ADJACENCY LIST FOR GRAPH REPRESENTATION

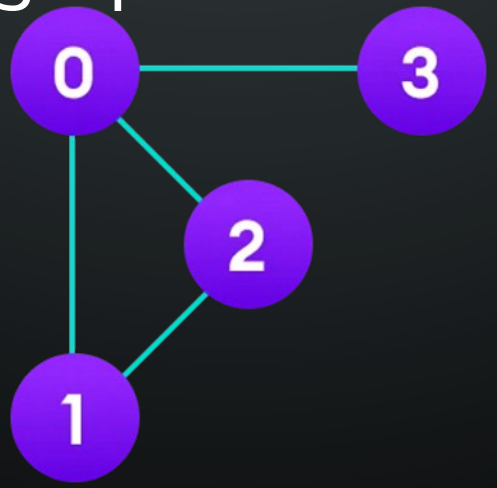
- UNDIRECTED GRAPH REPRESENTATION
- DIRECTED GRAPH REPRESENTATION
- WEIGHTED GRAPH REPRESENTATION
- ADVANTAGES OF THE ADJACENCY LIST
- DISADVANTAGES OF THE ADJACENCY LIST
- THE ADJACENCY LIST STRUCTURE
- PRACTICE: TASK SEQUENCE PROBLEM

Author: prof. Yevhenii Borodavka

UNDIRECTED GRAPH REPRESENTATION

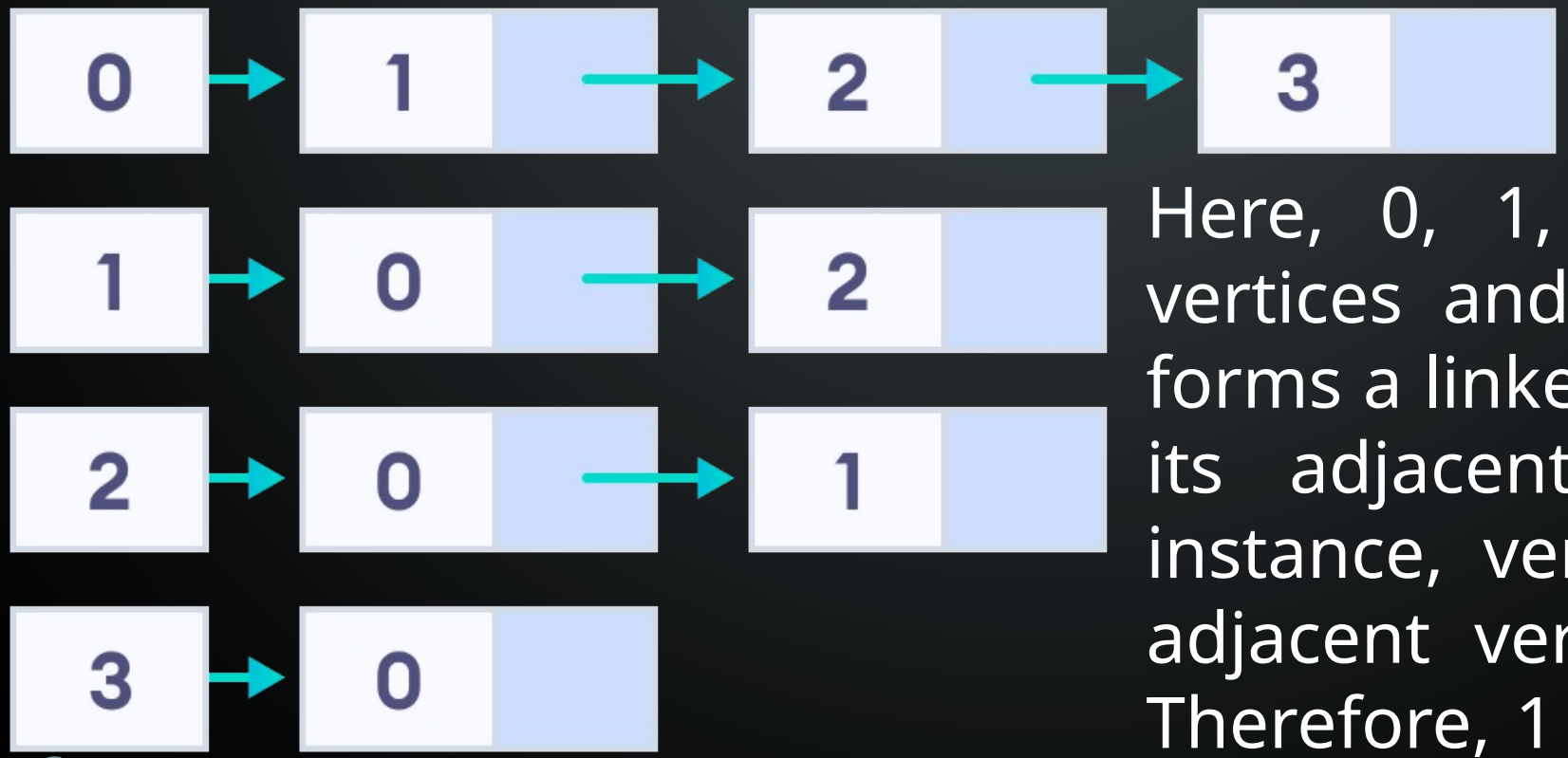
An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

For example, we have a graph below.



UNDIRECTED GRAPH REPRESENTATION

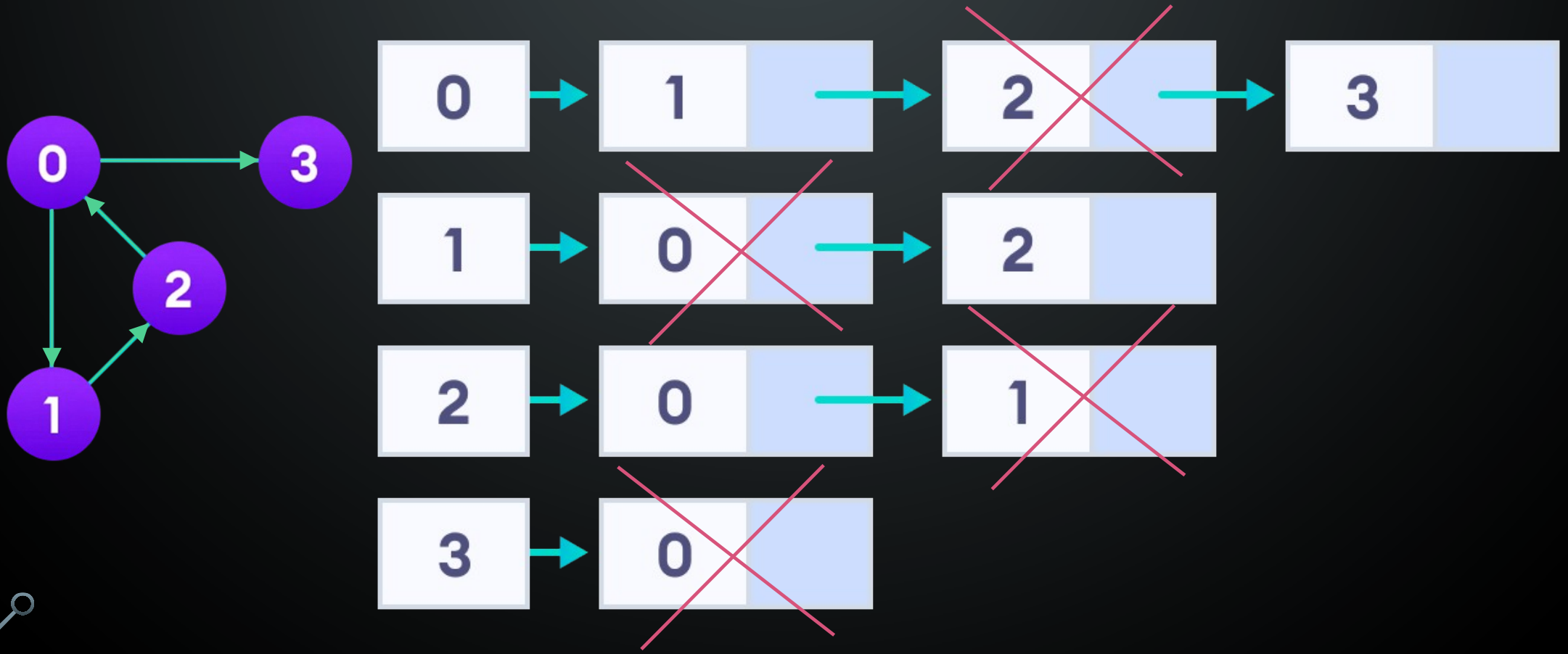
We can represent this graph in the form of a linked list on a computer as shown below.



Here, 0, 1, 2, 3 are the vertices and each of them forms a linked list with all of its adjacent vertices. For instance, vertex 1 has two adjacent vertices 0 and 2. Therefore, 1 is linked with 0 and 2 in the figure above.

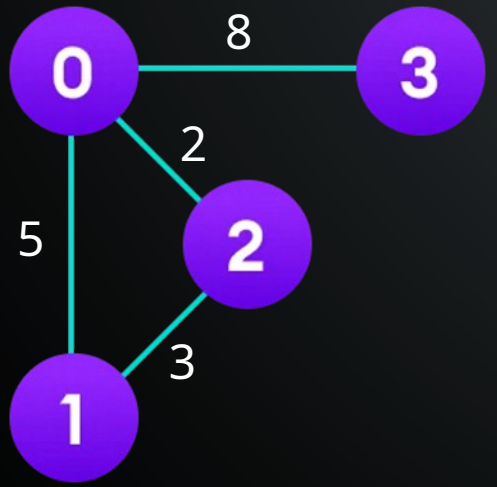
DIRECTED GRAPH REPRESENTATION

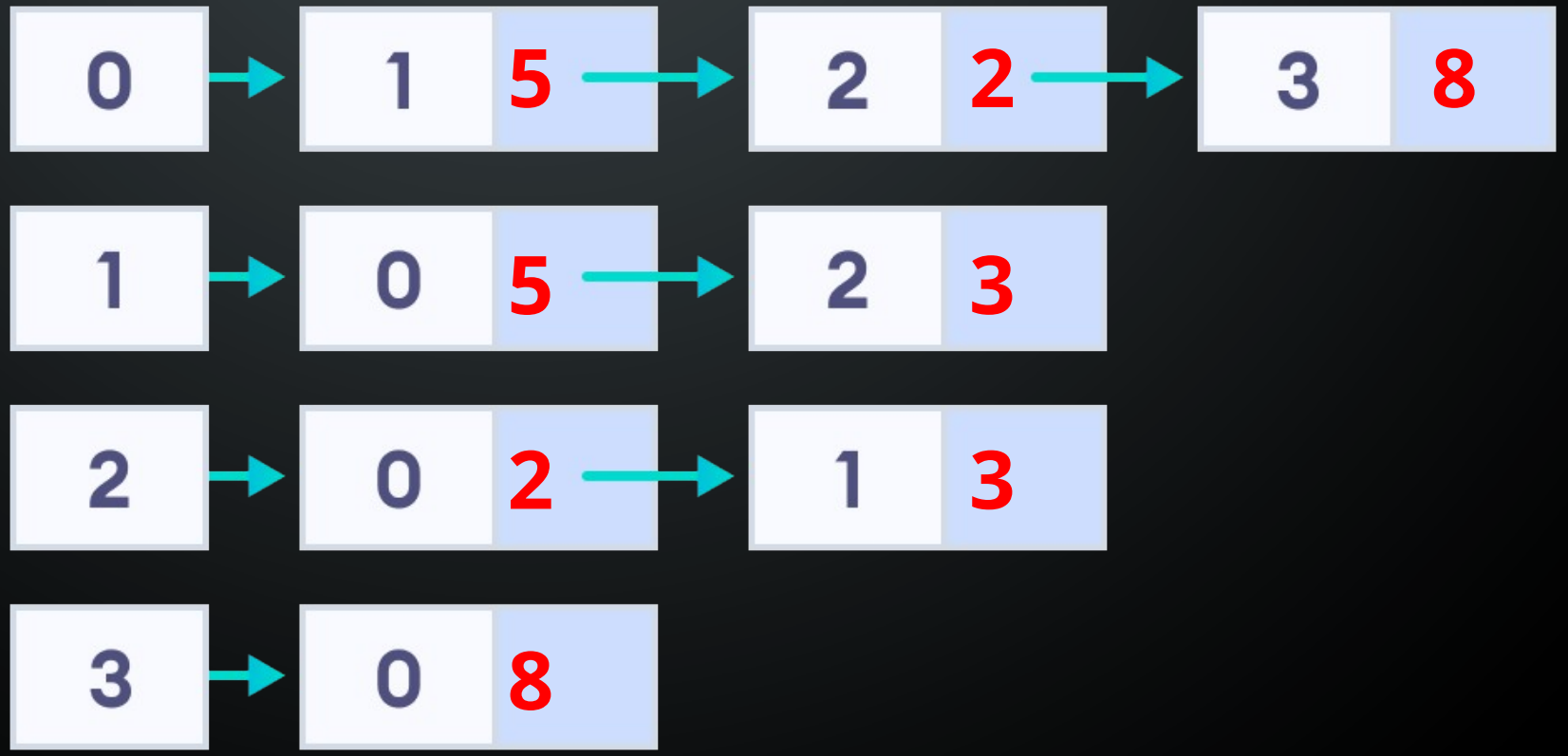
For directed graphs, the adjacency list is shorter, because only one direction is used.



WEIGHTED GRAPH REPRESENTATION

A weighted graph can be represented using the weight of the edge as additional information to the node structure.





ADVANTAGES OF THE ADJACENCY LIST

- A list of adjacencies saves a lot of space.
- We can easily insert and delete items because we are using a linked list.
- Such a representation is simple to understand and clearly shows the adjacent nodes.
- An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a sparse graph with millions of vertices and edges, this can mean a lot of saved space.
- It also helps to find all the vertices adjacent to a vertex easily.

DISADVANTAGES OF THE ADJACENCY LIST

- Finding the adjacent list is not quicker than the adjacency matrix because all the connected nodes must be first explored to find them.
- While the adjacency list allows you to test whether two vertices are adjacent, it is slower to perform this operation.

THE ADJACENCY LIST STRUCTURE

The simplest adjacency list needs a node data structure to store a vertex and a graph data structure to organize the nodes.

We stay close to the basic definition of a graph – a collection of vertices and edges {**V**, **E**}. For simplicity, we use an unlabeled graph as opposed to a labeled one i.e. the vertices are identified by their indices 0,1,2,3.

Let's dig into the data structures at play here.

THE ADJACENCY LIST STRUCTURE

```
struct node{
    int vertex;
    struct node* next;
};
struct Graph{
    int numVertices;
    struct node** adjLists;
};
```

THE ADJACENCY LIST STRUCTURE

Don't let the struct node** adjLists overwhelm you.

All we are saying is we want to store a pointer to struct node*. This is because we don't know how many vertices the graph will have and so we cannot create an array of Linked Lists at compile time.

For the weighted graph, the node structure needs to contain the weight of the edge as well.

```
struct node{
    int vertex;
    int weight;
    struct node* next; };
```

PRACTICE: TASK SEQUENCE PROBLEM

Problem. You have **N** tasks ($1<=N<=10^5$) you need to do. You also have a **K** pair of the tasks ($1<=K<=10^5$) which describes the task dependency – the second task must be done after the first one. You need to sort the tasks in the correct order for execution.

Example. You have **N**=9 tasks and **K**=9 pairs: (4,1) (1,2) (2,3) (2,7) (5,6) (7,6) (1,5) (8,5) (8,9). Correct orders are 8,9,4,1,5,2,3,7,6 or 4,1,2,3,8,5,7,6,9 or 4,8,9,1,2,3,5,7,6. But this order 4,1,5,2,3,7,6,8,9 is incorrect because the task 5 can't be done before the task 8.

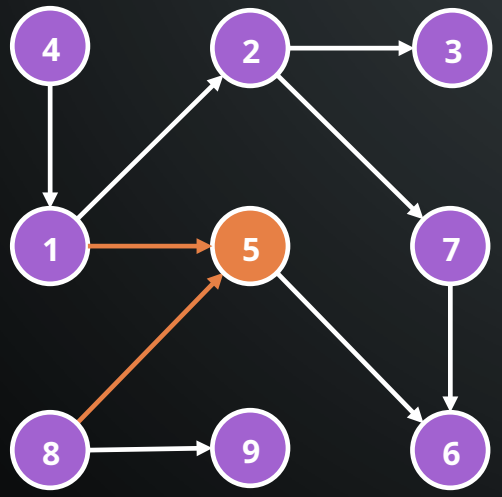
Task. Create a program using C/C++/Python to solve this problem.

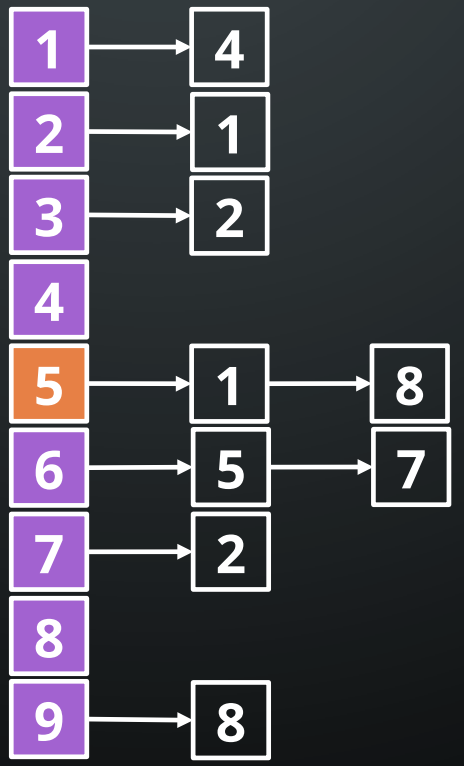
Input. Two strings: the first one with two numbers **N** and **K** divided by space and the second one with **K** pairs of the tasks indexes divided by spaces.

Output. The correct order of the task execution (tasks indexes divided by

PRACTICE: TASK SEQUENCE PROBLEM

Graph from example and its adjacency list. Each node contains a vertex index, visited value, and reference on the previous node.

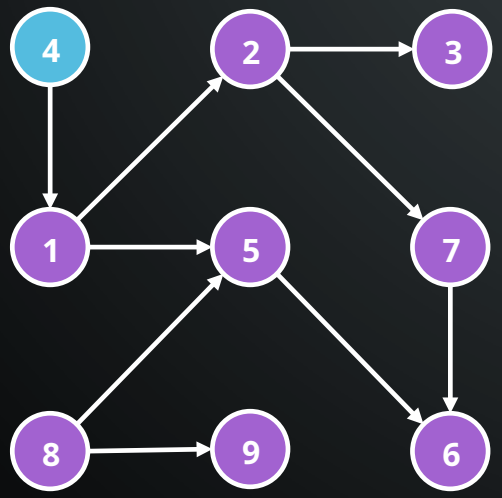


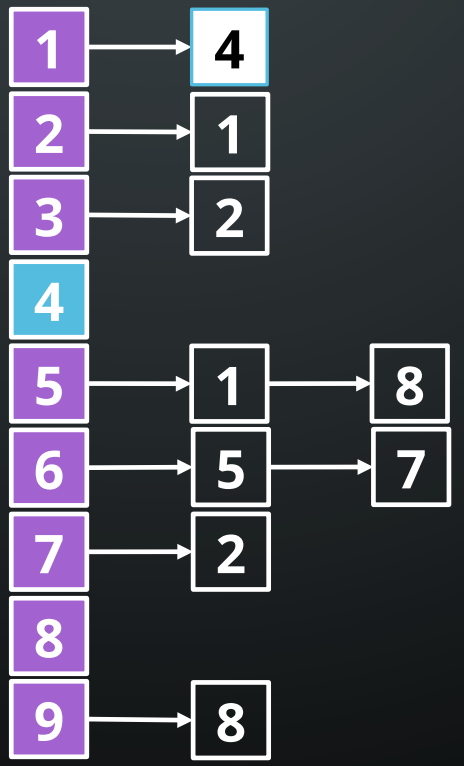


The **A** is output result.

PRACTICE: TASK SEQUENCE PROBLEM

Traverse the adjacency list **N** times. Step **1**.

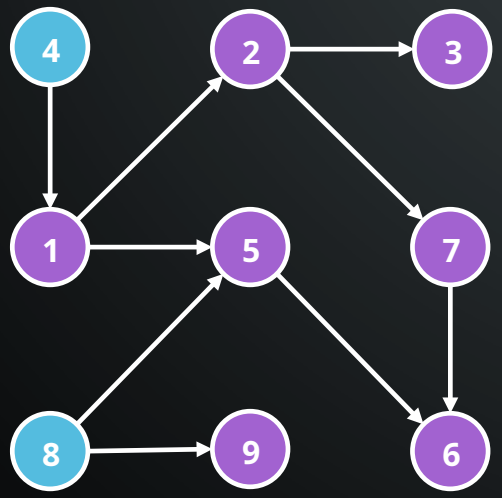


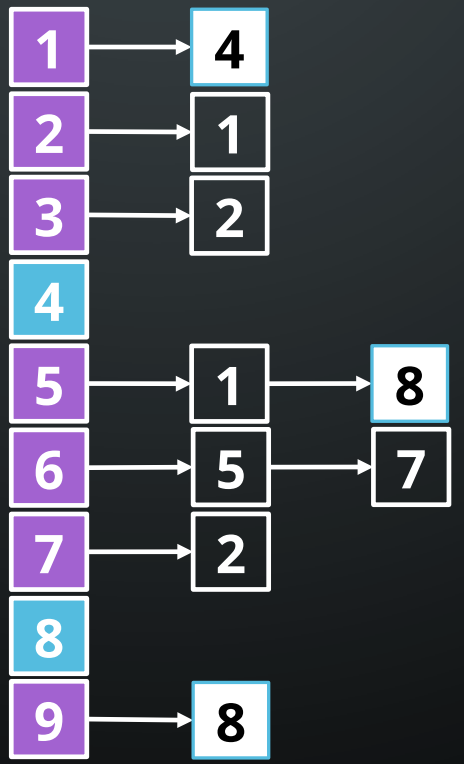


The **A** is output result.

PRACTICE: TASK SEQUENCE PROBLEM

Traverse the adjacency list **N** times. Step **2**.



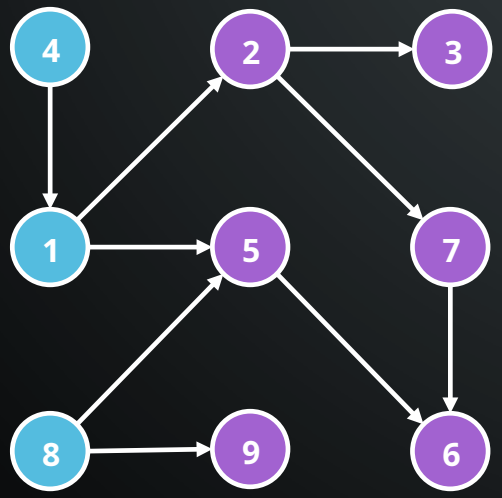


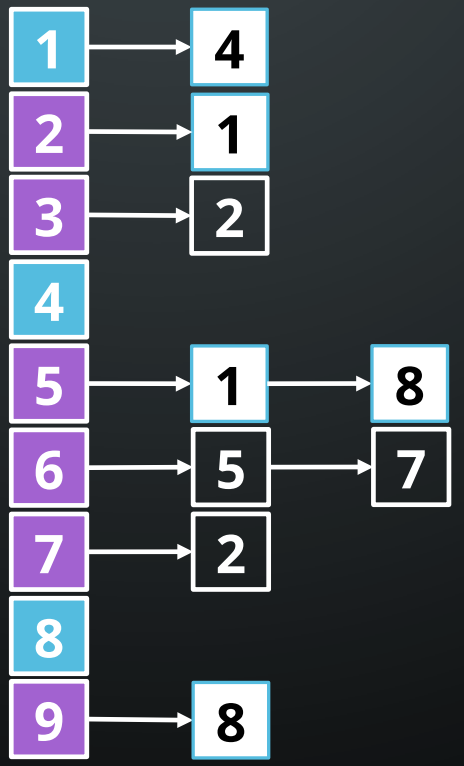
The **A** is output result.

A	4	8							

PRACTICE: TASK SEQUENCE PROBLEM

Traverse the adjacency list **N** times. Step **3**.



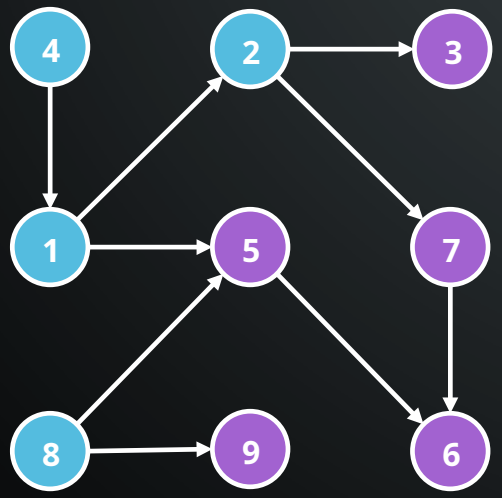


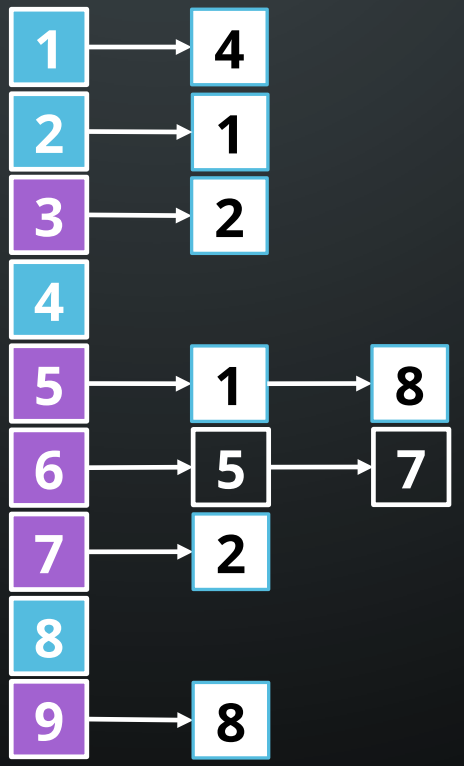
The **A** is output result.

A	4	8	1						

PRACTICE: TASK SEQUENCE PROBLEM

Traverse the adjacency list **N** times. Step **4**.



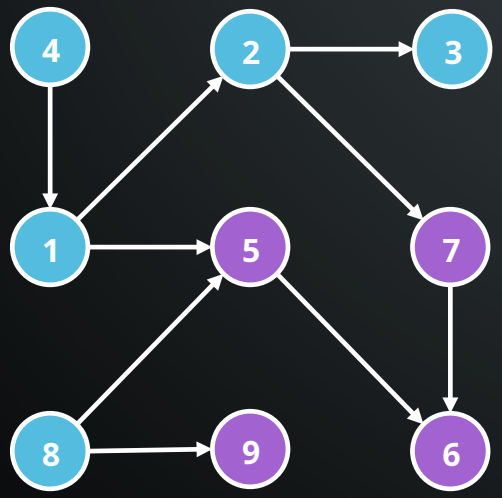


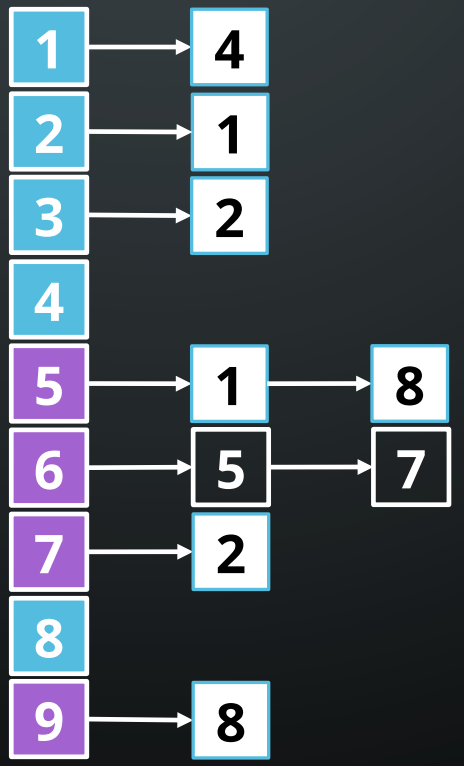
The **A** is output result.

A	4	8	1	2					

PRACTICE: TASK SEQUENCE PROBLEM

Traverse the adjacency list **N** times. Step **5**.



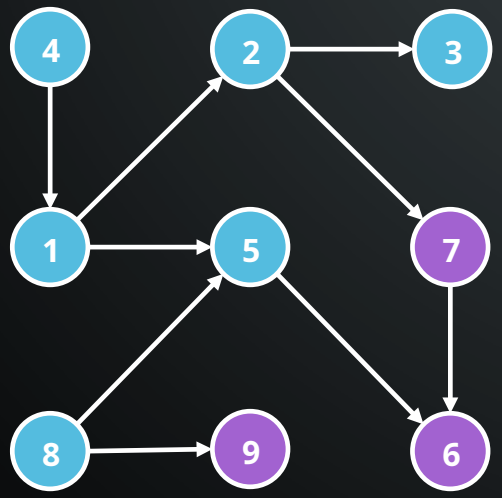


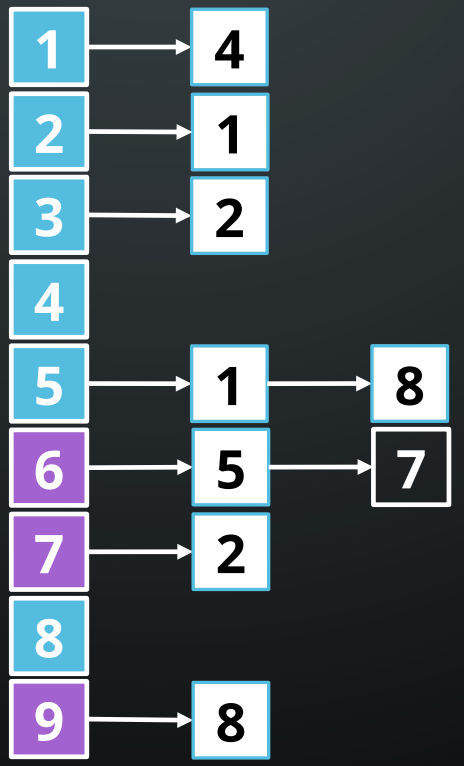
The **A** is output result.

A	4	8	1	2	3				

PRACTICE: TASK SEQUENCE PROBLEM

Traverse the adjacency list **N** times. Step **6**.



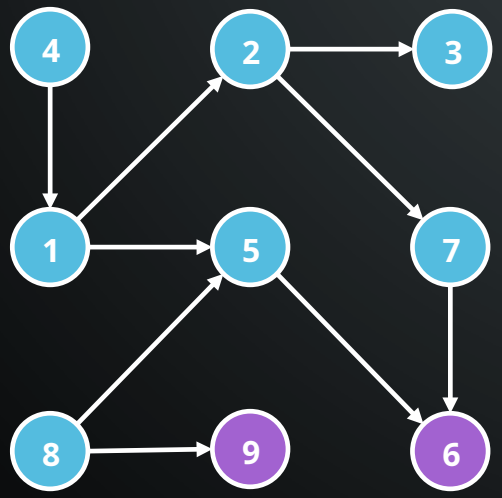


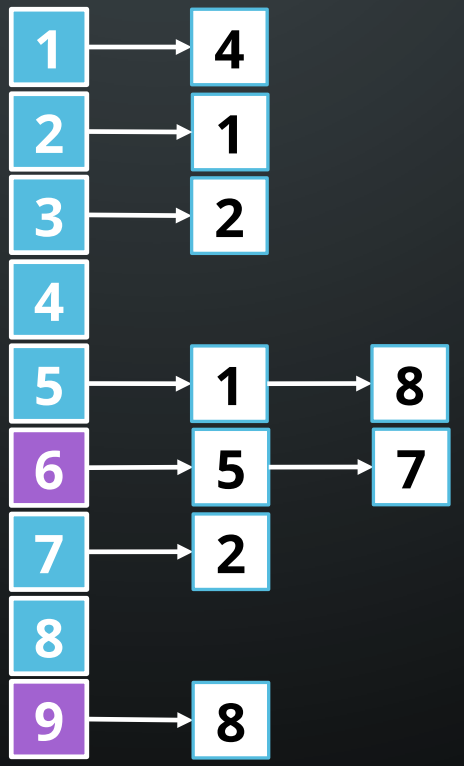
The **A** is output result.

A	4	8	1	2	3	5			

PRACTICE: TASK SEQUENCE PROBLEM

Traverse the adjacency list **N** times. Step **7**.



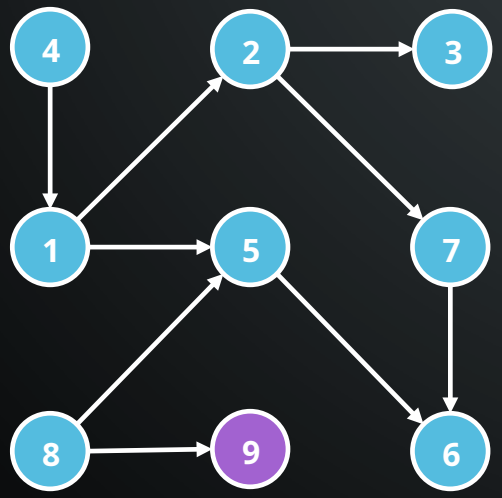


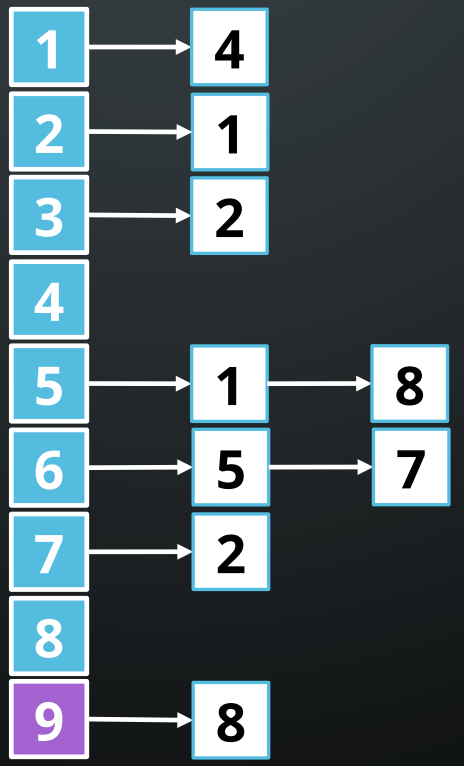
The **A** is output result.

A	4	8	1	2	3	5	7		

PRACTICE: TASK SEQUENCE PROBLEM

Traverse the adjacency list **N** times. Step **8**.



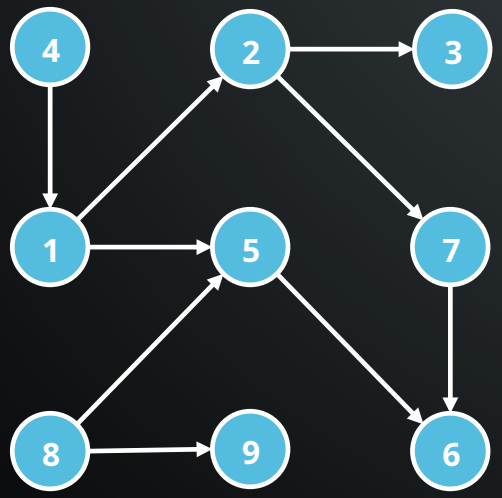


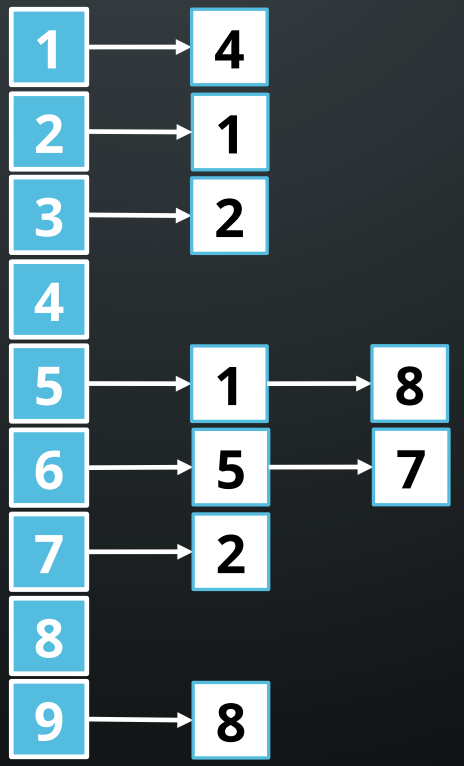
The **A** is output result.

A	4	8	1	2	3	5	7	6	

PRACTICE: TASK SEQUENCE PROBLEM

Traverse the adjacency list **N** times. Step **9**.





The **A** is output result.

A	4	8	1	2	3	5	7	6	9

PRACTICE: TASK SEQUENCE PROBLEM

Code example:

```
#include <iostream>
#define MAX 100001
typedef struct _node_t { int index; bool visited; struct _node_t* previous; } node_t;
int N, K;
node_t Graph[MAX];
void add_node(int curr, int prev) { /*your code here*/ };
int main()
{
    cin >> N >> K;
    for(int i = 1; i <= N; i++) Graph[i].index = i;
    int f, s;
    for(int i = 0; i < K; i++)
    {
        cin >> f >> s;
        add_node(s, f);
    }
    for(int i = 1; i <= N; i++)
    {   /*your code here*/
    }
    return 0;
}
```

THANK YOU!