Table of Contents

FIRST: PROBLEMS [WITH SOLUTION]

QUESTION1: DIGOKEYS - Find the Treasure

A psychic lock-lover called Digo likes playing games with Locks and Keys and also has very good logic. One day he buys a set of **N** boxes, each of them has an index between {**1, N**} (inclusive) and no two boxes have same index. There is a key inside every box except the **Nth** box which has great treasure. He eventually finds out that due to a defect in the manufacturing of the keys, most of them could open more than one box.

The rule is that you are allowed to open only one box with any key. Each box except the first is locked. Now as Digo couldn't wait long to acquire that great treasure he requests you to find a method to open the last box starting with the key in the first box with **minimum number of steps**.

INPUT

In first line, **T** no. of test cases.

For every test case:

In first line, there is an integer **N** : (number of locks)

In next **N-1** lines, on the **i'th** line there is an integer **Mi** the number of boxes which the key present in the **i'th** box can open. It is followed by the **Mi** integers (the indices of those boxes that can be opened by the key present in **i'th** box).

OUTPUT

For every test case:

one integer **q**, minimum number of boxes opened.

In the next line : the indices of the boxes opened in order separated by space. If there are many solutions print the one which is lexicographically smallest.

If there is no way to reach the last box print "-1".

Each test case is to be followed by a blank line.

Sample Input	Sample Output
2 3 1 2 1 3 4 2 2 3 1 1 2 2 4	2 1 2 2 1 3

Answer

```
main()
{
      NumOfCases = ReadInt()
      while(NumOfCases--)
      {
            N = ReadInt()
            for i=1 to N
            {
                  Mi = ReadInt()
                  while(Mi--)
                  {
                        box = ReadInt()
                        keys[i].insert(box)
                  }
            }
            parent[1] = 0
            parent[2…N] = -1

            box=1
            q.enqueue(box);
            res=false;
            while(!q.empty())
            {
                  k = q.dequeue();
                  For each box ∈ keys[k]
                  {
                        if(parent[box] == -1)
                        {
                              parent[box]=k;
                              if(box == N)
                              {
                                    res=true;
                                    break;
                              }
                              q.enqueue(box);
                        }
                  }
                  if(res==true)
                        break;
            }
            if(res==false)
                  printf("NOT FOUND")
            else
                  for(i=parent[N]; I != 0 ; i=parent[i])
                        print(i)
      }
}
```

QUESTION2: Graph Radius

In any undirected graph $G = (V, E)$, the **eccentricity** $e(u)$ of a vertex $u \in V$ is the shortest distance to its farthest vertex v, i.e., $e(u) = \max\{\delta(u, v) \mid v \in V\}$. The **radius** $R(G)$ of an undirected graph $G = (V, E)$ is the smallest eccentricity of any vertex, i.e., $R(G) = \min\{e(u) \mid u \in V\}$.

(a) Given connected undirected graph G, describe an $O(|V||E|)$-time algorithm to determine the radius of G.

(b) Given connected undirected graph G, describe an $O(|E|)$-time algorithm to determine an upper bound R^* on the radius of G, such that $R(G) \leq R^* \leq 2R(G)$.

Answer

a) Compute $R(G)$ directly: run breadth-first search from each node u and calculate $e(u)$ as the maximum of $\delta(u, v)$ for each $v \in V$. Then return the minimum of $e(u)$ for all $u \in V$. Each vertex is connected to an edge because G is connected so $|V| = O(|E|)$. Each BFS takes $O(|E|)$ time, leading to $O(|V||E|)$ time in total

b) Use breadth-first search to find $e(u)$ for any $u \in V$ in $O(|E|)$ time. We claim that $R(G) \leq e(u) \leq 2R(G)$, so we can choose $R^* = e(u)$. First, $e(u) \geq R(G)$ because $R(G)$ is the minimum $e(v)$ over all $v \in V$. Second, $2R(G) \geq e(u)$ because if a vertex x has eccentricity $R(G)$, we can construct a path from u to any other vertex v by concatenating a shortest path from u to x and a shortest path from x to v, both of which have length at most $R(G)$; thus $e(u)$ cannot be greater than $2R(G)$.

QUESTION3: DAG Root

Let $G = (V, E)$ be a DAG, where $|V| = n$ and $|E| = m$. A vertex v is called a *root* of G if there is a path from v to every vertex in G. A DAG may or may not have a root in general. Describe briefly an $O(n + m)$ time algorithm to check if G has a root. How should G be represented in order to achieve the desired time complexity of $O(n + m)$?

Answer

The idea is to run topological sorting on G and then check if the first vertex (instead of the last vertex) in the sorted order has a path to every other vertex in G by starting a DFS or BFS at this vertex.

If G is represented as adjacency lists, then this algorithm runs in time O(n + m) since both topological sorting and DFS/BFS take O(n + m) time.

QUESTION4: Ancient Avenue

The ancient kingdom of Pesomotamia was a fertile land near the Euphris and Tigrates rivers. Newly discovered clay tablets show a map of the kingdom on an $n \times n$ square grid, with each square either:

- part of a river, labeled as 'euphris' or 'tigrates'; or

- not part of a river, labeled with a string: the name of the farmer who owned the square plot of land.

The tablets accompanying the map state that ancient merchants built a **trade route** connecting the two rivers: a path along edges of the grid from some grid intersection adjacent to a 'euphris' square to a grid intersection adjacent to a 'tigrates' square. The tablets also state that the route:

- did not use any edge between two squares owned by the same farmer (so as not to trample crops); and
- was the shortest such path between the rivers (assume a shortest such path is unique).

Given the labeled map, describe an $O(n^2)$-time algorithm to determine path of the ancient trade route?

Answer

Let M be the $n \times n$ labeled map, where $M[r][c]$ is the label of the grid square in row r and column c. Construct graph G with $(n + 1)^2$ vertices (r, c) for all $r, c \in \{0, \ldots, n\}$, and an undirected edge between:

• vertex $(r, c - 1)$ and vertex (r, c) for $r \in \{0, \ldots, n\}$ and $c \in \{1, \ldots, n\}$, except when:

- $r \in \{1, \ldots, n - 1\}$ (not a boundary edge),
- $M[r][c]$ is not 'euphris' or 'tigrates' (is owned by a farmer), and
- $M[r][c - 1] = M[r][c]$ (owned by the same farmer);

• vertex $(r - 1, c)$ and vertex (r, c) for $c \in \{0, \ldots, n\}$ and $r \in \{1, \ldots, n\}$, except when:

- $c \in \{1, \ldots, n - 1\}$ *(not a boundary edge),*
- *$M[r][c]$ is not 'euphris' or 'tigrates' (is owned by a farmer), and*
- *$M[r - 1][c] = M[r][c]$ (owned by the same farmer).*

This graph has the property that a path between any two vertices corresponds to a route on the map that does not trample crops. It remains how to find the shortest such route between any vertex adjacent to a 'euphris' square to a vertex adjacent to a 'tigrates' square.

For each 'euphris' square $M[r][c]$, mark vertices $\{(r, c), (r+1, c), (r, c+1), (r+1, c+1)\}$ as 'euphris' vertices (possible starting intersection of the route), and mark 'tigrates' vertices similarly. We could solve SSSP from each 'euphris' vertex using BFS, but that would take too much time (potentially $\Omega(n^2)$ 'euphris' vertices, with each BFS taking $\Omega(n^2)$ time.)

So instead, add a supernode s to G with an undirected edge from s to each 'euphris' vertex in G, to construct graph $G0$. Then any possible trade route will correspond to a path from s to a 'tigrates' vertex in $G0$ (without the first edge), and the trade route will be the shortest among them. So run BFS from s to every 'tigrates' vertex, and return the shortest path to the closest one by traversing parent pointers back to the source. Graph $G0$ has $(n + 1)^2 + 1$ vertices and $O(n^2) + O(n^2)$ edges, so can be constructed in $O(n^2)$ time, and running BFS once from s also takes $O(n^2)$ time, so this algorithm runs in $O(n^2)$ time in total

QUESTION5: Dynamite Detonation

Superspy Bames Jond is fleeing the alpine mountain lair of Silvertoe, the evil tycoon. Bames has stolen a pair of skis and a trail map listing the mountain's clearings and slopes (n in total), and she wants to ski from the clearing L by the lair to a clearing S where a snowmobile awaits.

- Each **clearing** $c_i \in C$ has an integer **elevation** e_i above sea level.
- Each **slope** (c_i, c_j, l_{ij}) connects a pair of clearings c_i and c_j with a **monotonic** trail (strictly decreasing or increasing in elevation) with positive integer length l_{ij}. Bames doesn't have time to ski uphill, so she will only traverse slopes so as to decrease her elevation.
- On her way up the mountain, Bames laced the mountain with **dynamite** at known locations $C_D \subset C$. Detonating the dynamite will immediately change the elevation of clearings $c_i \in C_D$ by a known amount, from e_i to a lower elevation $e'_i < e_i$. The **detonator** exists at clearing $D \in C$ (where there is no dynamite). If she reaches clearing D, she can choose to detonate the dynamite before continuing on.

Given Bames' map and dynamite data, describe an $O(n)$-time algorithm to find the minimum distance she must ski to reach the snowmobile (possibly detonating the dynamite along the way).

Answer

We can model the mountain as a DAG of trails that Bames may ski downhill, both before and after possible detonation. Construct a graph G_1 with a vertex for every clearing and a directed edge for every slope pointing to the lower clearing; remove any slopes that do not change height along the way (since Bames will not traverse the slope in either direction). Construct a second graph G_2 in the same way as G_1, except using the modified heights e'_i that represents the state of the mountain after the dynamite explodes. In both graphs, we weight edges by the length l_{ij} of the corresponding slope. Connect G_1 and G_2 into a single graph G by adding one directed edge from node D_1 in graph G_1 to node D_2 in graph G_2 with weight 0 (taking this edge represents blowing up the dynamite); and add a supernode T that has incoming edges of zero weight from nodes S_1 in graph G_1 and S_2 in graph G_2 (representing reaching S by either not detonating or detonating the dynamite respectively). We can now run DAG Relaxation from node L_1 in graph G to the supernode T; the shortest such path is the minimum distance to get from L to S, with or without detonation, while respecting that Bames only decreases her elevation. Graph G has $O(n)$ vertices and edges, so DAG Relaxation runs in $O(n)$ time, as desired.

QUESTION6: Cloud Computing

Azrosoft Micure is a cloud computing company where users can upload **computing jobs** to be executed remotely. The company has a large number of identical cloud computers available to run code, many more than the number of pieces of code in any single job. Any

piece of code may be run on any available computer at any time. Each computing job consists of a code list and a dependency list.

A **code list** C is an array of code pairs $(f, t) \in C$, where string f is the file name of a piece of code, and t is the positive integer number of microseconds needed for that code to complete when assigned to a cloud computer. Assume file names are short and can be read in $O(1)$ time.

A **dependency list** D is an array of dependency pairs $(f1, f2) \in D$, where $f1$ and $f2$ are distinct file names that appear in C. A dependency pair $(f1, f2)$ indicates that the piece of code named $f1$ must be completed before the piece of code named $f2$ can begin. Assume that every file name exists in some dependency pair

(a) A job (C, D) can be **completed** if every piece of code in C can be completed while respecting the dependencies in D. Given job (C, D), describe an $O(|D|)$-time algorithm to decide whether the job can be completed?

(b) Azrosoft Micure wants to know how fast they can complete a given job. Given a job (C, D), describe an O(|D|)-time algorithm to determine the minimum number of microseconds that would be needed to complete the job (or return that the job cannot be completed).

Answer

a) **Solution:** Construct a graph G with a vertex for each piece of code in C and a directed edge from $f1$ to $f2$ for each dependency pair $(f1, f2) \in D$. Also add an auxiliary node s and add a directed edge (s, f) to every other vertex. A job can be completed as long as G does not contain any cycle. We can do this by running a depth-first search (DFS) from s, checking whether the reverse order of DFS finishing times is a topological order (i.e., check to make sure no edge in the graph goes against this order). Since $|C| \leq 2|D|$, G has size $O(|D|)$ and DFS can be be run in $O(|D|)$ time. Checking each edge against this order can also be done in $O(1)$ time per edge, so this algorithm takes $O(|D|)$ time as desired.

b) **Solution:** We can use the same graph as in part (a), adding weights corresponding to running times of pieces of code. Let $t(f)$ denote the time for code f to complete. Then for each edge (a, f) in G, weight it by $-t(f)$. Then the length of the shortest path from s to any other vertex in G will be equal to longest time for any piece of code to be completed. If G has a cycle, we can break as in (a); otherwise, G is a DAG, and we can run DAG Relaxation to compute the shortest path in $O(|D|)$ time

---

QUESTION7: Under Game

Atniss Keverdeen is a rebel spy who has been assigned to go on a reconnaissance mission to the mansion of the tyrannical dictator, President Rain. To limit exposure, she has decided to

travel via an underground sewer network. She has a map of the sewer, composed of n bidirectional pipes which connect to each other at junctions. Each junction connects to at most four pipes, and every junction is reachable from every other junction via the sewer network. Every pipe is marked with its positive integer length, while some junctions are marked as containing identical motion sensors, any of which will be able to sense Atniss if her distance to that sensor (as measured along pipes in the sewer network) is too close. Unfortunately, Atniss does not know the sensitivity of the sensors. Describe an $O(n \log n)$ time algorithm to find a path along pipes from a given entrance junction to the junction below President Rain's mansion that stays as far from motion sensors as possible.

Answer

Construct a graph G with a vertex for every junction in the sewer network and an undirected edge between junction a and junction b with weight w if a and b are connected by a pipe with length w. Since each junction connects to at most a constant number of pipes, the number of edges and vertices in this graph are both upper bounded by $O(n)$.

Let s be the vertex associated with the entrance junction, and let t be the junction below President Rain's mansion. Define G_k to be the subgraph of G on only the vertices whose distance to any sensor is strictly larger than k. Our goal will be to find k^* such that s and t are in the same connected component in G_{k^*}, but s and t are not in the same connected component in G_{k^*+1}, i.e., k^* is the maximum sensor sensitivity for which it is still possible for Atniss to reach t from s undetected. If we could find k^*, then we could just return any path from s to t in G_{k^*}.

To find k^*, first label each junction with its shortest distance to any sensor. To do this, construct a new graph G_0 from G by adding an auxiliary vertex x *(super node)* and an undirected edge of weight zero from x to every vertex in G marked as containing a motion sensor, and then run **Dijkstra** from x in $O(n \log n)$ time. Then $\delta(x, v)$ corresponds to the shortest distance labels from v to any sensor as desired. Then we can construct G_k for any k in $O(n)$ time by looping through vertices, removing any vertex v for which $\delta(v, k) \leq k$ (also removing any edge adjacent v). Further, we can check whether t is reachable from s in G_k in $O(n)$ time via **breath-first search** or depth-first search.

We could find k^* in $O(nk^*)$ time by simply constructing G_k for every $k \in \{1, \ldots, k^* + 1\}$, but we can find k^* faster via **binary search**. Sort the vertices by their distance $\delta(x, v)$ from any sensor in $O(n \log n)$ time using any optimal comparison sort algorithm (e.g., **merge sort**), and let d_i be the ith largest distance in this sorted order for $i \in \{1, \ldots, n\}$, where d_1 is the smallest and d_n is the largest. Then construct G_{d_i} for $i = n/2$ and check whether t is reachable from s in G_{d_i} in $O(n)$ time. If it is, recurse for $i > n/2$, and if not, recurse for $i < n/2$. The binary search proceeds in $O(\log n)$ iterations that each take at most $O(n)$ time until finding the smallest $d_i = d^*$ such that t is not reachable from s. Then t is reachable from s for $k < d^*$ but not for $k \geq d^*$, so $k^* = d^* - 1$.

Finally, we can use either **breadth-first search** or depth-first search while storing parent pointers to reconstruct some path from s to t in G_{k^*} in $O(n)$ time. If $k^* = 0$, every path from s to t goes through a junction containing a sensor, so any path from s to t may be returned.

QUESTION8: Critter Collection

Ashley Getem (from PS3) is trying to walk from Trundle Town to Blue Bluff, which are both clearings in the Tanko region. She has a map of all clearings and two-way trails in Tanko. Each of the n clearings connects to at most five trails, while each trail t directly connects two clearings and is marked with its length l_t and capacity c_t of critters living on it, both positive integers. Ashley is a compulsive collector and will collect every critter she comes across by throwing an empty Pocket Sphere at it (which will fill the Pocket Sphere so it can no longer be used). If she encounters a critter without an empty Pocket Sphere, she will be sad. Whenever Ashley reaches a clearing, all critters on all trails will respawn to max capacity. Some clearings contain stores where Ashley can buy empty Pocket Spheres, and deposit full ones. Ashley has more money than she knows what to do with, but her backpack can only hold k Pocket Spheres at a time. Given Ashley's map, describe an $O(nk \log(nk))$-time algorithm to return the shortest route for Ashley to walk from Trundle Town (with a backpack full of empty Pocket Spheres) to Blue Bluff without ever being sad, or return that sadness is unavoidable.

Answer

Construct a graph $G = (V, E)$ with $k+1$ vertices for each clearing, where vertex $v_{c,i}$ corresponds to being at clearing c while having i empty pocket spheres. Then, for each directed pair of clearings (a, b) for which a and b are connected by a trial t, having length l_t and critters, add the following directed edges:

- If clearing a does not contain a store, add an edge of weight l_t from $v_{a,i}$ to $v_{b,i-c_t}$ for every $i \in \{c_t, \ldots, k\}$, since Ashley will use up c_i pocket spheres by traversing the trail.
- Otherwise, if clearing a contains a store, add an edge of weight l_t from $v_{a,i}$ to $v_{b,k-c_t}$ for every $i \in \{0, \ldots, k\}$, since it is never bad for Ashley to completely fill her backpack with empty pocket spheres whenever she leaves a store.

G has $(k + 1)n = O(kn)$ vertices and at most k edges per trail. Since there are at most five trails per clearing, there are at most $5kn = O(kn)$ edges in G, so G has size $O(nk)$. Let s be the vertex associated with Trundle Town and t be the vertex associated with Blue Bluff. Then any path from $v_{s,k}$ to any vertex $v_{t,i}$ for $i \in \{0, \ldots, k\}$ in G avoids sadness, and every path that avoids sadness corresponds to a path in G. Since the weights in G are non-negative, we can run **Dijkstra**, storing parent pointers to reconstruct a shortest route from s to t that avoids sadness in $O(nk \log(nk))$ time. If the shortest path weight to every $v_{t,I}$ in G is infinite, then return that sadness is unavoidable.

SECOND: PROBLEMS [WITHOUT SOLUTION]

QUESTION1: Transformation: from A to B

Vasily has a number a, which he wants to turn into a number b. For this purpose, he can do two types of operations:

- multiply the current number by 2 (that is, replace the number x by $2 \cdot x$);
- append the digit 1 to the right of current number (that is, replace the number x by $10 \cdot x + 1$).

You need to help Vasily to transform the number a into the number b using only the operations described above, or find that it is impossible.

Note that in this task you are not required to minimize the number of operations. It suffices to find any way to transform a into b.

Input

The first line contains two positive integers a and b ($1 \le a < b \le 10^9$) — the number which Vasily has and the number he wants to have.

Output

If there is a way to get b from a, print 1.

If there is no way to get b from a, print -1.

Examples

```
2 162
1
```

```
4 42
-1
```

```
100 40021
1
```

QUESTION2: Maze with Traps

Jarmtin is interested in cultures and the history behind them. Of course this interest has a reason: as he studies the choivans' past he discovers the hidden entrances of mazes he knows contain valuable information. However there is a catch: the mazes contain spiky traps! Jarmtin is quite the agile type, but there is a limit to everyone, thus he will only be able to avoid a number of traps. This motivates the question can he make it through the mazes?

The question is there a path from any @ to x such that number of spikes in that path is less than j/2

Input

The first line of a test case contains three integers n, m and j. n (2<=n<=40) the number of rows, m (2<=n<=40) the width of each row and j (0<=j<=20) the number of times Jarmtin can avoid spikes. Then n lines containing m characters; The character 'x' will be used for the place of the treasure, '@' for an entrance (which is also an exit), '#' for walls, '.' for a safe walking tile and 's' for spikes. Note that you cannot walk into walls and the maze is completely surrounded by walls outside what you can see. There is always at least one entrance/exit and always an x where the treasure is.

Output

You should output "SUCCESS" if Jarmtin can make it in and out alive, and "IMPOSSIBLE" if there is no way you can make it out alive.

Sample Input / Output

Example 1:

Input:

3 3 2
#@#
#s#
#x#

Output: SUCCESS

Example 2:

Input:

4 4 3
####
@.s#
##.#
#xs#

Output: IMPOSSIBLE

QUESTION3: Restaurant Placement

Smiley Donuts, a new donuts restaurant chain, wants to build restaurants on many street corners with the goal of maximizing the number of locations.

The street network is described as an undirected graph $G = (V, E)$, where the potential restaurant sites are the vertices of the graph. Two restaurants cannot be built on adjacent vertices (to avoid self-competition). Suppose that the street network G is acyclic, i.e., a tree. Design an algorithm to find the restaurant placement with the largest number of locations?

QUESTION4: SparkPlug Derby

LightQueen McNing is a race car that wants to drive to California to compete in a road race: the annual SparkPlug Derby. She has a map depicting:

- the *n* intersections in the country, where each intersection *xi* is marked with its positive integer **elevation** *ei* and whether it contains a **gas station**; and
- the *r* roads connecting pairs of them, where each road *rj* is marked with the positive integer *tj* denoting the **driving time** it will take LightQueen to drive along it in either direction.

Some intersections connect to many roads, but the average number of roads at any intersection is less than 5. LightQueen needs to get from Carburetor Falls at intersection *s* to the SparkPlug Derby race track at intersection *t*, subject to the following conditions:

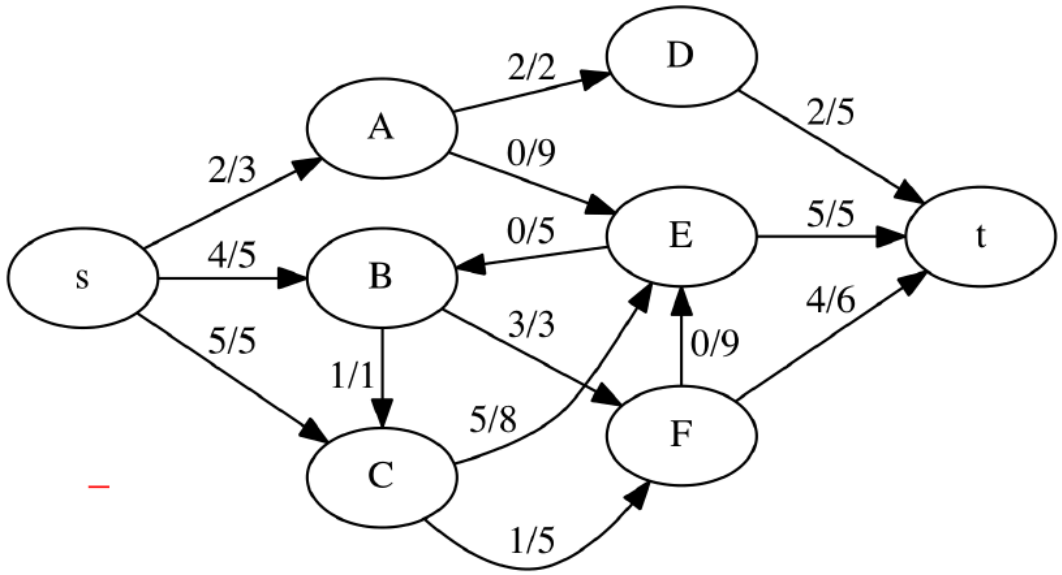
- LightQueen's gas tank has a positive integer **capacity** $g < n$: it can hold up to *g* units of gas at a time (starting full). Along the way, she can refill her tank (by any integer amount) at any intersection marked with a gas station. It takes exactly *tG* time for her to fill up 1 unit of gas.
- LightQueen uses gas only when **driving uphill**. Specifically, if she drives on a road from intersection *xi* to *xj* at elevations *ei* and *ej* respectively, LightQueen will use exactly $ej - ei$ units of gas to travel along it if $ej > ei$, and will use zero units of gas otherwise.

Given LightQueen's map, describe an $O(n^2 \log n)$-time algorithm to return a fastest route to the race that keeps a **strictly positive** amount of gas in her tank at all times (if such a route exists).

THIRD: TRACE QUESTIONS

QUESTION1: MAX-FLOW/MIN-CUT

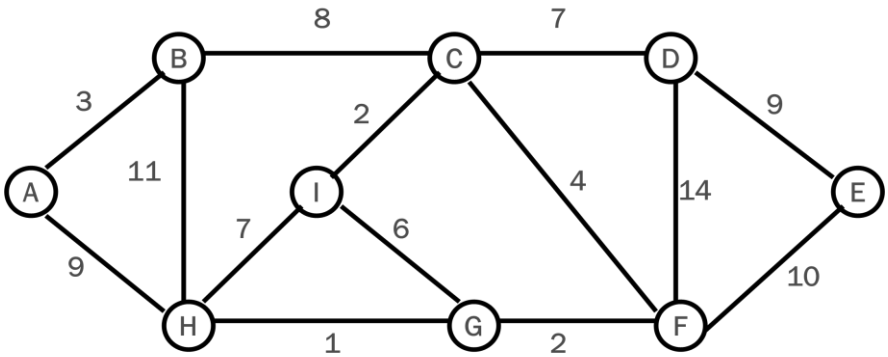
Consider the graph below



(a) What is the value of the flow shown above?
(b) What is the value of the max flow?
(c) Which vertices are on the s side of the min cut?

QUESTION2: Minimum Spanning Tree [Kruskal]

For the following graph, Show the **sequence of edges** in the minimum spanning tree (MST) in the order that **Kruskal's algorithm** includes them (by specifying the edge sides and weight)?

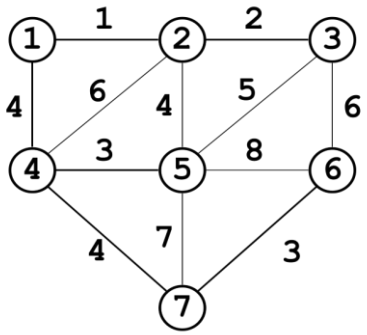


Answer

H-G:1, G-F:2, C-I:2, A-B:3, C-F:4, C-D:7, B-C:8, D-E:9

QUESTION3: Minimum Spanning Tree [Prim]

In the following Graph:



Starting from vertex number 1, apply the **Prim's** algorithm to find the minimum spanning tree (MST) and write-down the **final content** of the **distance array** (d[]) and the **parent array** (p[])?

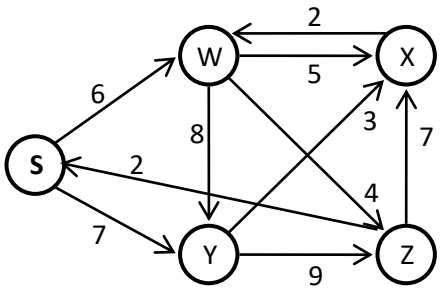
Vertex	1	2	3	4	5	6	7
Distance (d[])	0						
Parent (p[])	NIL						

Answer

Vertex	1	2	3	4	5	6	7
Distance (d[])	0	1	2	4 / 3	3 / 4	3	4
Parent (p[])	NIL	1	2	1 / 5	4 / 2	7	4

QUESTION4: Single Source Shortest Path (Dijkstra I)

Starting from vertex **S**, apply the **Dijkstra's** algorithm to find the shortest path to all other vertices and write-down the **final content** of the **distance array** (d[]) and the **parent array** (p[])?



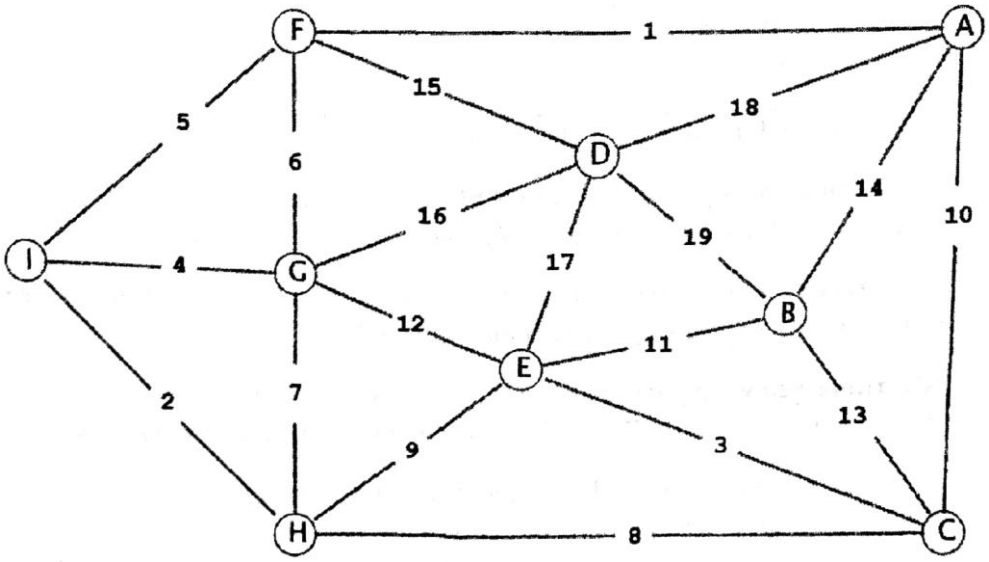
Vertex	S	W	X	Y	Z
Distance (d[])	0				
Parent (p[])	NIL				

Answer

Vertex	S	W	X	Y	Z
Distance (d[])	0	6	10	7	10
Parent (p[])	NIL	S	Y	S	W

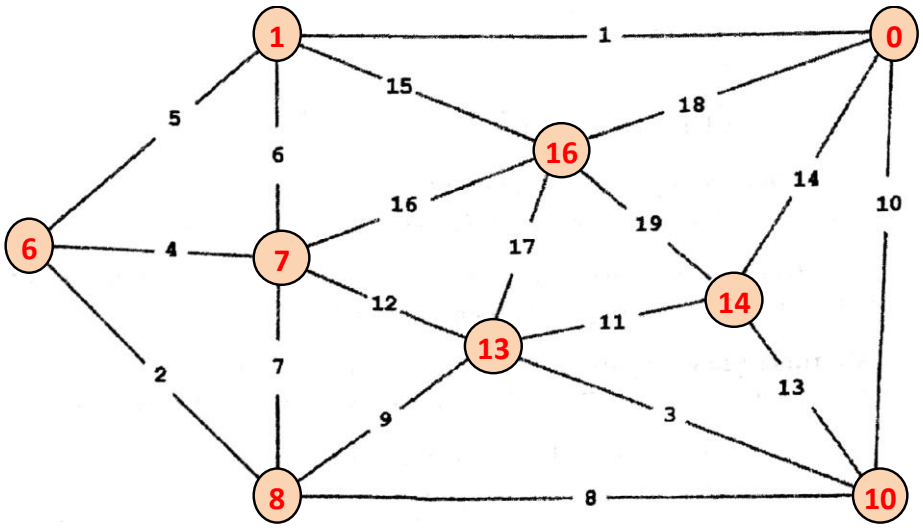
QUESTION5: Single Source Shortest Path (Dijkstra II)

In the following Graph:



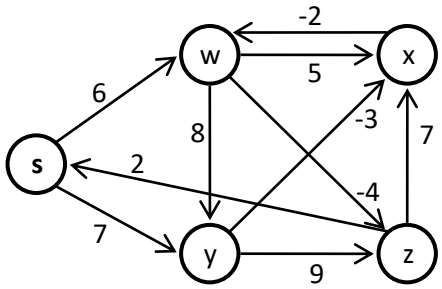
Using **Dijkstra's algorithm**, calculate the shortest path from vertex A to all other vertices? (For each destination vertex, just write down the **path cost only,** not the path itself)

Answer



QUESTION6: Single Source Shortest Path (Bellman-Ford)

Apply the Bellman-Ford algorithm to find the shortest path from vertex ***s*** to all other vertices:



1. What's the shortest path value from ***s*** to ***z***? What's the path itself?
2. From the filled DP table, find the shortest path value in each of the following cases:

(dest. vertex, max edges)	(y,0)	(y,1)	(w,2)	(z,3)	(w,3)	(x,4)
shortest path value	∞					

3. If the edge weight from ***x*** to ***w*** is changed to -4 instead of -2, can we find a shortest path from ***s*** to all other vertices? Why or why not?

Answer

1. What's the shortest path value from ***s*** to ***z***? What's the path itself?
   Value = -2, Path = s → y → x → w → z
2. From the filled DP table, find the shortest path value in each of the following cases:

(dest. vertex, max edges)	(y,0)	(y,1)	(w,2)	(z,3)	(w,3)	(x,4)
shortest path value	∞	7	6	2	2	4

3. If the edge weight from ***x*** to ***w*** is changed to -4 instead of -2, can we find a shortest path from ***s*** to all other vertices? Why or why not?
   No. It causes a negative cycle (w, z, x)

QUESTION7: Graph Traversal (DFS)

For the **above graph**, starting from vertex ***x***, apply the **Depth First Search** algorithm and answer the following questions (break ties (if any) by selecting the vertices according to an ascending alphabetic order):

1. Write-down the **discovery time (d[])** and **finish time (f[])** of each vertex?

Vertex	x	s	w	y	z
discovery (d[])	1				
finish (p[])					

2. What's the type of each of the following edges (normal, forward, backward, cross)?

Edge	(z, s)	(y, x)	(w, z)
Type			

Answer

1. Write-down the **discovery time (d[])** and **finish time (f[])** of each vertex?

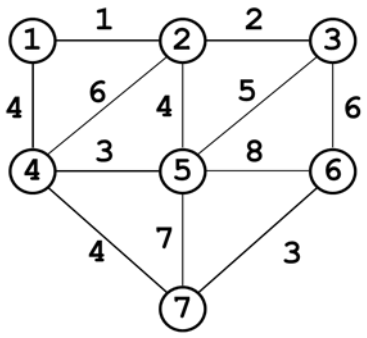
Vertex	x	s	w	y	z
discovery (d[])	1	5	2	3	4
finish (p[])	10	6	9	8	7

2. What's the type of each of the following edges (normal, forward, backward, cross)?

Edge	(z, s)	(y, x)	(w, z)
Type	normal	backward	forward

QUESTION8: Graph Traversal (BFS)

In the following undirected Graph:



Starting from vertex number 7, apply the **Breadth First Search (BFS)** algorithm and write-down the **discovery time (d[])** (i.e. level of each vertex) and **the parent** of each vertex? (break ties (if any) by selecting the vertices according to an ascending order)

Vertex	1	2	3	4	5	6	7
Discovery time (d[])							
parent (p[])							

Answer

Vertex	1	2	3	4	5	6	7
Discovery time (d[])	2	2	2	1	1	1	0
parent (p[])	4	4	5	7	7	7	NiL

FOURTH: Graph Applications

QUESTION1: Connected Components

Modify the following BFS algorithm to detect all connected components (i.e. graph pieces) in the given **undirected** graph? Components should be sequentially indexed starting from one. For each component, all vertices that belong to it should be assigned to the same index in **comp[v]** (e.g. check the example below)? *(ONLY copy code that contains your modification)*

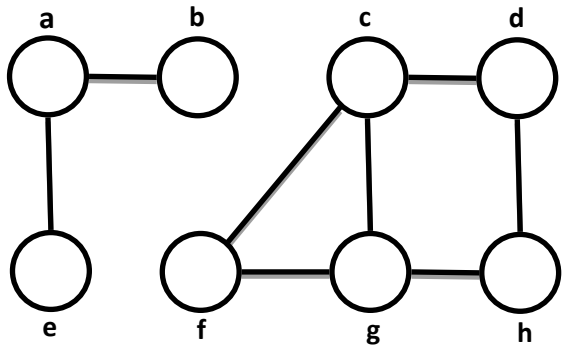
∏[v]: is the predecessor of v **d[v]**: distance from root to v **s**: start node
comp[v]: ***index*** of component that contains v [**TO BE FILLED** in your solution]

```
BFS(G, s)
1. For each u ∈ G.V - {s}
2.     color[u] = WHITE
3.     d[u] = ∞
4.     ∏[u] = NIL
5. color[s] = GRAY
6. d[s] = 0
7. ∏[s] = NIL
8. BFS-Visit(G, s)
```

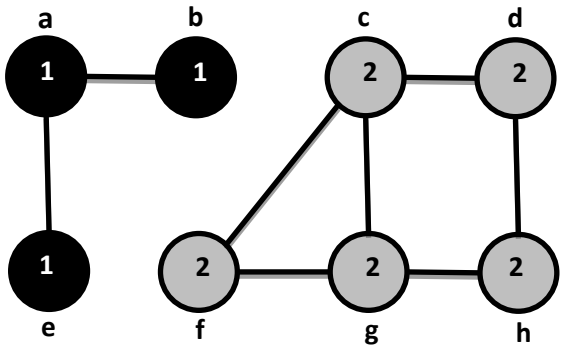
```
BFS-Visit(G, s)

1. Q = {s}
2. while (Q not empty)
3.     u = DE-QUEUE(Q)
4.     for each v ∈ G.Adj[u]
5.       if (color[v] == WHITE)
6.             color[v] = GREY
7.             d[v] = d[u] + 1
8.             ∏[v] = u
9.           EN-QUEUE(Q, v)
10.    color[u] = BLACK
```

Input Example



Its Output



v	a	b	c	d	e	f	g	h
comp[v]	1	1	2	2	1	2	2	2

Answer

∏[v]: is the predecessor of v **d[v]**: distance from root to v **s**: start node
comp[v]: ***index*** of component that contains v [**TO BE FILLED** in your solution]

```
BFS(G, s)
1. For each u ∈ G.V - {s}
2.     color[u] = WHITE
3.     d[u] = ∞
4.     ∏[u] = NIL
5. color[s] = GRAY
6. d[s] = 0
7. ∏[s] = NIL
8. Index = 0
9. For each u ∈ G.V
10.    If (color[u] == WHITE)
11.        Index++
12.        color[u] = GRAY
13.        d[u] = 0
14.        ∏[u] = NIL
15.        BFS-Visit(G, u)
```

```
BFS-Visit(G, s)

1. Q = {s};
2. while (Q not empty)
3.     u = DE-QUEUE(Q);
4.     comp[u] = Index
5.     for each v ∈ G.Adj[u]
6.         if (color[v] == WHITE)
7.             color[v] = GREY;
8.             d[v] = d[u] + 1;
9.             ∏[v] = u;
10.            EN-QUEUE(Q, v);
11.    color[u] = BLACK;
```

QUESTION2: Cycle Detection

Modify the following DFS algorithm to detect (i.e. print) whether there's a cycle in the graph or not? If there's a cycle, print its **start vertex** and **end vertex**? *(ONLY copy code that contains your modification)*

∏[v]: is the predecessor of v **time:** is a global timestamp
d[v]: ***discovery time*** (*v* turns from white to gray) **f[v]**: ***finishing time*** (*v* turns from gray to black)

```
DFS(G)
1. for each vertex u ∈ V[G]
2.     color[u] ← WHITE
3.     π[u] ← NIL
4. time ← 0
5. for each vertex u ∈ V[G]
6.     if color[u] = WHITE then
7.         DFS-Visit(u)
```

```
DFS-Visit(u)

11.    color[u] ← GRAY
12.    time ← time + 1;
13.    d[u] ← time
14.    for each v ∈ Adj[u]
15.        if color[v] = WHITE then
16.            π[v] ← u
17.            DFS-Visit(v)
18.    color[u] ← BLACK
```

	19. $f[u] \leftarrow time \leftarrow time + 1$

Answer

```
DFS-Visit(u)
1.  color[u] ← GRAY
2.  time ← time + 1
3.  d[u] ← time
4.  for each v ∈ Adj[u]
5.        if color[v] = WHITE then
6.            π[v] ← u
7.            DFS-Visit(v)
8.        Else color[v] = GRAY
9.               Print "cycle is detected"
10.              Print "cycle start vertex is" v
11.              Print "cycle end vertex is" u
12.   color[u] ← BLACK
13.   f[u] ← time ← time + 1
```

References

1. MIT 6.006 Introduction to Algorithms, Problem Sessions & Problem Sets, 2020
2. Sphere Online Judge, classical problems
3. Code forces, contest problems
4. FCIS Introduction to algorithms, previous exams