**C++ Standard Template Library**

***C++ STL Data Structures***

**Vector** *<vector>***:**

vector**<**T**>** V // O(1)

V**.**push\_back**(**T**)** // O(1)

V**.**insert**(**i**,** T**)** // O(n)

V**.**size**()** // O(1)

V**.**clear**()** // O(n)

V**[**i**]** // O(1)

**Stack** *<stack>***:**

stack**<**T**>** S // O(1)

S**.**push**(**T**)** // O(1)

S**.**pop**()** // O(1)

S**.**size**()** // O(1)

S**.**top**()** // O(1)

**Queue** *<queue>***:**

queue**<**T**>** Q // O(1)

Q**.**push**(**T**)** // O(1)

Q**.**pop**()** // O(1)

Q**.**size**()** // O(1)

Q**.**front**()** // O(1)

**Double Ended Queue** *<deque>***:**

deque**<**T**>** D // O(1)

D**.**push\_back**(**T**)** // O(1)

D**.**push\_front**(**T**)** // O(1)

D**.**pop\_back**()** // O(1)

D**.**pop\_front**()** // O(1)

D**.**size**()** // O(1)

D**.**back**()** // O(1)

D**.**front**()** // O(1)

D**.**clear**()** // O(n)

**Set** *<set>***:**

set**<**T**>** S // O(1)

S**.**insert**(**T**)** // O(log n)

S**.**erase**(**T**)** // O(log n)

S**.**count**(**T**)** // O(log n)

S**.**clear**()** // O(n)

**Map** *<map>***:**

map**<**K**,** V**>** M // O(1)

M**[**K**]** **=** V // O(log n)

M**.**count**(**K**)** // O(log n)

M**.**clear**()** // O(n)

**Priority Queue** *<queue>***:**

priority\_queue**<**T**>** PQ // O(1)

PQ**.**push**(**T**)** // O(log n)

PQ**.**pop**()** // O(log n)

PQ**.**top**()** // O(1)

PQ**.**empty**()** // O(1)

**Bit Set** *<bitset>***:**

bitset**<**n**>** B // O(1)

B**[**i**]** **=** 0 **|** 1 // O(1)

**Pair** *<utility>***:**

pair**<**T**,** T**>** P // O(1)

P**.**first **|** P**.**second // O(1)

make\_pair**(**T**,** T**);** // O(1)

**Override Comparison Operator:**

struct T**{** // Data

bool **operator** **<(**T cmp**)** const**{**

// Comparison

**}**

**};**

***C++ STL Algorithms***

**Sort:**

sort**(**array**,** array **+** n**)** // O(n log n)

sort**(**vector**.**begin**(),** vector**.**end**())** // O(n log n)

**Reverse:**

reverse**(**array**,** array **+** n**)** // O(n)

reverse**(**vector**.**begin**(),** vector**.**end**())** // O(n)

**Fill:**

fill**(**array**,** array **+** n**,** T**);** // O(n)

fill**(**vector**.**begin**(),** vector**.**end**(),** T**);** // O(n)

**Lower / Upper Bound:**

lower\_bound**(**array**,** array **+** n**,** T**)** // O(log n)

lower\_bound**(**vector**.**begin**(),** vector**.**end**(),** T**)**

upper\_bound**(**array**,** array **+** n**,** T**)** // O(log n)

upper\_bound**(**vector**.**begin**(),** vector**.**end**(),** T**)**

set**.**lower\_bound**(**T**)** // O(log n)

set**.**upper\_bound**(**T**)** // O(log n)

**Next / Previous Permutation:**

next\_permutation**(**array**,** array **+** n**)** // O(n)

prev\_permutation**(**array**,** array **+** n**)** // O(n)

**Minimum / Maximum:**

min**(**T**,** T**)** // O(1)

max**(**T**,** T**)** // O(1)

**Data Structures**

***Union–Find Disjoint Sets***

// NOTE: If there are v vertexes, their

// indexes are numbered from 0 to v - 1.

struct DisjointSet**{**

int sets**;**

int**\*** parent**;**

int**\*** members**;**

void init**(**int n**){** // O(v)

sets **=** n**;**

parent **=** **new** int**[**n**];**

members **=** **new** int**[**n**];**

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

parent**[**i**]** **=** i**,** members**[**i**]** **=** 1**;**

**}**

int find**(**int a**){** // O(1)

**if(**a **==** parent**[**a**])** **return** a**;**

**return** parent**[**a**]** **=** find**(**parent**[**a**]);**

**}**

int setsize**(**int a**){** // O(1)

**return** members**[**find**(**a**)];**

**}**

bool joined**(**int a**,** int b**){** // O(1)

**return** find**(**a**)** **==** find**(**b**);**

**}**

void join**(**int a**,** int b**){** // O(1)

**if(**joined**(**a**,** b**))** **return;**

members**[**find**(**b**)]** **+=** members**[**find**(**a**)];**

parent**[**find**(**a**)]** **=** find**(**b**);**

sets**--;**

**}**

**};**

***Segment Tree***

// NOTE: Tree indexed from 0 to n - 1.

// If you want lazy propagation on tree

// uncomment code tagged as 'Propagation'.

// IMPORTANT: Functions are very generic,

// the way of handling values may change.

#define M ((l + r) / 2)

#define Left (node \* 2)

#define Right (node \* 2 + 1)

struct SegmentTree**{**

int**\*** tree**;**

int**\*** lazy**;**

/\* Propagation

void propagate(int node, int l, int r){ // O(1)

if(l == r) return;

lazy[Left] = lazy[node];

lazy[Right] = lazy[node];

} \*/

int create**(**int node**,** int l**,** int r**){** // O(n log n)

lazy**[**node**]** **=** 0**;**

**if(**l **==** r**)** **return** tree**[**node**]** **=** 0**;**

int L **=** create**(**Left**,** l**,** M**);**

int R **=** create**(**Right**,** M **+** 1**,** r**);**

**return** tree**[**node**]** **=** L **+** R**;**

**}**

int update**(**int node**,** int l**,** int r**,** int a**,** int set**){**

/\* Propagation

if(lazy[node] != 0){

propagate(node, l, r);

tree[node] = lazy[node];

lazy[node] = 0;

} \*/

**if(**a **<** l **||** r **<** a**)**

**return** tree**[**node**];**

**if(**a **<=** l **&&** r **<=** a**){**

/\* Propagation

lazy[node] = set;

propagate(node, l, r);

lazy[node] = 0; \*/

**return** tree**[**node**]** **=** set**;**

**}**

int L **=** update**(**Left**,** l**,** M**,** a**,** set**);**

int R **=** update**(**Right**,** M **+** 1**,** r**,** a**,** set**);**

**return** tree**[**node**]** **=** L **+** R**;**

**}** // O(log n)

int query**(**int node**,** int l**,** int r**,** int a**,** int b**){**

**if(**b **<** l **||** r **<** a**)**

**return** 0**;**

/\* Propagation

if(lazy[node] != 0){

propagate(node, l, r);

tree[node] = lazy[node];

lazy[node] = 0;

}\*/

**if(**a **<=** l **&&** r **<=** b**)**

**return** tree**[**node**];**

int L **=** query**(**Left**,** l**,** M**,** a**,** b**);**

int R **=** query**(**Right**,** M **+** 1**,** r**,** a**,** b**);**

**return** L **+** R**;**

**}** // O(log n)

void init**(**int n**){** // O(n log n)

tree **=** **new** int**[**n **\*** 4**];**

lazy **=** **new** int**[**n **\*** 4**];**

create**(**1**,** 0**,** n **-** 1**);**

**}**

**};**

***Binary Indexed Tree***

// NOTE: Tree indexed from 1 to n.

struct BIT**{**

int size**;**

int**\*** tree**;**

void init**(**int n**){** // O(1)

tree **=** **new** int**[**n **+** 1**];**

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

tree**[**i**]** **=** 0**;**

size **=** n**;**

**}**

void update**(**int idx**,** int value**){** // O(log n)

**while(**idx **<=** size**){**

tree**[**idx**]** **+=** value**;**

idx **+=** **(**idx **&** **-** idx**);**

**}**

**}**

int query**(**int idx**){** // O(log n)

int sum **=** 0**;**

**while(**idx **>** 0**){**

sum **+=** tree**[**idx**];**

idx **-=** **(**idx **&** **-** idx**);**

**}**

**return** sum**;**

**}**

int range**(**int a**,** int b**){** // O(log n)

**return** query**(**b**)** **-** query**(**a **-** 1**);**

**}**

**};**

***Sparse Table***

// NOTE: Array indexed from 0 to n - 1.

#define LOGN 20

#define MAXN 100000

int num**[**MAXN**];**

int logs**[**MAXN**];**

int sparse**[**MAXN**][**LOGN**];**

void precalc\_sparse**(**int n**){** // O(n log n)

**for(**int i **=** 2**;** i **<** MAXN**;** i**++)**

logs**[**i**]** **=** logs**[**i **/** 2**]** **+** 1**;**

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

sparse**[**i**][**0**]** **=** num**[**i**];**

**for(**int i **=** 0**;** i **<** logs**[**n**];** i**++){**

**for(**int j **=** 0**;** j **<** n**;** j**++){**

int k **=** min**(**j **+** **(**1 **<<** i**),** n **-** 1**);**

int join **=** max**(**sparse**[**j**][**i**],** sparse**[**k**][**i**]);**

sparse**[**j**][**i **+** 1**]** **=** join**;**

**}**

**}**

**}**

int query**(**int a**,** int b**){** // O(1)

**if(**a **==** b**)** **return** sparse**[**a**][**0**];**

int binlog **=** logs**[**b **-** a **+** 1**];**

int k **=** b **-** **(**1 **<<** binlog**)** **+** 1**;**

**return** max**(**sparse**[**a**][**binlog**],** sparse**[**k**][**binlog**]);**

**}**

**Graph Algorithms**

***Topological Sort***

// NOTE: Vertexes indexed from 0 to v - 1.

// SCC algorithm below implicitly provides

// topological sort of the contracted DAG.

#include <vector>

#include <algorithm>

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

#define MAXV 100000

int sortedVertexes**[**MAXV**];** // Result

int vertexNumber**;**

bool visited**[**MAXV**];**

vector**<**int**>** E**[**MAXV**];**

void TopologicalTraverse**(**int u**){** // O(v + e)

visited**[**u**]** **=** **true;**

vector**<**int**>::**iterator v**;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++)**

**if(!**visited**[\***v**])** TopologicalTraverse**(\***v**);**

sortedVertexes**[**vertexNumber**++]** **=** u**;**

**}**

void TopologicalSort**(**int v**){** // O(v + e)

vertexNumber **=** 0**;**

fill**(**visited**,** visited **+** v**,** **false);**

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

**if(!**visited**[**i**])** TopologicalTraverse**(**i**);**

reverse**(**sortedVertexes**,** sortedVertexes **+** v**);**

**}**

***Articulation Points and Bridges***

// NOTE: Vertexes indexed from 0 to v - 1.

// Be careful using this function more than once.

// IMPORTANT: When calling the function, use the

// number of any vertex of the component and -1.

#include <set>

#include <vector>

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

#define MAXV 100000

set**<**int**>** ArticulationPoints**;** // Result

set**<** pair**<**int**,** int**>** **>** Bridges**;** // Result

int low**[**MAXV**];**

int num**[**MAXV**];**

int rootChild **=** 0**;**

int vertexCount **=** 1**;**

vector**<**int**>** E**[**MAXV**];**

void ArticulationPointsAndBridges**(**int u**,** int p**){**

vector**<**int**>::**iterator v**;**

low**[**u**]** **=** num**[**u**]** **=** vertexCount**++;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++){**

**if(**num**[\***v**]** **==** 0**){**

**if(**p **==** **-**1**)** rootChild**++;**

ArticulationPointsAndBridges**(\***v**,** u**);**

**if(**num**[**u**]** **<=** low**[\***v**])**

ArticulationPoints**.**insert**(**u**);**

**if(**num**[**u**]** **<** low**[\***v**])**

Bridges**.**insert**(**make\_pair**(**u**,** **\***v**));**

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

**}** **else** **if(\***v **!=** p**)**

low**[**u**]** **=** min**(**num**[\***v**],** low**[**u**]);**

**}**

**if(**p **==** **-**1 **&&** rootChild **<** 2**)**

ArticulationPoints**.**erase**(**u**);**

**}** // O(v + e)

***Strongly Connected Components***

// NOTE: Vertexes indexed from 0 to v - 1.

// Be careful using this function more than once.

#include <stack>

#include <vector>

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

#define MAXV 100000

vector**<** vector**<**int**>** **>** SCC**;** // Result

int low**[**MAXV**];**

int num**[**MAXV**];**

bool inStack**[**MAXV**];**

stack**<**int**>** Stack**;** // WARNING: This may be slow.

vector**<**int**>** E**[**MAXV**];**

int vertexCount **=** 1**;**

void StronglyConnected**(**int u**){** // O(v + e)

Stack**.**push**(**u**);**

inStack**[**u**]** **=** **true;**

vector**<**int**>::**iterator v**;**

low**[**u**]** **=** num**[**u**]** **=** vertexCount**++;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++){**

**if(**num**[\***v**]** **==** 0**)** StronglyConnected**(\***v**);**

**if(**inStack**[\***v**])** low**[**u**]** **=** min**(**low**[\***v**],** low**[**u**]);**

**}**

**if(**low**[**u**]** **==** num**[**u**]){**

vector**<**int**>** newSCC**;**

**while(!**Stack**.**empty**()){**

int v **=** Stack**.**top**();**

newSCC**.**push\_back**(**v**);**

inStack**[**v**]** **=** **false;**

Stack**.**pop**();**

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

**}**

SCC**.**push\_back**(**newSCC**);**

**}**

**}**

***Lowest Common Ancestor***

// NOTE: Vertexes indexed from 0 to v - 1.

// IMPORTANT: parent[root] must be root.

#include <algorithm>

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

#define MAXV 100000

#define LOGV 20

int level**[**MAXV**];** // Level of vertex u from root.

int parent**[**MAXV**];** // Direct parent of vertex u.

int P**[**MAXV**][**LOGV**];**

int logarithm**(**int v**){** // O(log v)

**for(**int i **=** 0**;** **true;** i**++)**

**if(**v **<=** 1 **<<** i**)** **return** i**;**

**}**

void PreprocessDP**(**int v**){** // O(v log v)

int log **=** logarithm**(**v**);**

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

P**[**i**][**0**]** **=** parent**[**i**];**

**for(**int j **=** 1**;** j **<=** log**;** j**++)**

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

P**[**i**][**j**]** **=** P**[**P**[**i**][**j **-** 1**]][**j **-** 1**];**

**}**

int LCA**(**int v**,** int a**,** int b**){** // O(log v)

int log **=** logarithm**(**v**);**

**if(**level**[**a**]** **<** level**[**b**])** swap**(**a**,** b**);**

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

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

a **=** P**[**a**][**i**];**

**if(**a **==** b**)** **return** a**;**

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

**if(**P**[**a**][**i**]** **!=** P**[**b**][**i**])**

a **=** P**[**a**][**i**],** b **=** P**[**b**][**i**];**

**return** P**[**a**][**0**];**

**}**

***Maximum Cardinality Bipartite Matching***

// NOTE: Number of vertexes v = L + R.

// Left vertexes indexed from 0 to L - 1.

// Right vertexes indexed from L to v - 1.

// Matching can be reconstructed from array match.

#include <vector>

#include <algorithm>

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

#define MAXV 1000

int match**[**MAXV**];**

bool visited**[**MAXV**];**

vector**<**int**>** E**[**MAXV**];**

int AlternatingPath**(**int u**){** // O(v + e)

**if(**u **==** **-**1**)** **return** 1**;**

**if(**visited**[**u**])** **return** 0**;**

visited**[**u**]** **=** **true;**

vector**<**int**>::**iterator v**;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++){**

**if(**AlternatingPath**(**match**[\***v**])){**

match**[\***v**]** **=** u**;**

**return** 1**;**

**}**

**}**

**return** 0**;**

**}**

int MaximumBipartiteMatching**(**int L**,** int R**){**

int matching **=** 0**;**

fill**(**match**,** match **+** L **+** R**,** **-**1**);**

**for(**int u **=** 0**;** u **<** L**;** u**++){**

fill**(**visited**,** visited **+** L**,** **false);**

matching **+=** AlternatingPath**(**u**);**

**}**

**return** matching**;**

**}** // O(v^2 + ve)

***Maximum Flow***

// Edmonds Karp implementation using BFS.

// NOTE: Vertexes indexed from 0 to n - 1.

// Remember to reset edges and capacity values.

// Complexity O(ve^2) but much better on practice.

#include <queue>

#include <vector>

#include <climits>

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

#define MAXV 100

#define INF INT\_MAX

int dist**[**MAXV**];**

int parent**[**MAXV**];**

int capacity**[**MAXV**][**MAXV**];**

vector**<**int**>** E**[**MAXV**];**

int AugmentingPath**(**int v**,** int source**,** int sink**){**

fill**(**dist**,** dist **+** v**,** INF**);**

fill**(**parent**,** parent **+** v**,** **-**1**);**

queue**<**int**>** bfs**;**

bfs**.**push**(**source**);**

dist**[**source**]** **=** 0**;**

vector**<**int**>::**iterator w**;**

**while(!**bfs**.**empty**()){**

int u **=** bfs**.**front**();**

bfs**.**pop**();**

**if(**u **==** sink**)** **break;**

**for(**w **=** E**[**u**].**begin**();** w **!=** E**[**u**].**end**();** w**++){**

**if(**capacity**[**u**][\***w**]** **>** 0 **&&** dist**[\***w**]** **==** INF**){**

dist**[\***w**]** **=** dist**[**u**]** **+** 1**;**

parent**[\***w**]** **=** u**;**

bfs**.**push**(\***w**);**

**}**

**}**

**}**

int u **=** sink**;**

int flow **=** INF**;**

**while(**parent**[**u**]** **!=** **-**1**){**

flow **=** min**(**capacity**[**parent**[**u**]][**u**],** flow**);**

u **=** parent**[**u**];**

**}**

**if(**flow **==** INF**)**

**return** 0**;**

u **=** sink**;**

**while(**parent**[**u**]** **!=** **-**1**){**

capacity**[**parent**[**u**]][**u**]** **-=** flow**;**

capacity**[**u**][**parent**[**u**]]** **+=** flow**;**

u **=** parent**[**u**];**

**}**

**return** flow**;**

**}** // O(e)

int MaximumFlow**(**int v**,** int source**,** int sink**){**

int maxFlow **=** 0**;**

int augmentFlow **=** 1**;**

**while(**augmentFlow **>** 0**){** // O(ve)

augmentFlow **=** AugmentingPath**(**v**,** source**,** sink**);**

maxFlow **+=** augmentFlow**;**

**}**

**return** maxFlow**;**

**}** // O(ve^2)

***Minimum Cut***

// NOTE: Vertexes indexed from 0 to n - 1.

// Maximum Flow required before using Minimum Cut.

// Remember to define value of UNDIRECTED properly.

#include <vector>

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

#define MAXV 100

#define UNDIRECTED true

vector**<** pair**<**int**,** int**>** **>** MinCut**;** // Result

bool inSetS**[**MAXV**];**

void ReachableFromSource**(**int u**){** // O(v)

inSetS**[**u**]** **=** **true;**

vector**<**int**>::**iterator v**;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++)**

**if(**capacity**[**u**][\***v**]** **&&** **!**inSetS**[\***v**])**

ReachableFromSource**(\***v**);**

**}**

void MinimumCut**(**int v**,** int source**){** // O(v + e)

MinCut**.**clear**();**

fill**(**inSetS**,** inSetS **+** v**,** **false);**

ReachableFromSource**(**source**);**

vector**<**int**>::**iterator w**;**

**for(**int u **=** 0**;** u **<** v**;** u**++){**

**for(**w **=** E**[**u**].**begin**();** w **!=** E**[**u**].**end**();** w**++){**

**if(**UNDIRECTED **&&** **\***w **<** u**)** **continue;**

**if(**inSetS**[**u**]** **==** inSetS**[\***w**])** **continue;**

MinCut**.**push\_back**(**make\_pair**(**u**,** **\***w**));**

**}**

**}**

**}**

**String Algorithms**

***Suffix Array***

// String must be indexed from 0 to n and

// finished by an '$' symbol or similar.

#include <stdio.h>

#include <string.h>

#include <algorithm>

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

#define MAXN 100000

int SA**[**MAXN**];**

int rank**[**MAXN**];**

int tray**[**MAXN**];**

int tempSA**[**MAXN**];**

int tempRank**[**MAXN**];**

void CountingSort**(**int n**,** int k**){** // O(n)

int ranks **=** max**(**n**,** 256**);**

fill**(**tray**,** tray **+** ranks**,** 0**);**

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

**if(**n **<=** i **+** k**)** tray**[**0**]++;**

**else** tray**[**rank**[**i **+** k**]]++;**

**for(**int i **=** 0**,** j **=** 0**;** i **<** ranks**;** i**++)**

j **+=** tray**[**i**],** tray**[**i**]** **=** j **-** tray**[**i**];**

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

**if(**n **<=** SA**[**i**]** **+** k**)** tempSA**[**tray**[**0**]++]** **=** SA**[**i**];**

**else** tempSA**[**tray**[**rank**[**SA**[**i**]** **+** k**]]++]** **=** SA**[**i**];**

copy**(**tempSA**,** tempSA **+** n**,** SA**);**

**}**

void SuffixArray**(**char**\*** s**,** int n**){** // O(n log n)

copy**(**s**,** s **+** n**,** rank**);**

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

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

CountingSort**(**n**,** i**);**

CountingSort**(**n**,** 0**);**

int ranks **=** 0**;**

tempRank**[**SA**[**0**]]** **=** 0**;**

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

**if(**rank**[**SA**[**j**]]** **!=** rank**[**SA**[**j **-** 1**]]** **||**

rank**[**SA**[**j**]+**i**]** **!=** rank**[**SA**[**j**-**1**]** **+** i**])**ranks**++;**

tempRank**[**SA**[**j**]]** **=** ranks**;**

**}**

copy**(**tempRank**,** tempRank **+** n**,** rank**);**

**if(**rank**[**SA**[**n **-** 1**]]** **==** n **-** 1**)** **break;**

**}**

**}**

// Starts code for Longest Common Prefix

int LCP**[**MAXN**];**

int prev**[**MAXN**];**

int plcp**[**MAXN**];**

void LongestCommonPrefix**(**char**\*** s**,** int n**){** // O(n)

prev**[**SA**[**0**]]** **=** **-**1**;**

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

prev**[**SA**[**i**]]** **=** SA**[**i **-** 1**];**

int L **=** 0**;**

plcp**[**SA**[**0**]]** **=** 0**;**

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

**if(**prev**[**i**]** **==** **-**1**)** **continue;**

**while(**prev**[**i**]** **+** L **<** n **&&** i **+** L **<** n

**&&** s**[**i **+** L**]** **==** s**[**prev**[**i**]** **+** L**])** L**++;**

plcp**[**i**]** **=** L**,** L **=** max**(**L **-** 1**,** 0**);**

**}**

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

LCP**[**i**]** **=** plcp**[**SA**[**i**]];**

**}**

***Knuth-Morris-Pratt***

// Before using Text Search call Pattern Fail.

// By default, Text Search does nothing, modify

// code tagged with "Matching" and return value

// to handle the way of processing matches.

#define MAXT 1000000

#define MAXP 1000000

char T**[**MAXT**];** // Text.

char P**[**MAXP**];** // Pattern.

int F**[**MAXP**];**

void PatternFail**(**int p**){** // O(p)

F**[**0**]** **=** **-**1**;** int j **=** **-**1**;**

**for(**int i **=** 0**;** i **<** p**;** i**++){**

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

F**[**i **+** 1**]** **=** **++**j**;**

**}**

**}**

void TextSearch**(**int t**,** int p**){** // O(t + p)

**for(**int i **=** 0**,** j **=** 0**;** i **<** t**;** i**++){**

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

**if(++**j **==** p**){**

// Matching

j **=** F**[**j**];**

**}**

**}**

**}**

**Computational Geometry**

***Points and Vectors***

#include <cmath>

#define PI M\_PI

#define ERROR 1e-12

#define RAD (PI / 180)

bool equal**(**double a**,** double b**){** // O(1)

**return** fabs**(**a **-** b**)** **<** ERROR**;**

**}**

// Functions to manipulate points and vectors

// in a 2D euclidean space, many functions may

// change if another dimension is required.

struct Point**{**

double x**,** y**;**

bool **operator** **<(**Point cmp**)** const**{**

**if(**equal**(**x**,** cmp**.**x**))**

**return** y **<** cmp**.**y**;**

**return** x **<** cmp**.**x**;**

**}**

**};**

Point P**(**double x**,** double y**){** // O(1)

Point**\*** p **=** **new** Point**;**

p**->**x **=** x**,** p**->**y **=** y**;**

**return** **\***p**;**

**}**

// Vectors represented as points.

Point Vector**(**Point o**,** Point p**){** // O(1)

**return** P**(**p**.**x **-** o**.**x**,** p**.**y **-** o**.**y**);**

**}**

Point Scale**(**Point p**,** double k**){** // O(1)

**return** P**(**p**.**x **\*** k**,** p**.**y **\*** k**);**

**}**

// Rotate a point around origin.

Point Rotate**(**Point p**,** double angle**){** // O(1)

double rad **=** angle **\*** RAD**;**

double x **=** cos**(**rad**)** **\*** p**.**x **-** sin**(**rad**)** **\*** p**.**y**;**

double y **=** sin**(**rad**)** **\*** p**.**x **+** cos**(**rad**)** **\*** p**.**y**;**

**return** P**(**x**,** y**);**

**}**

Point Translate**(**Point p**,** Point v**){** // O(1)

**return** P**(**p**.**x **+** v**.**x**,** p**.**y **+** v**.**y**);**

**}**

double Dot**(**Point u**,** Point v**){** // O(1)

**return** u**.**x **\*** v**.**x **+** u**.**y **\*** v**.**y**;**

**}**

double Cross**(**Point u**,** Point v**){** // O(1)

**return** u**.**x **\*** v**.**y **-** v**.**x **\*** u**.**y**;**

**}**

double Magnitude**(**Point v**){** // O(1)

**return** sqrt**(**Dot**(**v**,** v**));**

**}**

// Euclidean distance between point p and q.

double Distance**(**Point p**,** Point q**){** // O(1)

**return** hypot**(**p**.**x **-** q**.**x**,** p**.**y **-** q**.**y**);**

**}**

// Angle between two vectors.

double Angle**(**Point u**,** Point v**){** // O(1)

double U **=** Magnitude**(**u**);**

double V **=** Magnitude**(**v**);**

**if(**U **==** 0 **||** V **==** 0**)** **return** 0**;**

**return** acos**(**Dot**(**u**,** v**)** **/** **(**U **\*** V**))** **/** RAD**;**

**}**

// Angle POQ between three points.

double Angle**(**Point o**,** Point p**,** Point q**){** // O(1)

**return** Angle**(**Vector**(**o**,** p**),** Vector**(**o**,** q**));**

**}**

***Lines and Segments***

// Line structure consists in two points p and q

// and three doubles a, b and c for the line equation.

// Segments represented as lines, when using segments,

// points p and q are meant to be the edges and the

// values for line equation may not be necessary.

struct Line**{**

Point p**,** q**;**

double a**,** b**,** c**;**

**};**

Line L**(**Point p**,** Point q**){** // O(1)

**if(**q **<** p**)** **return** L**(**q**,** p**);**

Line**\*** l **=** **new** Line**();**

l**->**p **=** p**,** l**->**q **=** q**;**

l**->**c **=** Cross**(**p**,** q**);**

l**->**a **=** p**.**y **-** q**.**y**;**

l**->**b **=** q**.**x **-** p**.**x**;**

**return** **\***l**;**

**}**

// Determine if two lines are parallel.

bool Parallel**(**Line l**,** Line m**){** // O(1)

**return** equal**(**l**.**a**,** m**.**a**)** **&&** equal**(**l**.**b**,** m**.**b**);**

**}**

// Compare if two lines are the same.

bool **operator** **==(**Line l**,** Line m**){** // O(1)

**return** Parallel**(**l**,** m**)** **&&** equal**(**l**.**c**,** m**.**c**);**

**}**

// Intersection point of two lines.

Point Intersection**(**Line l**,** Line m**){** // O(1)

double div **=** m**.**a **\*** l**.**b **-** l**.**a **\*** m**.**b**;**

double x **=** m**.**b **\*** l**.**c **-** l**.**b **\*** m**.**c**;**

double y **=** m**.**a **\*** l**.**c **-** l**.**a **\*** m**.**c**;**

**return** P**(**x **/** div**,** y **/** **-**div**);**

**}**

Point CloserPointInLine**(**Point p**,** Line l**){** // O(1)

Point u **=** Vector**(**l**.**p**,** l**.**q**),** v **=** Vector**(**l**.**p**,** p**);**

double scale **=** Dot**(**u**,** v**)** **/** pow**(**Magnitude**(**u**),** 2**);**

**return** Translate**(**l**.**p**,** Scale**(**u**,** scale**));**

**}**

Point CloserPointInSegment**(**Point p**,** Line s**){** // O(1)

Point u **=** Vector**(**s**.**p**,** s**.**q**),** v **=** Vector**(**s**.**p**,** p**);**

double scale **=** Dot**(**u**,** v**)** **/** pow**(**Magnitude**(**u**),** 2**);**

**if(**scale **<** 0**)** **return** P**(**s**.**p**.**x**,** s**.**p**.**y**);**

**if(**scale **>** 1**)** **return** P**(**s**.**q**.**x**,** s**.**q**.**y**);**

**return** Translate**(**s**.**p**,** Scale**(**u**,** scale**));**

**}**

// Minimum distance between point p and line l.

double DistanceToLine**(**Point p**,** Line l**){** // O(1)

**return** Distance**(**CloserPointInLine**(**p**,** l**),** p**);**

**}**

// Minimum distance between point p and segment s.

double DistanceToSegment**(**Point p**,** Line s**){** // O(1)

**return** Distance**(**CloserPointInSegment**(**p**,** s**),** p**);**

**}**

// Counter clockwise test, returns true if point r

// is on the left side of line pq, false otherwise.

bool CCW**(**Point p**,** Point q**,** Point r**){** // O(1)

**return** Cross**(**Vector**(**p**,** q**),** Vector**(**p**,** r**))** **>** 0**;**

**}**

bool Collinear**(**Point p**,** Point q**,** Point r**){** // O(1)

**return** equal**(**Cross**(**Vector**(**p**,** q**),** Vector**(**p**,** r**)),** 0**);**

**}**

bool PointInSegment**(**Point p**,** Line s**){**

double y1 **=** s**.**p**.**y **<** s**.**q**.**y**?** s**.**p**.**y**:** s**.**q**.**y**;**

double y2 **=** s**.**p**.**y **>** s**.**q**.**y**?** s**.**p**.**y**:** s**.**q**.**y**;**

**if(**p**.**x **<** s**.**p**.**x **||** s**.**q**.**x **<** p**.**x**)** **return** **false;**

**if(**p**.**y **<** y1 **||** y2 **<** p**.**y**)** **return** **false;**

**return** Collinear**(**p**,** s**.**p**,** s**.**q**);**

**}**

***Polygons***

// Polygons represented as a vector of points,

// the first point in vector is also the last one.

#include <vector>

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

// Perimeter of polygon.

double Perimeter**(**vector**<**Point**>** P**){**

double result **=** 0**;**

**for(**int i **=** 1**;** i **<** P**.**size**();** i**++)**

result **+=** Distance**(**P**[**i **-** 1**],** P**[**i**]);**

**return** result**;**

**}**

// Area of polygon.

double Area**(**vector**<**Point**>** P**){**

double result **=** 0**;**

**for(**int i **=** 1**;** i **<** P**.**size**();** i**++)**

result **+=** Cross**(**P**[**i **-** 1**],** P**[**i**]);**

**return** fabs**(**result**)** **/** 2.0**;**

**}**

// Tests if a point p is inside of polygon P.

// If the point is on the bounds is considered inside.

bool PointInPolygon**(**Point p**,** vector**<**Point**>** P**){**

**for(**int i **=** 1**;** i **<** P**.**size**();** i**++)**

**if(**PointInSegment**(**p**,** L**(**P**[**i**],** P**[**i **-** 1**])))**

**return** **true;**

double angle **=** 0**;**

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

double alpha **=** Angle**(**p**,** P**[**i **-** 1**],** P**[**i**]);**

**if(**CCW**(**p**,** P**[**i**],** P**[**i **-** 1**]))**

alpha **=** **-**alpha**;**

angle **+=** alpha**;**

**}**

**if(**round**(**angle**)** **==** 360**)**

**return** **true;**

**return** **false;**

**}**

***Convex Hull***

// Receives a vector of points in the 2D plane,

// returns a polygon with the points that forms

// the convex hull sorted in counter-clockwise order.

// NOTE: Function deletes collinear points, if this

// points are required, modify CCW function.

#include <vector>

#include <algorithm>

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

vector**<**Point**>** ConvexHull**(**vector**<**Point**>** P**){**

vector**<**Point**>** U**,** L**;**

sort**(**P**.**begin**(),** P**.**end**());**

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

**while(**L**.**size**()** **>** 1**){**

Point p **=** L**[**L**.**size**()** **-** 2**];**

Point q **=** L**[**L**.**size**()** **-** 1**];**

**if(**CCW**(**p**,** q**,** P**[**i**]))** **break;**

**else** L**.**pop\_back**();**

**}**

L**.**push\_back**(**P**[**i**]);**

**}**

**if(**L**.**size**()** **>** 1**)** L**.**pop\_back**();**

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

**while(**U**.**size**()** **>** 1**){**

Point p **=** U**[**U**.**size**()** **-** 2**];**

Point q **=** U**[**U**.**size**()** **-** 1**];**

**if(**CCW**(**p**,** q**,** P**[**i**]))** **break;**

**else** U**.**pop\_back**();**

**}**

U**.**push\_back**(**P**[**i**]);**

**}**

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

L**.**push\_back**(**U**[**i**]);**

**return** L**;**

**}**