |
|  | #define rep(i,n) for(int i=0;i<n;i++) |
|  | #define fd(i,a,b) for(int i=a; i>=b; i--) |
|  |  |
|  | #define N 111111 |
|  | #define LN 19 |
|  | int v[N], pa[N][LN], RM[N], depth[N], maxi = 0; |
|  | vector <int> adj[N]; |
|  | map <int, int> M; |
|  |  |
|  | struct node |
|  | { |
|  | int count; |
|  | node \*left, \*right; |
|  |  |
|  | node(int count, node \*left, node \*right): |
|  | count(count), left(left), right(right) {} |
|  |  |
|  | node\* insert(int l, int r, int w); |
|  | }; |
|  |  |
|  | node \*null = new node(0, NULL, NULL); //see line 135 |
|  |  |
|  | node \* node::insert(int l, int r, int w) |
|  | { |
|  | if(l <= w && w < r) |
|  | { |
|  | // With in the range, we need a new node |
|  | if(l+1 == r) |
|  | { |
|  | return new node(this->count+1, null, null); |
|  | } |
|  |  |
|  | int m = (l+r)>>1; |
|  |  |
|  | return new node(this->count+1, this->left->insert(l, m, w), this->right->insert(m, r, w)); |
|  | } |
|  |  |
|  | // Out of range, we can use previous tree node. |
|  | return this; |
|  | } |
|  |  |
|  | node \*root[N]; |
|  | void dfs(int cur, int prev) |
|  | { |
|  | pa[cur][0] = prev; |
|  | depth[cur] = (prev == -1 ? 0 : depth[prev] + 1); |
|  |  |
|  | // Construct the segment tree for this node using parent segment tree |
|  | // This is the formula we derived |
|  | root[cur] = ( prev == -1 ? null : root[prev] )->insert( 0, maxi, M[v[cur]] ); |
|  |  |
|  | rep(i, adj[cur].sz) |
|  | if(adj[cur][i] != prev) |
|  | dfs(adj[cur][i], cur); |
|  | } |
|  |  |
|  | int LCA(int u, int v) |
|  | { |
|  | if(depth[u] < depth[v]) |
|  | return LCA(v, u); |
|  |  |
|  | int diff = depth[u] - depth[v]; |
|  |  |
|  | rep(i, LN) |
|  | if((diff>>i) & 1) |
|  | u = pa[u][i]; |
|  |  |
|  | if(u != v) |
|  | { |
|  | fd(i, LN-1, 0) |
|  | if(pa[u][i] != pa[v][i]) |
|  | { |
|  | u = pa[u][i]; |
|  | v = pa[v][i]; |
|  | } |
|  | u = pa[u][0]; |
|  | } |
|  |  |
|  | return u; |
|  | } |
|  |  |
|  | int query(node \*a, node \*b, node \*c, node \*d, int l, int r, int k) |
|  | { |
|  | if(l+1 == r) |
|  | { |
|  | return l; |
|  | } |
|  |  |
|  | // This is the formula we derived |
|  | int count = a->left->count + b->left->count - c->left->count - d->left->count; |
|  | int m = (l+r)>>1; |
|  |  |
|  | // We have enough on left, so go left |
|  | if(count >= k) |
|  | return query(a->left, b->left, c->left, d->left, l, m, k); |
|  |  |
|  | // We do not have enough on left, go right, remove left elements count |
|  | return query(a->right, b->right, c->right, d->right, m, r, k - count); |
|  | } |
|  |  |
|  | int main() |
|  | { |
|  | int n, m; |
|  |  |
|  | scanf("%d%d", &n, &m); |
|  |  |
|  | rep(i, n) |
|  | { |
|  | scanf("%d", &v[i]); |
|  | M[v[i]]; |
|  | } |
|  |  |
|  | maxi = 0; |
|  | for( map <int, int > :: iterator it = M.begin(); it != M.end(); it++ ) |
|  | { |
|  | M[it->first] = maxi; |
|  | RM[maxi] = it->first; |
|  | maxi++; |
|  | } |
|  |  |
|  | // We compressed the given weights into the range [0..n) |
|  |  |
|  | rep(i, n-1) |
|  | { |
|  | int u, v; |
|  | scanf("%d%d", &u, &v); |
|  | u--; v--; |
|  | adj[u].pb(v); |
|  | adj[v].pb(u); |
|  | } |
|  |  |
|  | // Root the tree at some node. |
|  | memset(pa, -1, sizeof pa); |
|  | null->left = null->right = null; |
|  | dfs(0, -1); |
|  |  |
|  | // Build jump table for LCA in O( log N ) |
|  | rep(i, LN-1) |
|  | rep(j, n) |
|  | if(pa[j][i] != -1) |
|  | pa[j][i+1] = pa[pa[j][i]][i]; |
|  |  |
|  | while(m--) |
|  | { |
|  | int u, v, k; |
|  | scanf("%d%d%d", &u, &v, &k); |
|  | u--; v--; |
|  |  |
|  | int lca = LCA(u, v); |
|  | // Four nodes we spoke about are u, v, lca, parent(lca) |
|  | int ans = query(root[u], root[v], root[lca], (pa[lca][0] == -1 ? null : root[ pa[lca][0] ]), 0, maxi, k); |
|  |  |
|  | // Reverse Map the values, that is, uncompress |
|  | printf("%d\n", RM[ans]); |
|  | } |
|  | } |

-----------------findCentroid---------------------------  
int FindCentroid(int root){

//long st = System.currentTimeMillis() ;

Stack<Integer> S = new Stack<Integer>() ;

S.push(root) ;

par[root]=root ;

while(!S.isEmpty()){

int node = S.peek() ;

if(vis[node]){

S.pop() ;

szsub[par[node]]+=szsub[node] ;

vis[node]=false ;

}else{

for(int child : adj[node])

if(!vis[child]){

S.push(child) ;

par[child]=node ;

}

szsub[node]=1 ;

vis[node]=true ;

}

}

szsub[root]/=2 ;

int n = szsub[root] ;

int node=root ;

boolean found = false ;

while(!found){

boolean flag=true ;

for(int child : adj[node])

if(szsub[child]>n/2 && child!=par[node]){

node=child ;

flag=false ;

break ;

}

found=flag ;

}

Given a string *S* of length *n*, the Z Algorithm produces an array *Z* where *Z*[*i*] is the length of the longest substring starting from *S*[*i*]which is also a prefix of *S*, i.e. the maximum *k* such that *S*[*j*] = *S*[*i* + *j*] for all 0 ≤ *j* < *k*. Note that *Z*[*i*] = 0 means that *S*[0] ≠ *S*[*i*]. For easier terminology, we will refer to substrings which are also a prefix as prefix-substrings.

The algorithm relies on a single, crucial invariant. As we iterate over the letters in the string (index *i* from 1 to *n* - 1), we maintain an interval [*L*, *R*] which is the interval with maximum *R* such that 1 ≤ *L* ≤ *i* ≤ *R* and *S*[*L*...*R*] is a prefix-substring (if no such interval exists, just let *L* = *R* =  - 1). For *i* = 1, we can simply compute *L* and *R* by comparing *S*[0...] to *S*[1...]. Moreover, we also get *Z*[1] during this.

Now suppose we have the correct interval [*L*, *R*] for *i* - 1 and all of the *Z* values up to *i* - 1. We will compute *Z*[*i*] and the new [*L*, *R*] by the following steps:

* If *i* > *R*, then there does not exist a prefix-substring of *S* that starts before *i* and ends at or after *i*. If such a substring existed, [*L*, *R*]would have been the interval for that substring rather than its current value. Thus we "reset" and compute a new [*L*, *R*] by comparing *S*[0...] to *S*[*i*...] and get *Z*[*i*] at the same time (*Z*[*i*] = *R* - *L* + 1).
* Otherwise, *i* ≤ *R*, so the current [*L*, *R*] extends at least to *i*. Let *k* = *i* - *L*. We know that *Z*[*i*] ≥ *min*(*Z*[*k*], *R* - *i* + 1) because *S*[*i*...]matches *S*[*k*...] for at least *R* - *i* + 1 characters (they are in the [*L*, *R*] interval which we know to be a prefix-substring). Now we have a few more cases to consider.
* If *Z*[*k*] < *R* - *i* + 1, then there is no longer prefix-substring starting at *S*[*i*] (or else *Z*[*k*] would be larger), meaning *Z*[*i*] = *Z*[*k*] and [*L*, *R*] stays the same. The latter is true because [*L*, *R*] only changes if there is a prefix-substring starting at *S*[*i*] that extends beyond *R*, which we know is not the case here.
* If *Z*[*k*] ≥ *R* - *i* + 1, then it is possible for *S*[*i*...] to match *S*[0...] for more than *R* - *i* + 1 characters (i.e. past position *R*). Thus we need to update [*L*, *R*] by setting *L* = *i* and matching from *S*[*R* + 1] forward to obtain the new *R*. Again, we get *Z*[*i*] during this.

The process computes all of the *Z* values in a single pass over the string, so we're done. Correctness is inherent in the algorithm and is pretty intuitively clear.

Analysis

We claim that the algorithm runs in *O*(*n*) time, and the argument is straightforward. We never compare characters at positions less than *R*, and every time we match a character *R* increases by one, so there are at most *n* comparisons there. Lastly, we can only mismatch once for each *i* (it causes *R* to stop increasing), so that's another at most *n* comparisons, giving *O*(*n*) total.

Code

Simple and short. Note that the optimization *L* = *R* = *i* is used when *S*[0] ≠ *S*[*i*] (it doesn't affect the algorithm since at the next iteration *i* > *R* regardless).

int L = 0, R = 0;

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

if (i > R) {

L = R = i;

while (R < n && s[R-L] == s[R]) R++;

z[i] = R-L; R--;

} else {

int k = i-L;

if (z[k] < R-i+1) z[i] = z[k];

else {

L = i;

while (R < n && s[R-L] == s[R]) R++;

z[i] = R-L; R--;

}

}

}

Application

One application of the Z Algorithm is for the standard string matching problem of finding matches for a pattern *T* of length *m* in a string *S* of length *n*. We can do this in *O*(*n* + *m*) time by using the Z Algorithm on the string *T* Φ *S* (that is, concatenating *T*, Φ, and *S*) where Φ is a character that matches nothing. The indices *i* with *Z*[*i*] = *m* correspond to matches of *T* in *S*.

Lastly, to solve Problem B of Beta Round 93, we simply compute *Z* for the given string *S*, then iterate from *i* to *n* - 1. If *Z*[*i*] = *n* - *i* then we know the suffix from *S*[*i*] is a prefix, and if the largest *Z* value we've seen so far is at least *n* - *i*, then we know some string inside also matches that prefix. That gives the result.

int maxz = 0, res = 0;

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

if (z[i] == n-i && maxz >= n-i) { res = n-i; break; }

maxz = max(maxz, z[i]);

}

**using** **namespace** std;

#include <bits/stdc++.h>

// Max number of states in the matching machine.

// Should be equal to the sum of the length of all keywords.

**const** **int** MAXS = 500;

// Maximum number of characters in input alphabet

**const** **int** MAXC = 26;

// OUTPUT FUNCTION IS IMPLEMENTED USING out[]

// Bit i in this mask is one if the word with index i

// appears when the machine enters this state.

**int** out[MAXS];

// FAILURE FUNCTION IS IMPLEMENTED USING f[]

**int** f[MAXS];

// GOTO FUNCTION (OR TRIE) IS IMPLEMENTED USING g[][]

**int** g[MAXS][MAXC];

// Builds the string matching machine.

// arr -   array of words. The index of each keyword is important:

//         "out[state] & (1 << i)" is > 0 if we just found word[i]

//         in the text.

// Returns the number of states that the built machine has.

// States are numbered 0 up to the return value - 1, inclusive.

**int** buildMatchingMachine(string arr[], **int** k)

{

    // Initialize all values in output function as 0.

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

    // Initialize all values in goto function as -1.

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

    // Initially, we just have the 0 state

**int** states = 1;

    // Construct values for goto function, i.e., fill g[][]

    // This is same as building a Trie for arr[]

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

    {

**const** string &word = arr[i];

**int** currentState = 0;

        // Insert all characters of current word in arr[]

**for** (**int** j = 0; j < word.size(); ++j)

        {

**int** ch = word[j] - 'a';

            // Allocate a new node (create a new state) if a

            // node for ch doesn't exist.

**if** (g[currentState][ch] == -1)

                g[currentState][ch] = states++;

            currentState = g[currentState][ch];

        }

        // Add current word in output function

        out[currentState] |= (1 << i);

    }

    // For all characters which don't have an edge from

    // root (or state 0) in Trie, add a goto edge to state

    // 0 itself

**for** (**int** ch = 0; ch < MAXC; ++ch)

**if** (g[0][ch] == -1)

            g[0][ch] = 0;

    // Now, let's build the failure function

    // Initialize values in fail function

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

    // Failure function is computed in breadth first order

    // using a queue

    queue<**int**> q;

     // Iterate over every possible input

**for** (**int** ch = 0; ch < MAXC; ++ch)

    {

        // All nodes of depth 1 have failure function value

        // as 0. For example, in above diagram we move to 0

        // from states 1 and 3.

**if** (g[0][ch] != 0)

        {

            f[g[0][ch]] = 0;

            q.push(g[0][ch]);

        }

    }

    // Now queue has states 1 and 3

**while** (q.size())

    {

        // Remove the front state from queue

**int** state = q.front();

        q.pop();

        // For the removed state, find failure function for

        // all those characters for which goto function is

        // not defined.

**for** (**int** ch = 0; ch <= MAXC; ++ch)

        {

            // If goto function is defined for character 'ch'

            // and 'state'

**if** (g[state][ch] != -1)

            {

                // Find failure state of removed state

**int** failure = f[state];

                // Find the deepest node labeled by proper

                // suffix of string from root to current

                // state.

**while** (g[failure][ch] == -1)

                      failure = f[failure];

                failure = g[failure][ch];

                f[g[state][ch]] = failure;

                // Merge output values

                out[g[state][ch]] |= out[failure];

                // Insert the next level node (of Trie) in Queue

                q.push(g[state][ch]);

            }

        }

    }

**return** states;

}

// Returns the next state the machine will transition to using goto

// and failure functions.

// currentState - The current state of the machine. Must be between

//                0 and the number of states - 1, inclusive.

// nextInput - The next character that enters into the machine.

**int** findNextState(**int** currentState, **char** nextInput)

{

**int** answer = currentState;

**int** ch = nextInput - 'a';

    // If goto is not defined, use failure function

**while** (g[answer][ch] == -1)

        answer = f[answer];

**return** g[answer][ch];

}

// This function finds all occurrences of all array words

// in text.

**void** searchWords(string arr[], **int** k, string text)

{

    // Preprocess patterns.

    // Build machine with goto, failure and output functions

    buildMatchingMachine(arr, k);

    // Initialize current state

**int** currentState = 0;

    // Traverse the text through the nuilt machine to find

    // all occurrences of words in arr[]

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

    {

        currentState = findNextState(currentState, text[i]);

        // If match not found, move to next state

**if** (out[currentState] == 0)

**continue**;

        // Match found, print all matching words of arr[]

        // using output function.

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

        {

**if** (out[currentState] & (1 << j))

            {

                cout << "Word " << arr[j] << " appears from "

                     << i - arr[j].size() + 1 << " to " << i << endl;

            }

        }

    }

}

// Driver program to test above

**int** main()

{

    string arr[] = {"he", "she", "hers", "his"};

    string text = "ahishers";

**int** k = **sizeof**(arr)/**sizeof**(arr[0]);

    searchWords(arr, k, text);

**return** 0;

}

struct Point

{

    int x;

    int y;

};

// Given three colinear points p, q, r, the function checks if

// point q lies on line segment 'pr'

bool onSegment(Point p, Point q, Point r)

{

    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&

        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))

       return true;

    return false;

}

// To find orientation of ordered triplet (p, q, r).

// The function returns following values

// 0 --> p, q and r are colinear

// 1 --> Clockwise

// 2 --> Counterclockwise

int orientation(Point p, Point q, Point r)

{

    // See <http://www.geeksforgeeks.org/orientation-3-ordered-points/>

    // for details of below formula.

    int val = (q.y - p.y) \* (r.x - q.x) -

              (q.x - p.x) \* (r.y - q.y);

    if (val == 0) return 0;  // colinear

    return (val > 0)? 1: 2; // clock or counterclock wise

}

// The main function that returns true if line segment 'p1q1'

// and 'p2q2' intersect.

bool doIntersect(Point p1, Point q1, Point p2, Point q2)

{

    // Find the four orientations needed for general and

    // special cases

    int o1 = orientation(p1, q1, p2);

    int o2 = orientation(p1, q1, q2);

    int o3 = orientation(p2, q2, p1);

    int o4 = orientation(p2, q2, q1);

    // General case

    if (o1 != o2 && o3 != o4)

        return true;

    // Special Cases

    // p1, q1 and p2 are colinear and p2 lies on segment p1q1

    if (o1 == 0 && onSegment(p1, p2, q1)) return true;

    // p1, q1 and p2 are colinear and q2 lies on segment p1q1

    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

    // p2, q2 and p1 are colinear and p1 lies on segment p2q2

    if (o3 == 0 && onSegment(p2, p1, q2)) return true;

     // p2, q2 and q1 are colinear and q1 lies on segment p2q2

    if (o4 == 0 && onSegment(p2, q1, q2)) return true;

    return false; // Doesn't fall in any of the above cases

}

// Driver program to test above functions

int main()

{

    struct Point p1 = {1, 1}, q1 = {10, 1};

    struct Point p2 = {1, 2}, q2 = {10, 2};

    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

    p1 = {10, 0}, q1 = {0, 10};

    p2 = {0, 0}, q2 = {10, 10};

    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

    p1 = {-5, -5}, q1 = {0, 0};

    p2 = {1, 1}, q2 = {10, 10};

    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

    return 0;

}