Homework Review

Cross road III - P4067

SAMPLE INPUT:

```
3
2 1
8 3
5 7
```

SAMPLE OUTPUT:

```
15
```

```
7   int n;
8   cin >> n;
9   map<int, vector<int>> mymap; // record time and arrives
10  for(int i = 0; i < n; i++) {
11      int arrive, time;
12      cin >> arrive >> time;
13      if (mymap.count(arrive) == 0) {
14          vector<int> tmp;
15          tmp.push_back(time);
16          mymap[arrive] = tmp;
17      }
18      else {
19          mymap[arrive].push_back(time);
20      }
21  }
22  int total = 0;
23  for(auto it = mymap.begin(); it != mymap.end(); it++) {
24      if (it->first >= total) // if arrival time larger than total
25          total = it->first;
26      for(int i = 0; i < (it->second).size(); i++)
27          total += (it->second)[i];
28  }
29
```

Multiple of N - P1339

Representation	Variables	Code
State	struct node { string number; int remainder; };	
Transition	int digits[M];	y.number += digits[i] + '0'; y.remainder = (y.remainder * 10 + digits[i])%N;
Visited tracking	No need to check visited but should check end status	if (y.remainder == 0)
Distance	No need to check	

```
Sample input:
4999
4
7
6
9
0

Sample output:
60007996
```

Multiple of N - P1339

```
18  struct node {
19    string number;
20    int remainder;
21  };
22
23  using namespace  std;
24
25  int main () {
26    int N, M;
27    queue<node> q;
28    cin >> N >> M;
29    int digits[M];
30    for (int i=0; i<M; i++) {
31      cin >> digits[i];
32    }
33    sort(digits, digits+M);
34    for (int i=0; i<M; i++) {
35      if (digits[i]) // make "4", "6", "7", ...
36        q.push({string(1, digits[i]+'0'), digits[i]%N});
37    }
38
```

```
39    while (!q.empty()) {
40      node x = q.front();
41      q.pop();
42      if (x.number.size() > 500) break;
43      for (int i=0; i<M; i++) {
44        node y = x;
45        y.number += digits[i] + '0';
46        // construct new remander and test it
47        y.remainder = (y.remainder * 10 + digits[i])%N;
48        if (y.remainder == 0) {
49          cout << y.number << endl;
50          return 0;
51        }
52        q.push(y);
53      }
54    }
55    cout << 0 << endl;
56  }
57
```

Livestock Lineup - P15360

Some questions to answer:

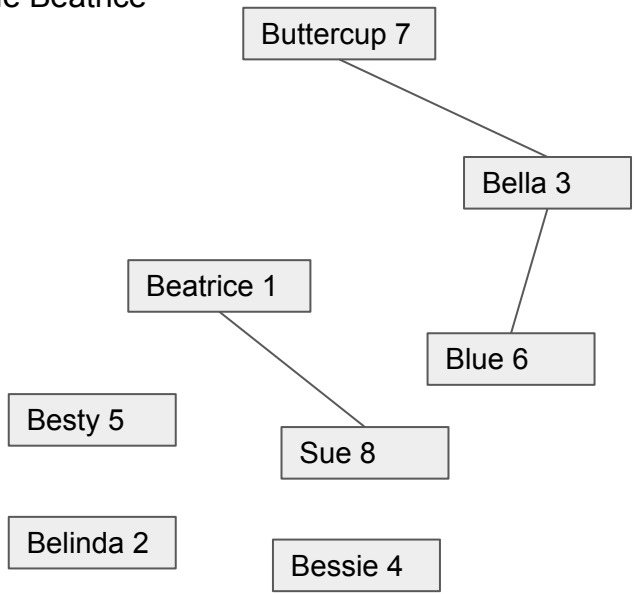
- How do we use graph concept?
- Directed graph or undirected graph?
- How to build data structure?
- Some cows may not be lead, e.g. Bella
- How do we choose lead?
- How do we expand to others?

SAMPLE INPUT:

3
Buttercup must be milked beside Bella
Blue must be milked beside Bella
Sue must be milked beside Beatrice

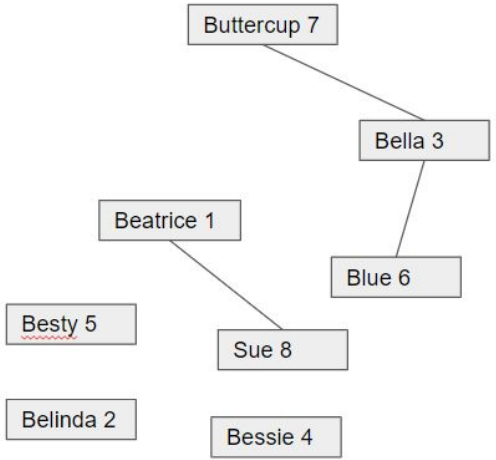
SAMPLE OUTPUT:

Beatrice
Sue
Belinda
Bessie
Betsy
Blue
Bella
Buttercup



Livestock Lineup - P15360

```
6  string cows[] = {
7    "Bessie", "Buttercup", "Belinda", "Beatrice",
8    "Bella", "Blue", "Betsy", "Sue"
9  };
10
11 int main () {
12   int n;
13   string name1, name2, temp;
14   set<string> lead_cows;
15   map<string, bool> visited;
16
17   cin >> n;
18   for (int i=0; i<n; i++) {
19     cin >> name1 >> temp >> temp >> temp >> temp >> name2;
20     adj[name1].push_back(name2);
21     adj[name2].push_back(name1);
22   }
23
24   for (int i=0; i<8; i++) {
25     if (adj[cows[i]].size() != 2) {
26       lead_cows.insert(cows[i]);
27     }
28   }
29
```



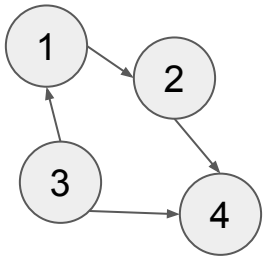

```
30   for (auto x : lead_cows) {
31     if (visited[x]) continue;
32     visited[x] = true;
33     cout << x << endl;
34     if (adj[x].size() == 0) continue; // single cow
35     string y = adj[x][0];
36     while (true) {
37       visited[y] = true;
38       cout << y << endl;
39       if (adj[y].size() > 1)
40         y = (visited[adj[y][0]]) ? adj[y][1] : adj[y][0];
41       else
42         break;
43     }
44   }
45 }
```

Graph Basics

Review - Graph Representation

There are typically two computer representations of graphs:

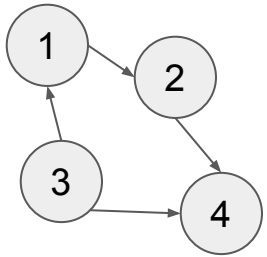
- Adjacency matrix representation
- Adjacency lists representation



Adjacency Matrix

- A square grid of boolean values
- If the graph contains N vertices, then the grid contains N rows and N columns
- For two vertices numbered i and j, the element at row i and column j is true if there is an edge from i to j, otherwise false

	Vert 1	Vert 2	Vert 3	Vert 4
Vert 1	0	1	0	0
Vert 2	0	0	0	1
Vert 3	1	0	0	1
Vert 4	0	0	0	0



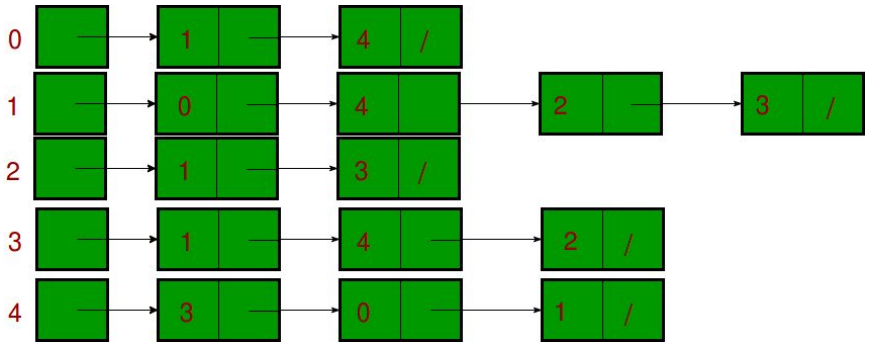
Adjacency Lists Representation

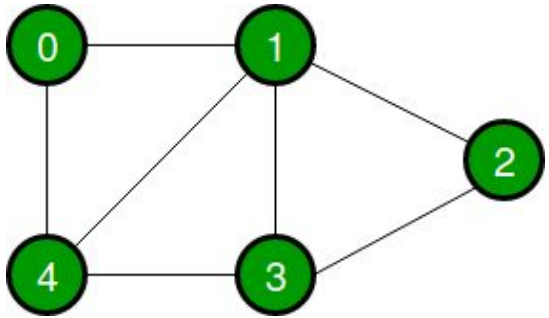
A graph of n nodes is represented by a one dimensional array L of linked lists, where

- L[i] is the linked list containing all the nodes adjacent from node i.
- The nodes in the list L[i] are in no particular order

Example:

- L[0] = {1,4}
- L[1] = {0,4,2,3}
- L[2] = {1,3}
- L[3] = {1,4,2}
- L[4] = {3,0,1}



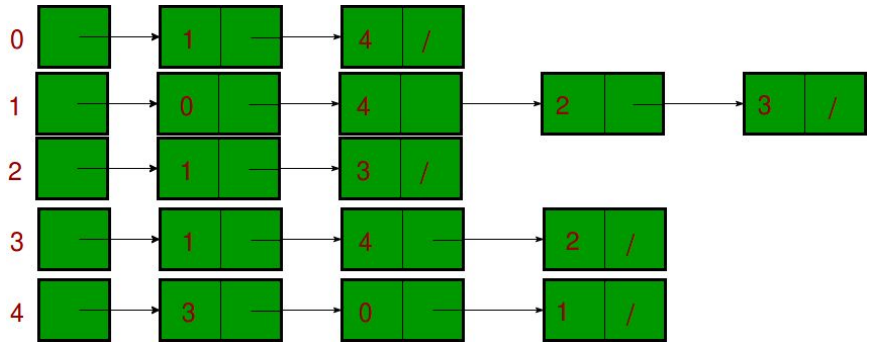


Common implementations: use vector<int> L[]; or list<int> L[];.

Graph Representation

What are pros/cons for the two representations?

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	0
3	1	0	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0



Topics for today:

- Detect a cycle
- Topological sort
- Euler path/circuit

The Great Revegetation - P15373

A lengthy drought has left Farmer John's N pastures devoid of grass. However, with the rainy season arriving soon, the time has come to "revegetate".
In Farmer John's shed, he has four buckets, each with a different type of grass seed. He wishes to sow each pasture with one of these types of seeds. Being a dairy farmer, Farmer John wants to make sure each of his cows has a varied diet. Each of his M cows has two favorite pastures, and he wants to be sure different types of grass are planted in each, so every cow can choose between two types of grass. Farmer John knows that no pasture is a favorite of more than 3 cows.
Please help Farmer John choose a grass type for each pasture so that the nutritional needs of all cows are satisfied.

INPUT FORMAT (file revegetate.in):
The first line of input contains N (2≤N≤100) and M (1≤M≤150). Each of the next M lines contains two integers in the range 1…N, describing the pair of pastures that are the two favorites for one of Farmer John's cows.

OUTPUT FORMAT (file revegetate.out):
Output an N-digit number, with each digit in the range 1…4, describing the grass type to be planted in each field. The first digit corresponds to the grass type for field 1, the second digit to field 2, and so on. If there are multiple valid solutions, print only the N-digit number that is smallest among all of them.

SAMPLE INPUT:
5 6
4 1
4 2
4 3
2 5
1 2
1 5

SAMPLE OUTPUT:
12133

The Great Revegetation - P15373

- Build constraint graph
- Undirected
- Each pasture has 4 options
- Remove options if there is a constraint
- Take the smallest in the left options

SAMPLE INPUT:

5 6
4 1
4 2
4 3
2 5
1 2
1 5

SAMPLE OUTPUT:

12133

DFS Tree on Directed Graph

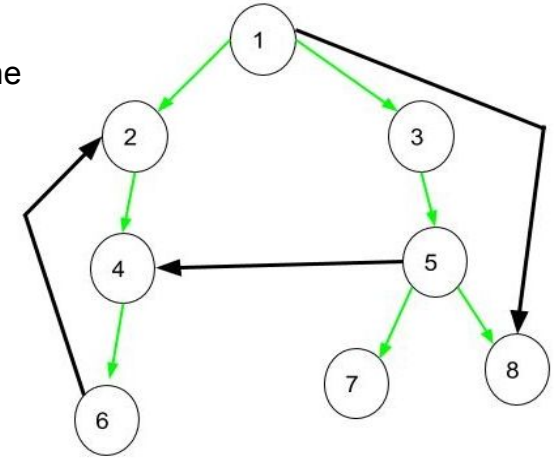
Consider a directed graph, DFS of the below graph is 1 2 4 6 3 5 7 8. Apply DFS on this graph, meanwhile if the vertex are not already in the path, successively add the vertex and the corresponding edge to a tree. DFS tree is connected using green edges.

Tree Edge: It is an edge which is present in the tree obtained after applying DFS on the graph. All the Green edges are tree edges.

Forward Edge: It is an edge (u, v) such that v is descendant of u, but the edge is not part of the DFS tree. Edge from **1 to 8** is a forward edge.

Back edge: It is an edge (u, v) such that v is ancestor of u but the edge is not part of DFS tree. Edge from **6 to 2** is a back edge.

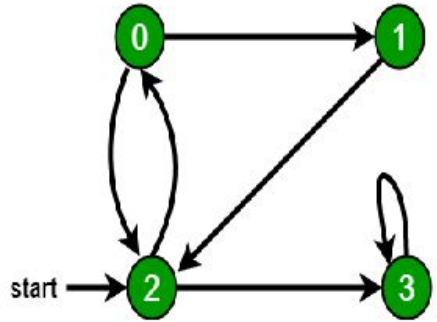
Cross Edge: It is a edge which connects two node such that they do not have any ancestor and a descendant relationship between them. Edge from node **5 to 4** is cross edge.

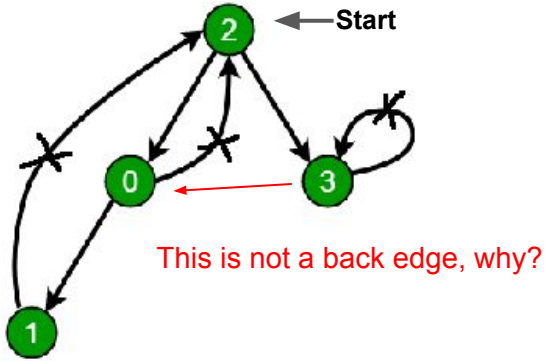


Detect Cycle in a Directed Graph

DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a **back edge** present in the graph. A back edge is an edge that is from a node to **itself** (self-loop) or one of its **ancestors** in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign.

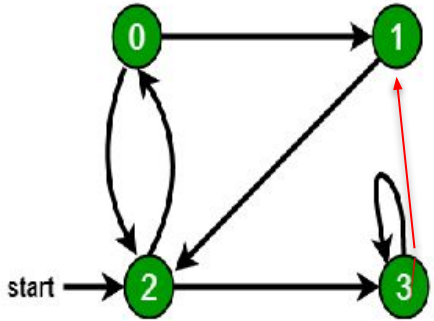
Approach: note that when we recursively reach the current vertex, its ancestors are all in the system stack. We can maintain an recStack[] array to indicate whether a vertex is in the system recursion stack! Go through all the adjacent vertex, check if it's an ancestor of the current vertex.





Detect Cycle in a Directed Graph

```
bool isCyclic(int v, bool visited[], bool recStack[])
{
     // Mark the current node as visited and part of recursion stack
     visited[v] = true;
     recStack[v] = true;
     list<int>::iterator i;
      // Recur for all the vertices adjacent to this vertex
     for(i = adj[v].begin(); i != adj[v].end(); ++i)
     {
         if ( !visited[*i] && isCyclic(*i, visited, recStack) )
             return true;
         else if (recStack[*i]) // can't just check visited, red arrow
             return true;
     }
     recStack[v] = false;  // remove the vertex from recursion stack, back tracking
     return false;
}
```



Detect Cycle in an Simple Undirected Graph

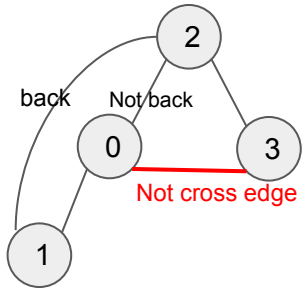
Difference from directed graph in terms of DFS tree:

1. Attn — The edge incident on current vertex and current vertex's parent is not a back edge.
2. Every edge is either a back edge or an DFS tree edge. The cross edge will not exist, e.g. if the red edge exists, the node 3 should be a child of node 0 in the DFS tree, and the edge is not cross edge

Approach: similar to the one in a directed graph.

Special check for (current, parent) edge.

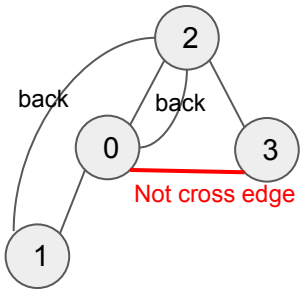
recStack not required because **visited** is enough to identify the back edge.



Start

Detect Cycle in an Simple Undirected Graph

```
// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool isCyclic(int v, bool visited[], int parent)
{
   // Mark the current node as visited
   visited[v] = true;
    // Recur for all the vertices adjacent to this vertex
   list<int>::iterator i;
   for (i = adj[v].begin(); i != adj[v].end(); ++i)
   {
       // If an adjacent is not visited, then recur for that adjacent
       if (!visited[*i])
       {
          if (isCyclic(*i, visited, v))
             return true;
       }
        // If an adjacent is visited and not parent of current vertex,
       // then there is a cycle.
       else if (*i != parent)
          return true;
   }
   return false;
}
```



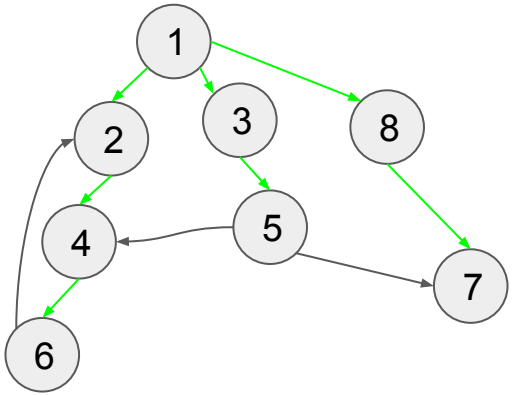
Breadth First Search Tree for a Graph

For simplicity, it is assumed that all vertices are reachable from the starting vertex.

BFS — Start with a vertex, and explores all of the neighbor vertices at the present depth prior to moving on to the vertices at the next depth level. Is_visited array is still required.

if BFS is applied on this graph a tree is obtained which is connected using green edges.

The BFS tree is a shortest path tree starting from its root. Every vertex has a path to the root, with path length equal to its level (just follow the tree itself).

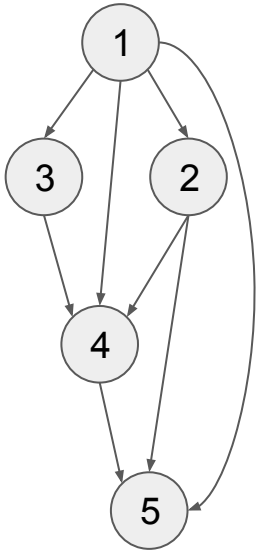


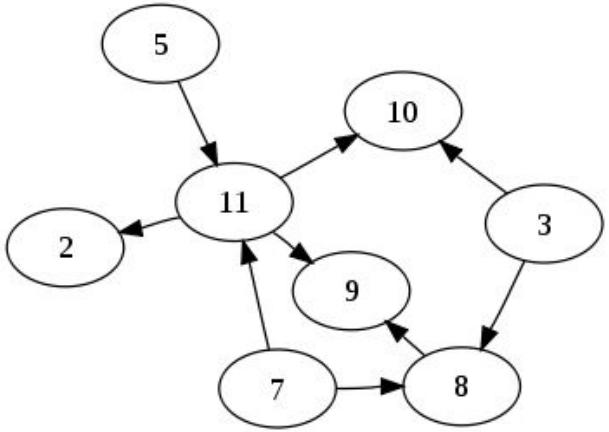
Breadth First Search for a Graph

```
void BFS(int s) {
 bool *visited = new bool[V];
 for(int i = 0; i < V; i++) // Mark all the vertices as not visited
   visited[i] = false;
 list<int> queue; // Create a queue for BFS
 visited[s] = true; // Mark the current node as visited and enqueue it
 queue.push_back(s);
 while(!queue.empty()) {
   // Dequeue a vertex from queue and print it
   s = queue.front();
   cout << s << " ";
   queue.pop_front();
   // Get all adjacent vertices of the dequeued vertex s.
   // If a adjacent has not been visited, then mark it visited and enqueue it
   for (list<int>::iterator i = adj[s].begin(); i != adj[s].end(); ++i) {
     if (!visited[*i]) {
       visited[*i] = true;
       queue.push_back(*i);
     }
   }
 }
}
```

Directed Acyclic Graph (DAG)

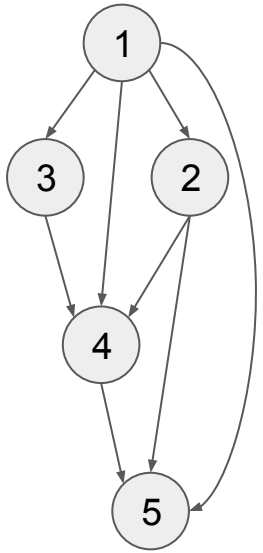
Directed graph with no cycles



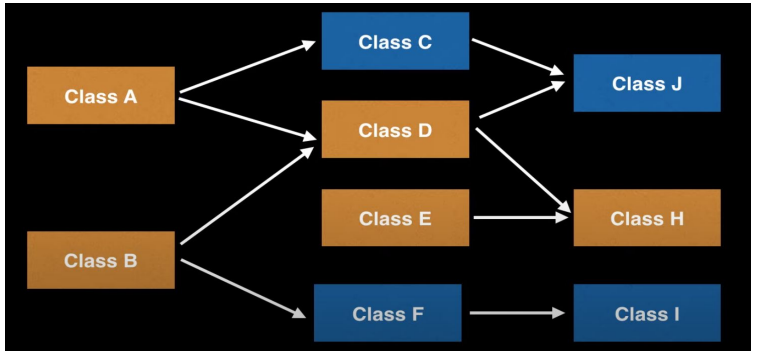


Topological Ordering

Topological Ordering — Every directed acyclic graph has a topological ordering, an ordering of the vertices such that the starting endpoint of every edge occurs earlier in the ordering than the ending endpoint of the edge. The existence of such an ordering can be used to characterize DAGs: a directed graph is a DAG if and only if it has a topological ordering. In general, this ordering is not unique.

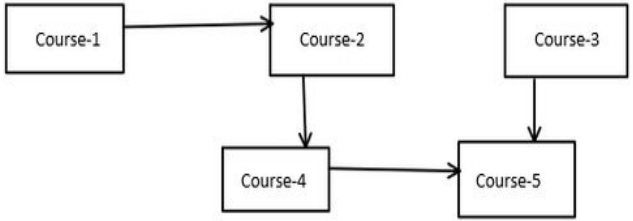


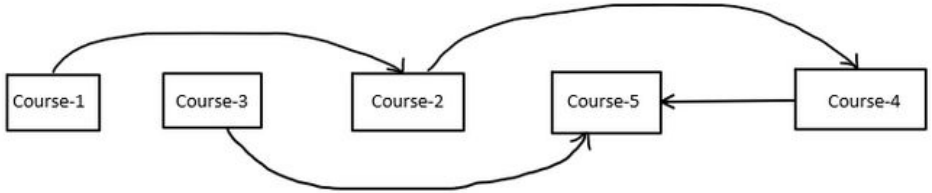
Any examples in life?

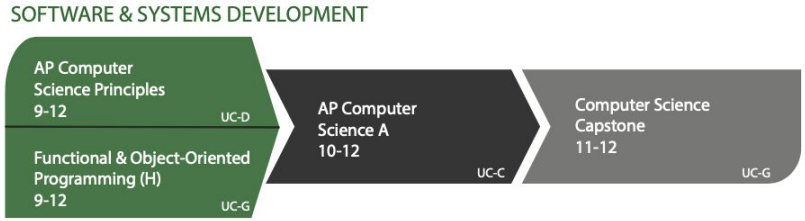


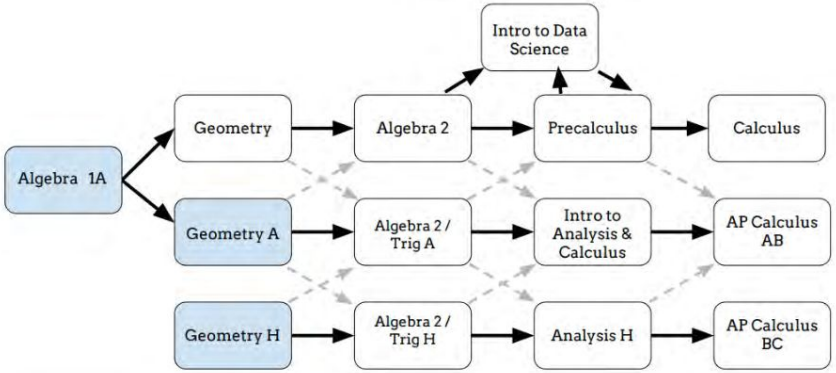
High school course planning

Prerequisites:



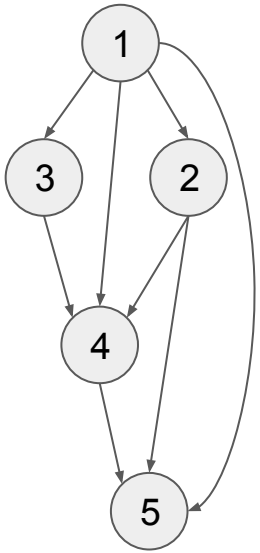


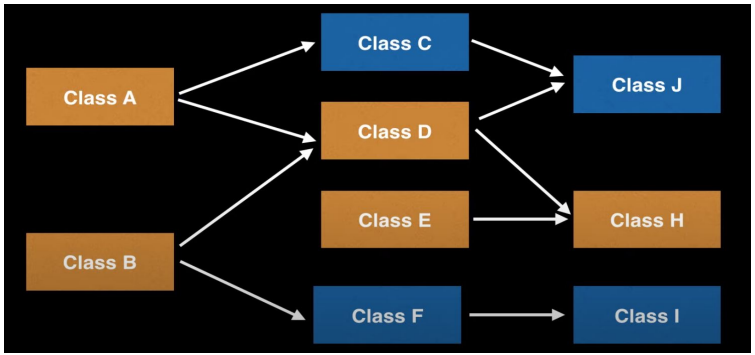




Directed Acyclic Graph (DAG)

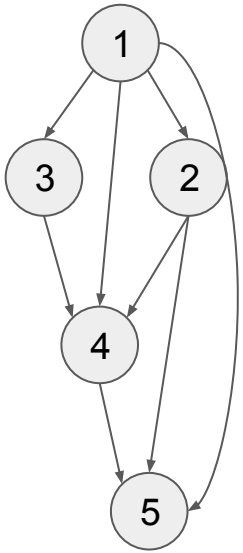
What will be a valid topological order for the two graphs below? Is it unique?

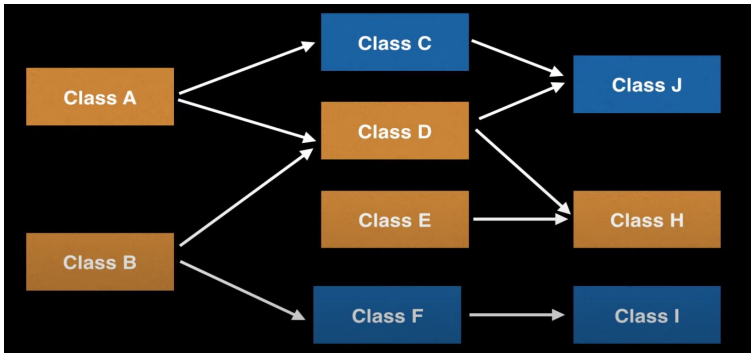




Directed Acyclic Graph (DAG)

How do we generate a topological order based on the graph?
What to consider?





Topological Sorting Algorithm (BFS based)

Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)

Step-3: Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.

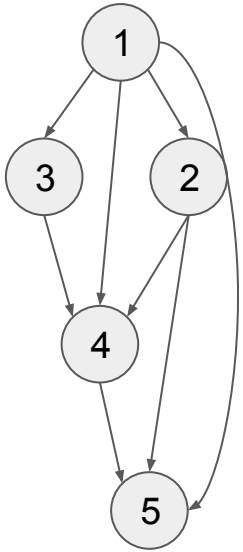
Step 4: Repeat Step 3 until the queue is empty.

If count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph. Why?

Topological Sorting

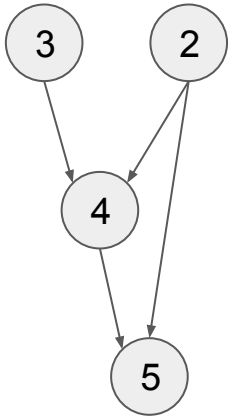
vertex	1	2	3	4	5
In degree	0	1	1	3	3

Queue: [1]



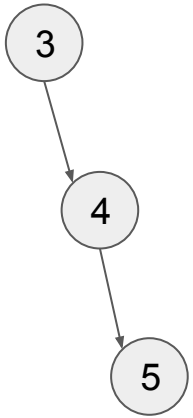
vertex	1	2	3	4	5
In degree	0	0	0	2	2

Queue: [2 3]
Top_order: [1]



vertex	1	2	3	4	5
In degree	0	0	0	1	1

Queue: [3]
Top_order: [1 2]



Topological Sorting

vertex	1	2	3	4	5
In degree	0	0	0	0	1

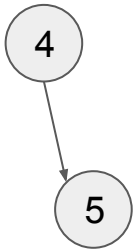
Queue: [4]
Top_order: [1 2 3]

vertex	1	2	3	4	5
In degree	0	0	0	0	0

Queue: [5]
Top_order: [1 2 3 4]

vertex	1	2	3	4	5
In degree	0	0	0	0	0

Queue: []
Top_order: [1 2 3 4 5]

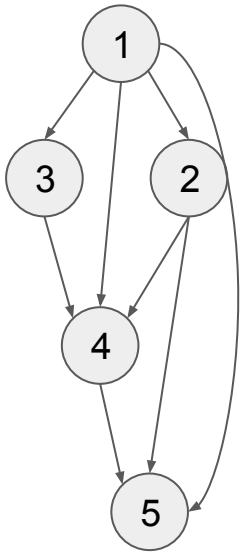


Exercise

https://onlinegdb.com/C_mQzUxnN

Sample input (5 nodes, 8 edges):

```
5 8
1 2
1 3
1 4
1 5
2 4
2 5
3 4
4 5
```

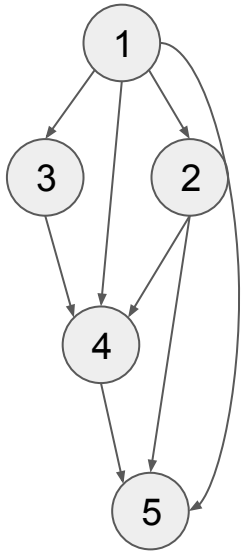


Exercise (Sample code)

https://onlinegdb.com/Luva4CztD

Sample input (5 nodes, 8 edges):

```
5 8
1 2
1 3
1 4
1 5
2 4
2 5
3 4
4 5
```



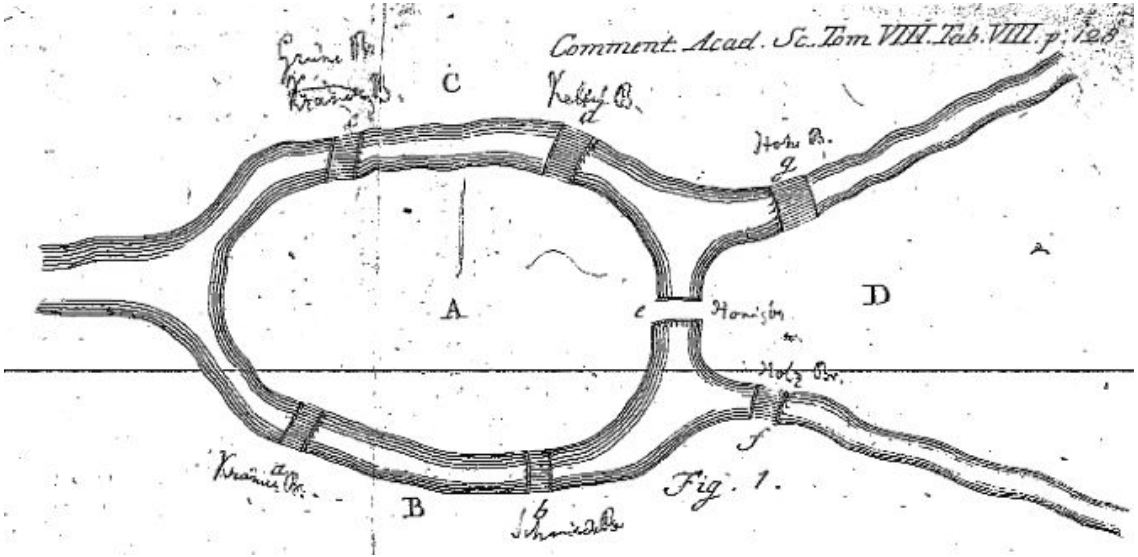
Topological Sorting — if you use list

```
// Create a vector to store in-degrees of all vertices.
vector<int> in_degree(V, 0);
 // Traverse adjacency lists to fill in-degrees of vertices.
for (int u = 0; u < V; u++) {
    list<int>::iterator itr;
    for (itr = adj[u].begin();
         itr != adj[u].end(); itr++)
        in_degree[*itr]++;
}
 // Create an queue and enqueue all vertices with indegree 0
queue<int> q;
for (int i = 0; i < V; i++)
    if (in_degree[i] == 0)
        q.push(i);
 // Initialize count of visited vertices
int cnt = 0;
 // Create a vector to store result
vector<int> top_order;
```

```
// One by one dequeue vertices from queue and
// enqueue adjacents if indegree of adjacent becomes 0
while (!q.empty()) {
     // Extract front of queue and add it to topological order
     int u = q.front();
     q.pop();
     top_order.push_back(u);
     // Iterate through u's neighbouring nodesand decrease their
in-degree by 1.
     list<int>::iterator itr;
     for (itr = adj[u].begin(); itr != adj[u].end(); itr++)
          // If in-degree becomes zero, add it to queue
         if (--in_degree[*itr] == 0) q.push(*itr);
      cnt++;
 }
   // Check if there was a cycle
  if (cnt != V) {
      cout << "There exists a cycle in the graphn";
      return;
  }
```

Euler Path and Euler Circuit

Konigsberg Bridge Problem



Can a tourist cross each of the 7 bridges exactly once?

Swiss mathematician Leonhard Euler proved no in 1735

This is the first theorem in graph theory

https://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem-konigsberg

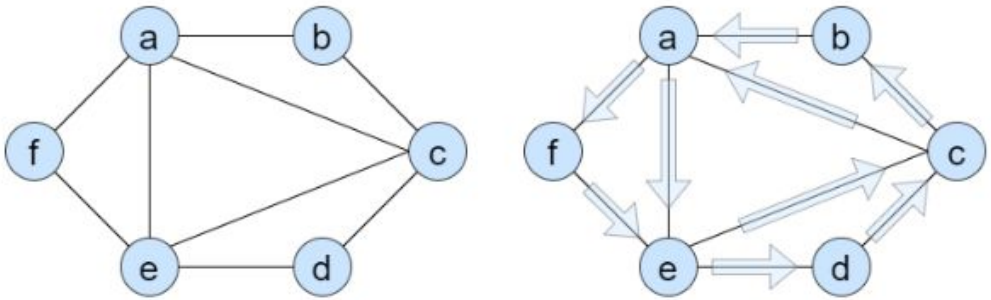
Definition

An **Euler circuit** is a **circuit** starts and ends at the same vertices, and uses every edge of a graph exactly once.

An **Euler path** starts and ends at different vertices.

Euler circuit example

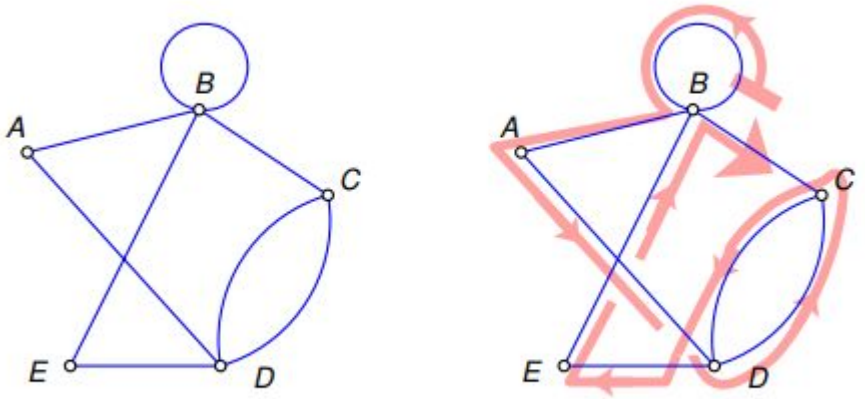
An **Euler circuit** is a **circuit** starts and ends at the same vertices, and uses every edge of a graph exactly once.



Euler Circuit: a-e-c-a-f-e-d-c-b-a

Euler path examples

An **Euler path** is a **path** starts and ends at different vertices, and uses every edge of a graph exactly once.



An Euler path: BBADCDEBC

Euler circuit and Euler path

What kind of graph will have Euler circuit and Euler path?

We know Konigsberg Bridge won’t have a Euler circuit or Euler path. Why and how do we know?

Existence

Existence of Euler Circuit: **Every** vertex has **even** degree.

Existence of Euler Path: Every vertex has even degree except two odd vertices. (They will be the start and end vertex)

Euler circuit and Euler path

How to find Euler circuit and Euler path for a graph?

Algorithm to Find Euler Path and Euler Circuit

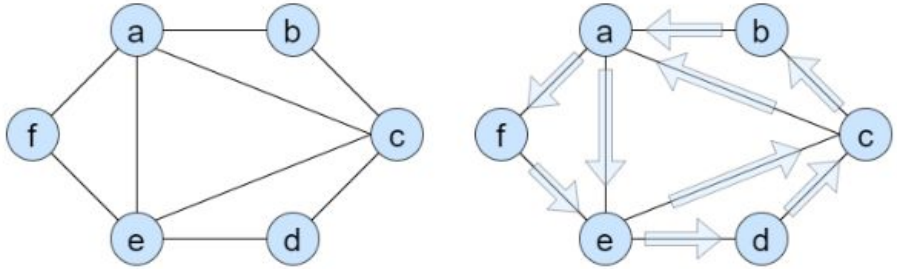
For Euler Circuit, start at any vertex.

Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because indegree and outdegree of every vertex must be same, when the trail enters another vertex w there must be an unused edge leaving w.
The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.

As long as there exists a vertex u that belongs to the current tour, but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and **join** the tour formed in this way to the previous tour.

Euler circuit and Euler path

How to find a Euler circuit for below graph?

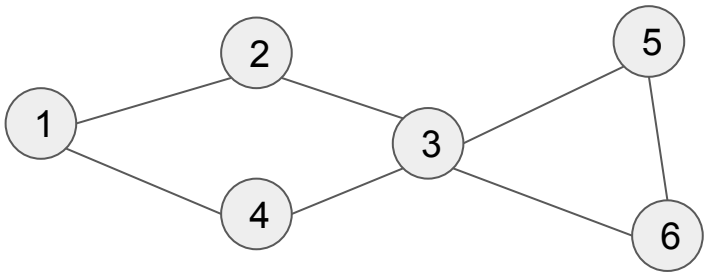


Euler Circuit: a-e-c-a-f-e-d-c-b-a

Euler circuit and Euler path

For Euler circuit, If you start at an even vertex (you are finding Euler Circuit), you must stuck at the same vertex, otherwise it must have one more edge going out.

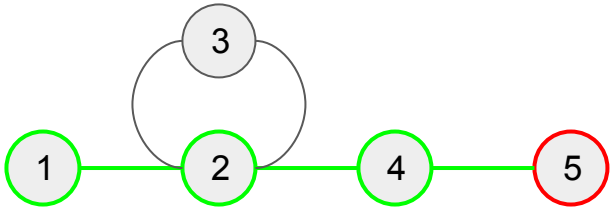
As long as there exists a vertex u that belongs to the current tour, but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.



Algorithm to Find Euler Path and Euler Circuit

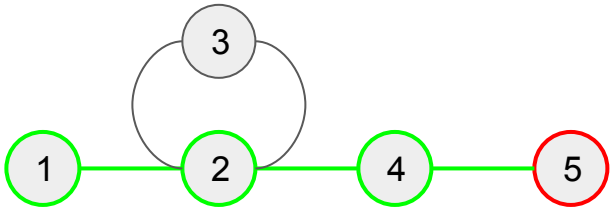
For Euler Path, start at one of the odd vertex.

1. DFS, Keep following unused edges and removing them until we get stuck.
2. Once we get stuck, we backtrack to the nearest vertex in our current path that has unused edges, and we repeat the process until all the edges have been used. We can use another container to maintain the final path



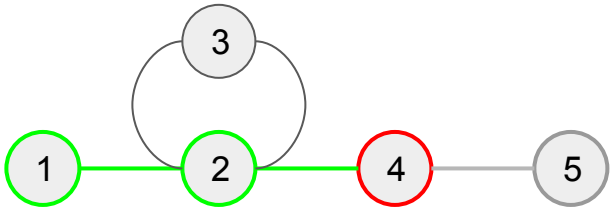
Suppose we start from node 1

Stuck at 5



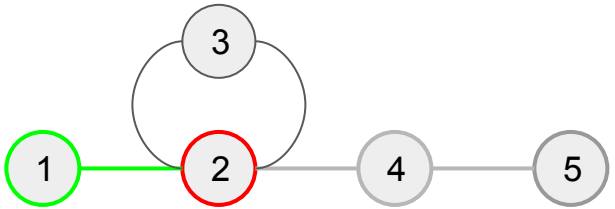
Backtrack to 4, Added 5 to the front of solution

Solution: [5]



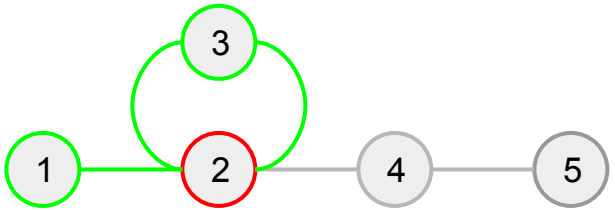
Backtrack to 2, Added 4 to the front of solution

Solution: [4, 5]



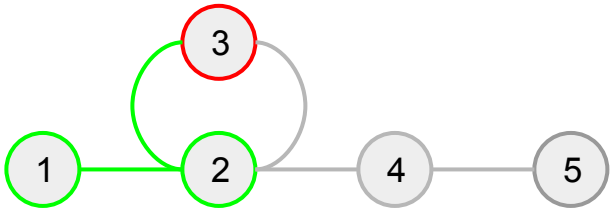
Visited 3, Stuck at 2 again

Solution: [4, 5]



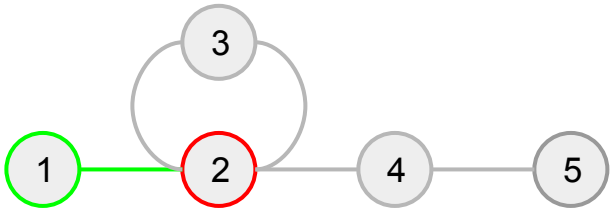
Backtrack to 3, Added 2 to the front of solution

Solution: [2, 4, 5]



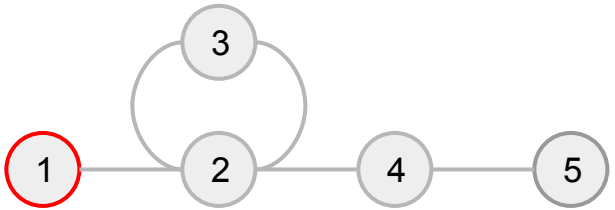
Backtrack to 2, Added 3 to the front of solution

Solution: [3, 2, 4, 5]



Backtrack to 1, Added 2 to the front of solution

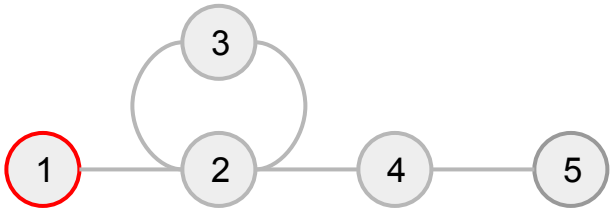
Solution: [2, 3, 2, 4, 5]



Final Solution: [1, 2, 3, 2, 4, 5]

Backtrack to 1, Added 2 to the front of solution

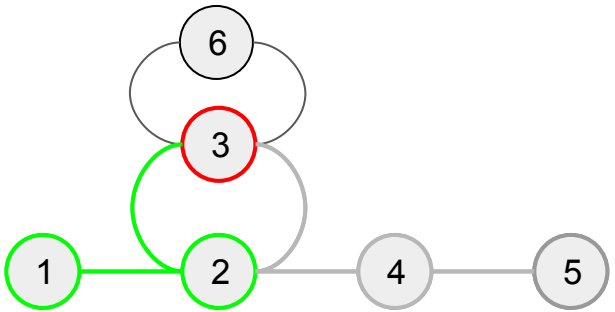
Solution: [2, 3, 2, 4, 5]



Idea:
If you start at the odd vertex (for Euler path), you must stuck at the other odd vertex (otherwise it must have one more edge going out).

Final Solution: [1, 2, 3, 2, 4, 5]

How the algorithm works if there is one more vertex

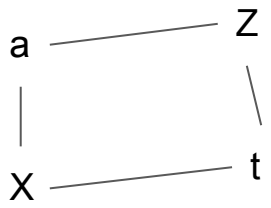


It’s like insert [3, 6, 3] in the middle of [1, 2, 3, 2, 4, 5] => [1, 2, 3, 6, 3, 2, 4, 5].

When we backtrack to 3, we will go up to 6 by an unused edge and go down to 3 again. Then backtrack.

[2, 4, 5] => [3, 2, 4, 5] => [6, 3, 2, 4, 5] => [3, 6, 3, 2, 4, 5] => ….

Unordered letter pairs (P8264)

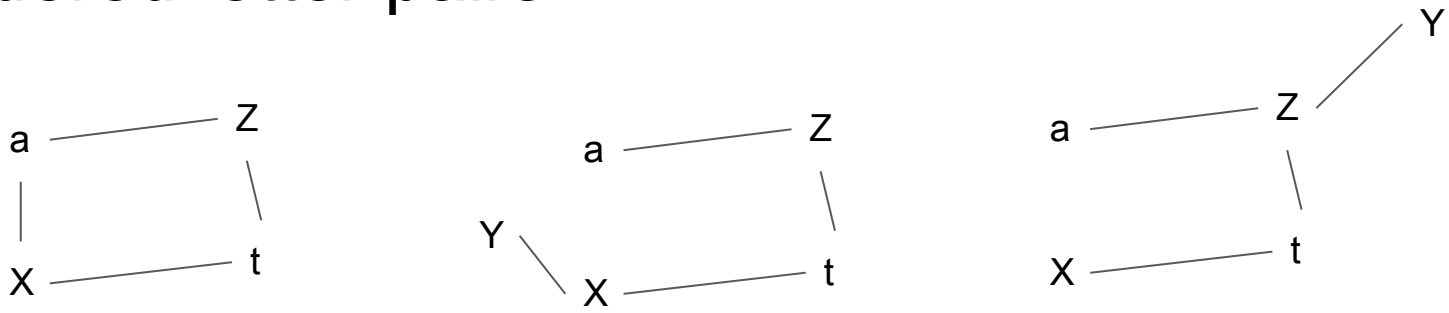


1. We have N connections (edges) and want to build a string with size of N+1
2. It is possible that no solution exists
3. What graph will have solution and what graph won't have solution?

Sample input 1:
4
aZ
tZ
Xt
aX

Sample output 2:
XaZtX

Unordered letter pairs



1. What graph will have solution and what graph won't have solution?
2. This is to determine existence of Euler circuit or Euler path
3. Once we know there is solution, we need to find the smallest string
4. What will be smallest string for above valid graphs?
5. Select the right starting node to make it smallest
6. Once we have the starting node, we can finish the path