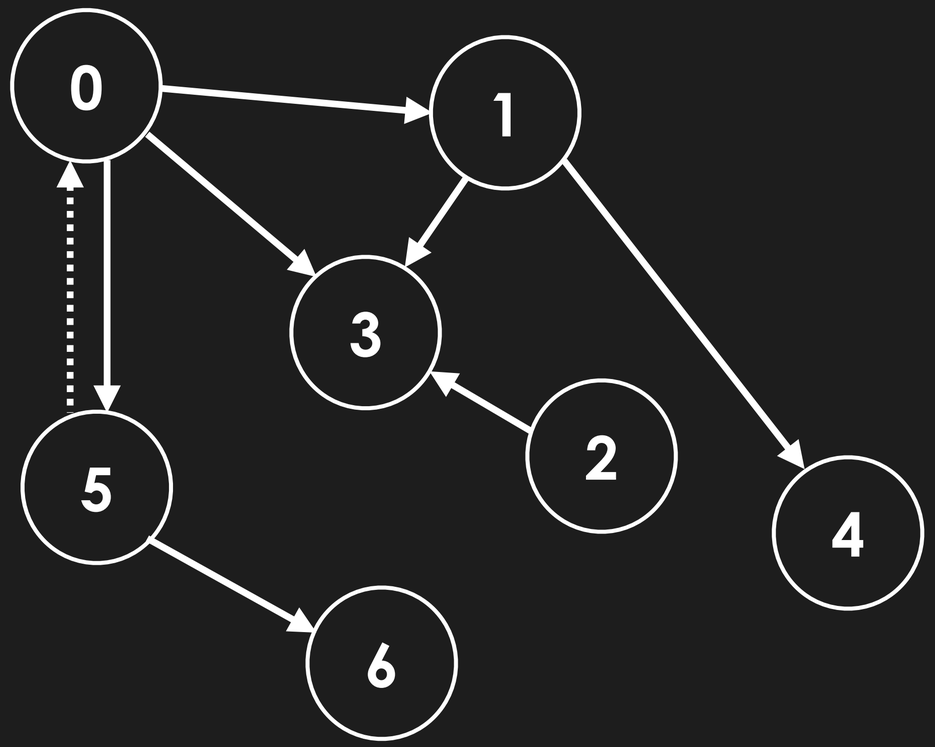

C
CodeIT

# Directed Graphs

# Directed Graphs

Directed Graphs are graphs where edges have a direction

# Example



# Implementation of Digraph

# DirectedEdge

```
class DirectedEdge:
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest
```

# DirectedEdge

```
class DirectedEdge:
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest
```

Creating a class for our edge makes things easier. This will come in esp handy when dealing with weighted graphs!

# Digraph

```
class Digraph:
    def __init__(self, V):
        self.adjList = [[] for i in range(V)]

    def addEdge(self, edge):
        src, dest = edge
        self.adjList[src].append(DirectedEdge(src, dest))
```

# Digraph

```
class Digraph:
    def __init__(self, V):
        self.adjList = [[] for i in range(V)]

    def addEdge(self, edge):
        src, dest = edge
        self.adjList[src].append(DirectedEdge(src, dest))
```

**Note how when we add an edge, we only add it to src's list**

# printGraph

```
def printGraph(self):
    for v in range(len(self.adjList)):
        print("vertex {}: ".format(v), end="")
        for edge in self.adjList[v]:
            print("({}, {})".format(edge.src, edge.dest), end=" ")
        print()
```

# Digraph Demo

```
V = 7
digraph = Digraph(V)
edges = [(0, 1), (1, 4), (1, 3), (2, 3), (0, 3), (0, 5), (5, 0), (5, 6)]
for edge in edges:
    digraph.addEdge(edge)

digraph.printGraph()
```

```
V = 7
digraph = Digraph(V)
edges = [(0, 1), (1, 4), (1, 3), (2, 3), (0, 3), (0, 5), (5, 0), (5, 6)]
for edge in edges:
    digraph.addEdge(edge)

digraph.printGraph()
```

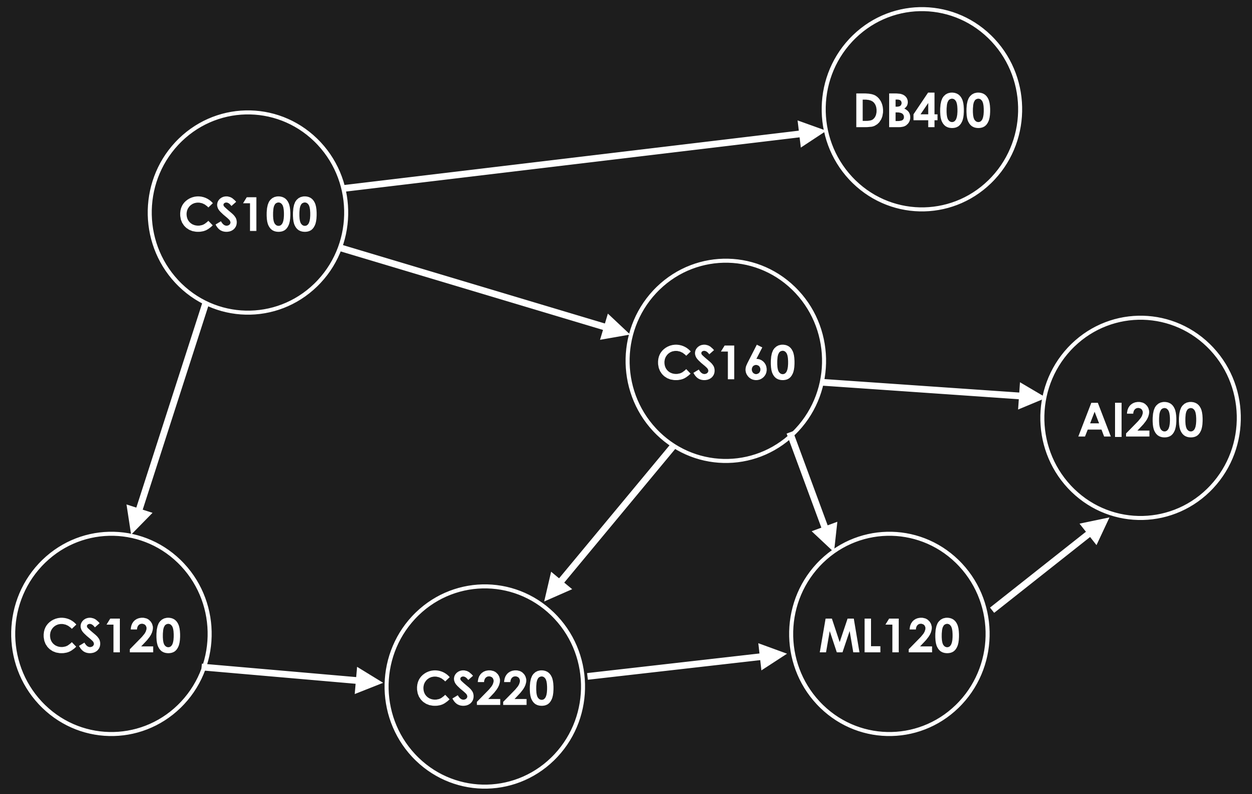
```
vertex 0: (0, 1) (0, 3) (0, 5)
vertex 1: (1, 4) (1, 3)
vertex 2: (2, 3)
vertex 3:
vertex 4:
vertex 5: (5, 0) (5, 6)
vertex 6:
```

# Topological Sort

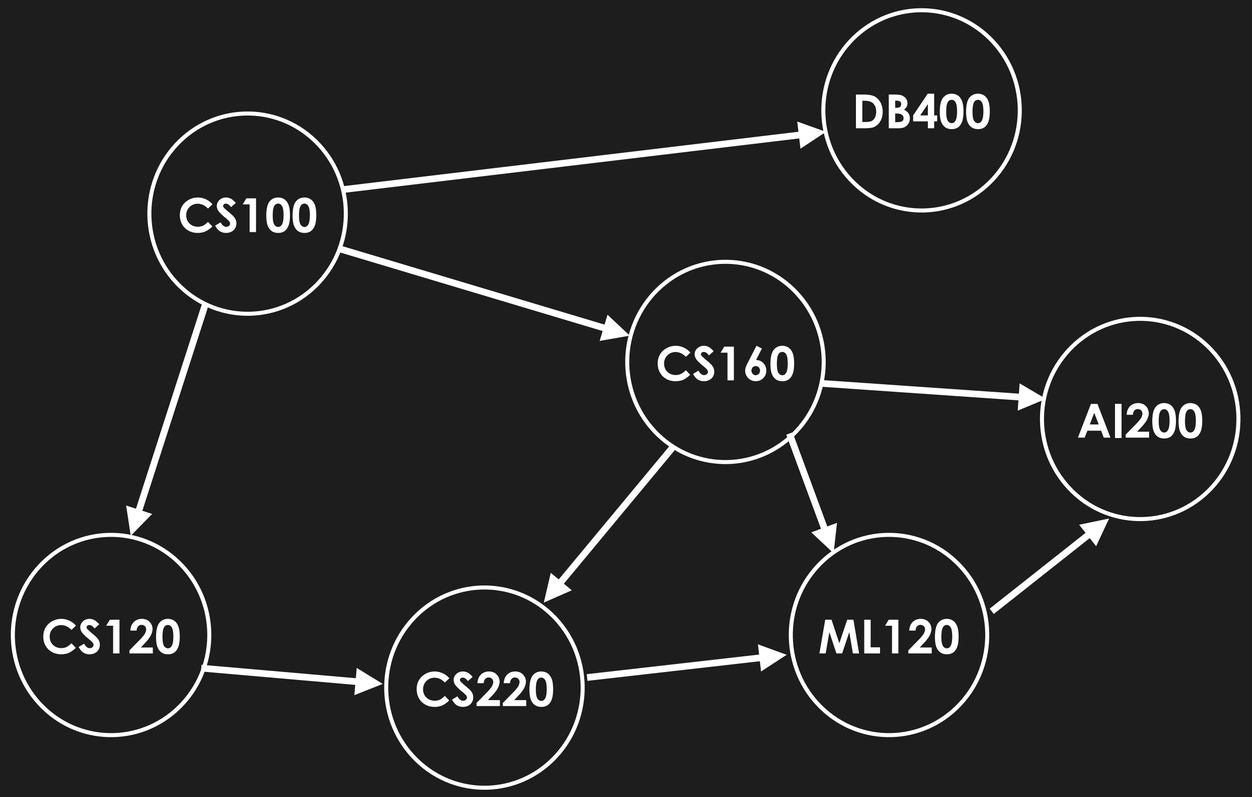
# Topological Sort

Topological sort is an algorithm to order vertices in a **Acylclic Digraph** such that **for any edge, src will come before dest** in the ordering

# Imagine we have a graph representing enrolments and pre-requisites

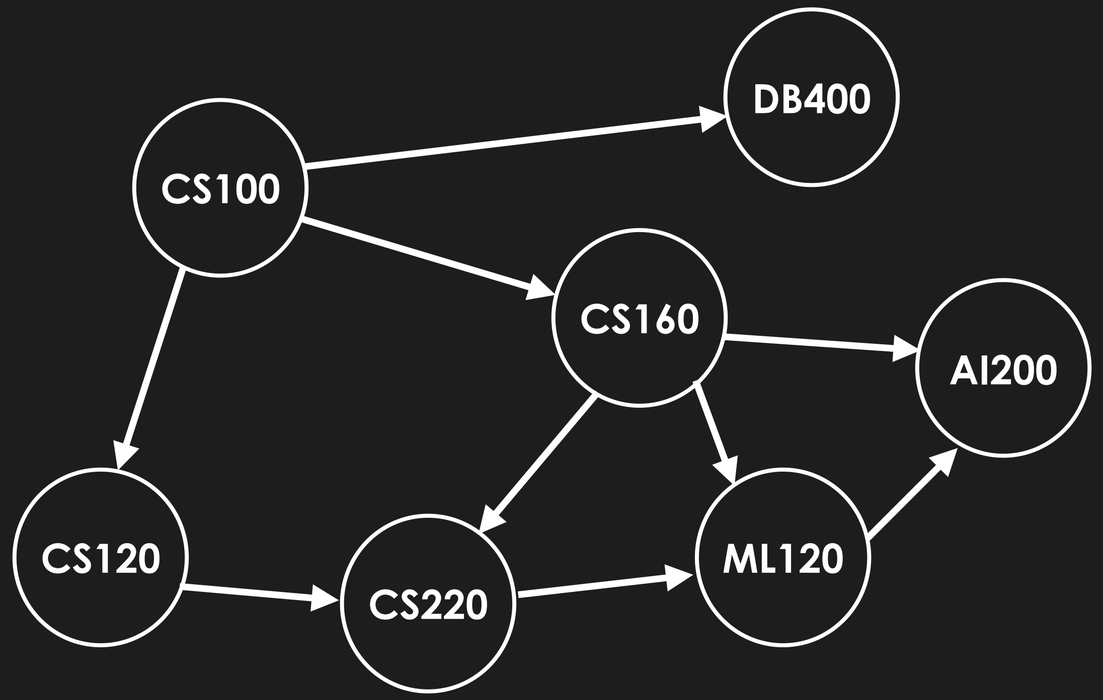



CodeIT




This graph is **acyclic** because there are no cycles formed by the edges

CodeIT

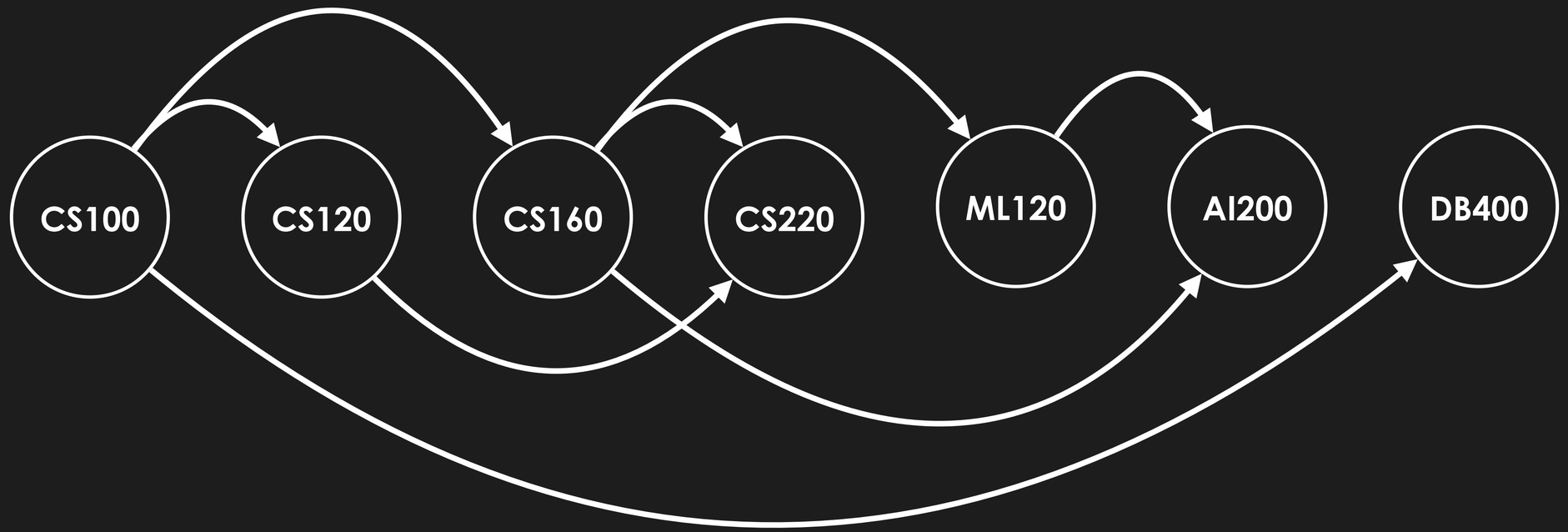



# Topological Ordering

CS100 CS120 CS160 CS220 ML120 AI200 DB400

CodeIT

# Topological Ordering




**Arrows only move right!**

CodeIT

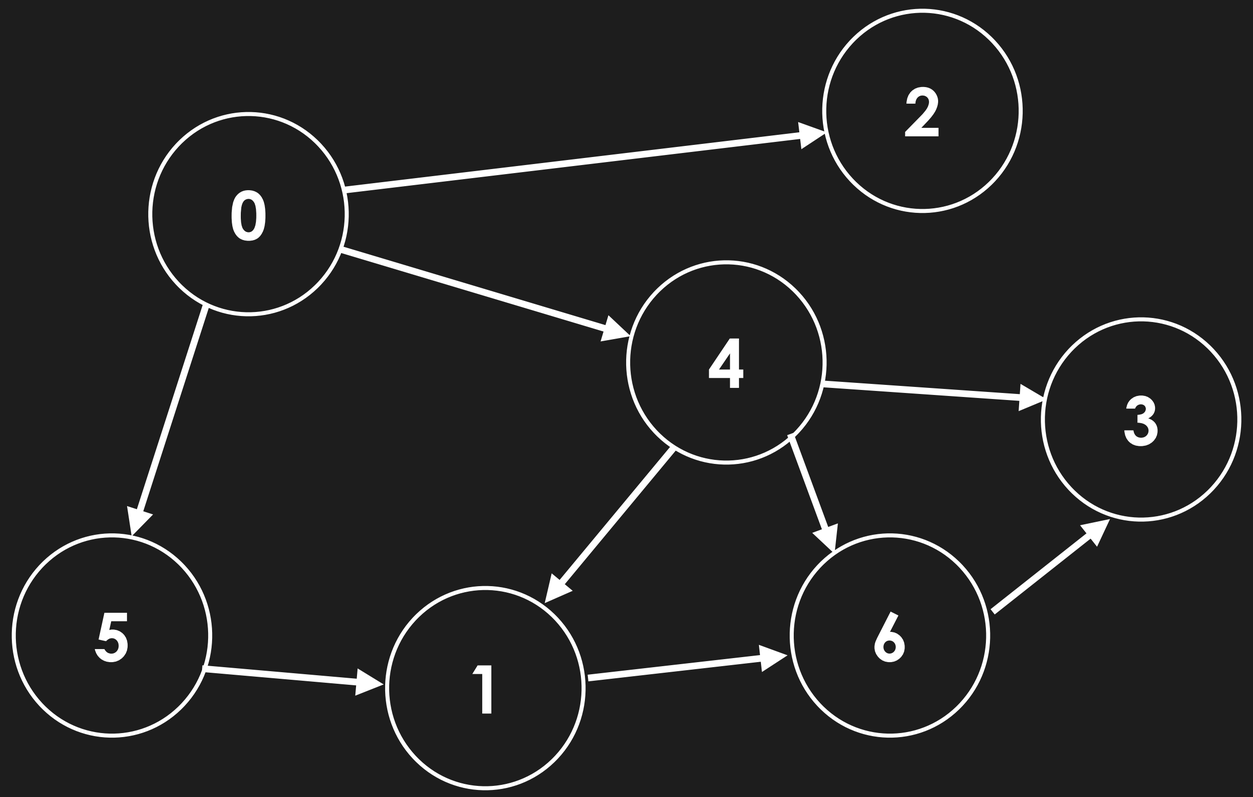
# Top Sort Algorithm

# Top Sort (reverse postorder DFS)

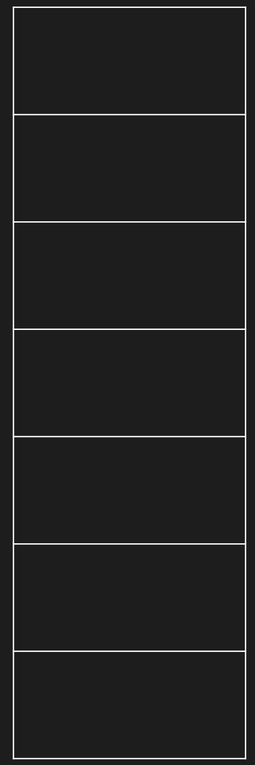
1. For each vertex, if not visited, DFS graph and at the end of each **function call**, insert into topological array

2. **Reverse** array

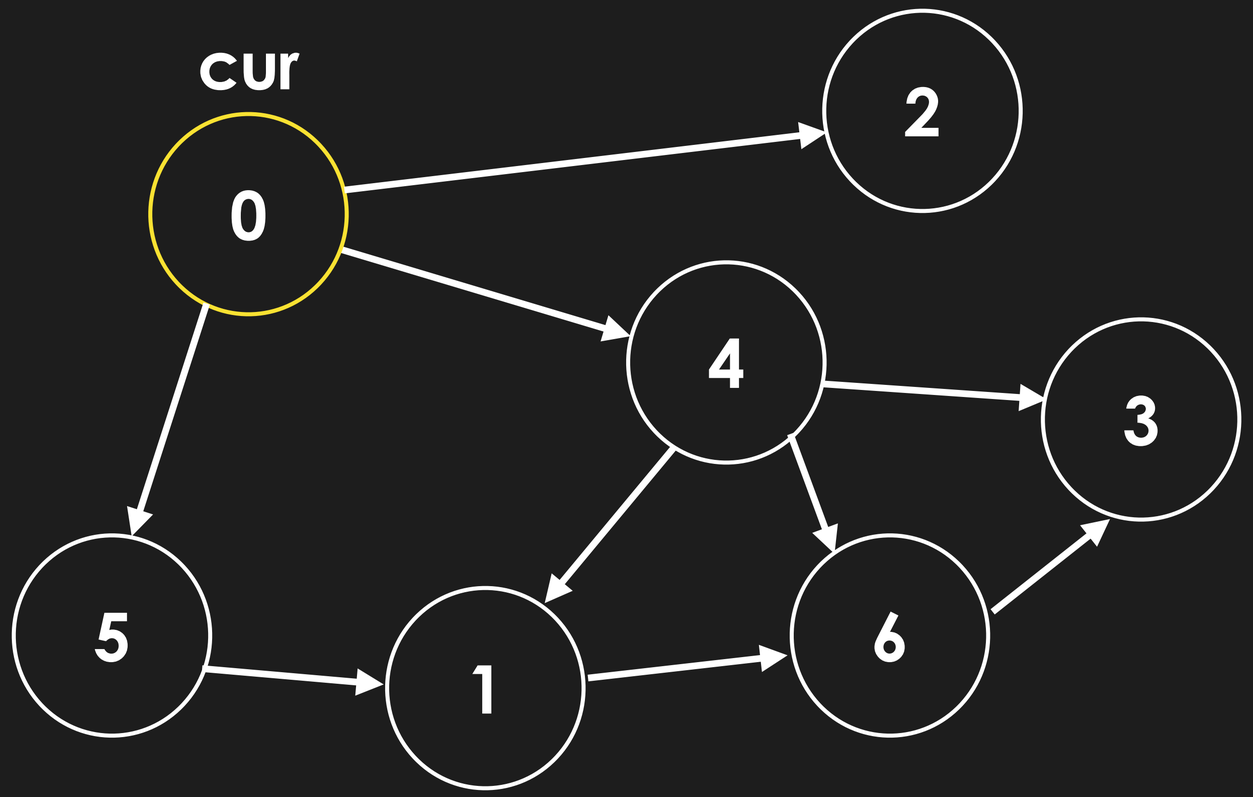
# Top Sort



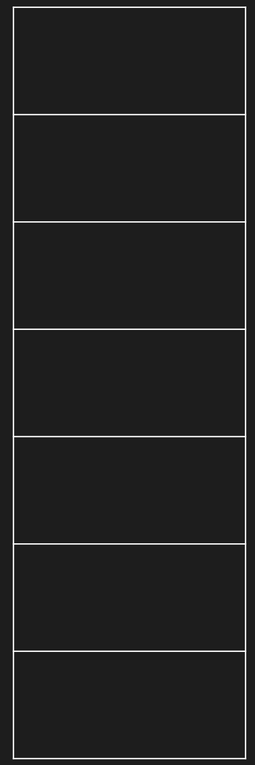

## Sorted Array



# Top Sort

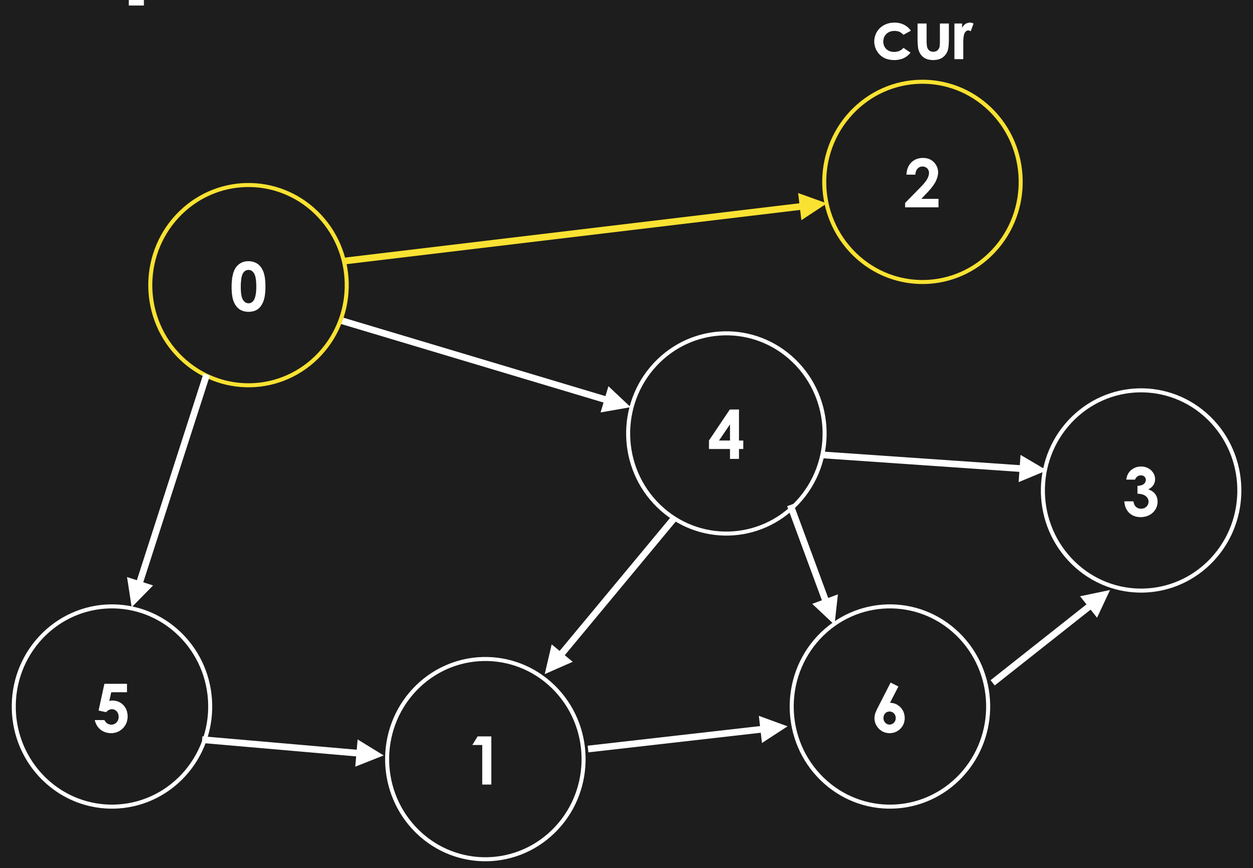



## Sorted Array

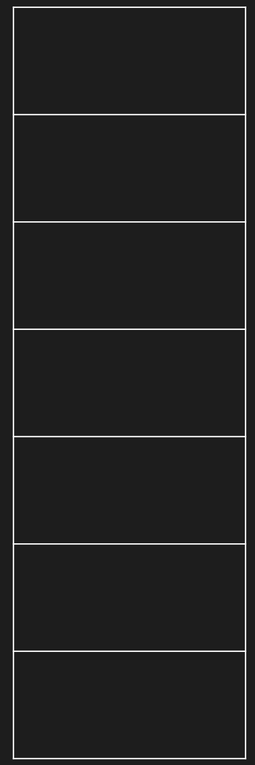


CodeIT

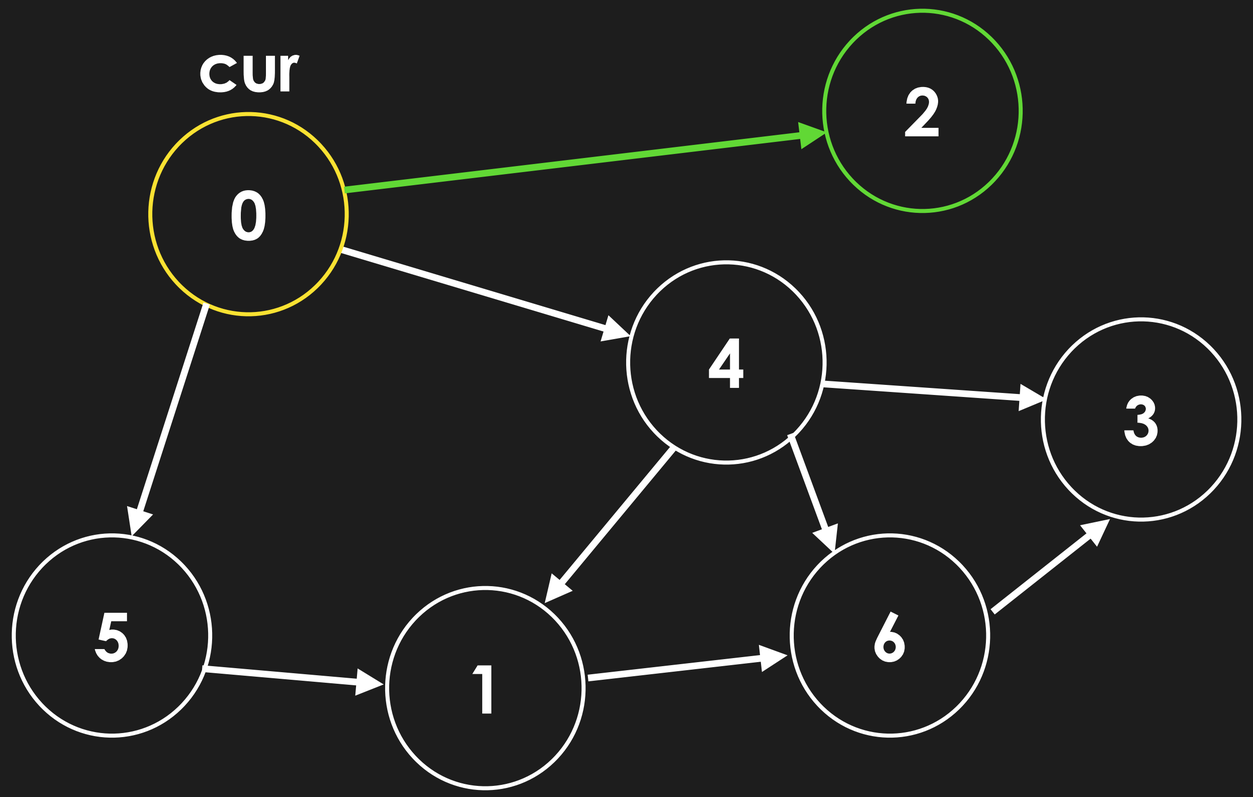
# Top Sort



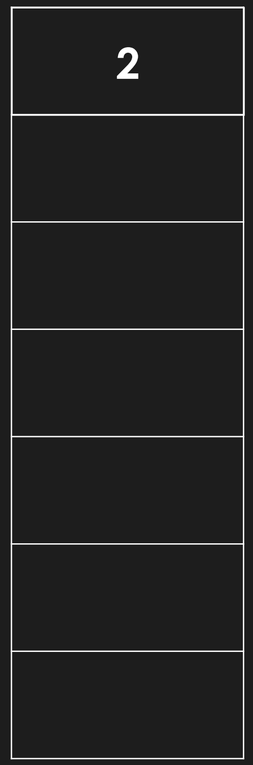

## Sorted Array



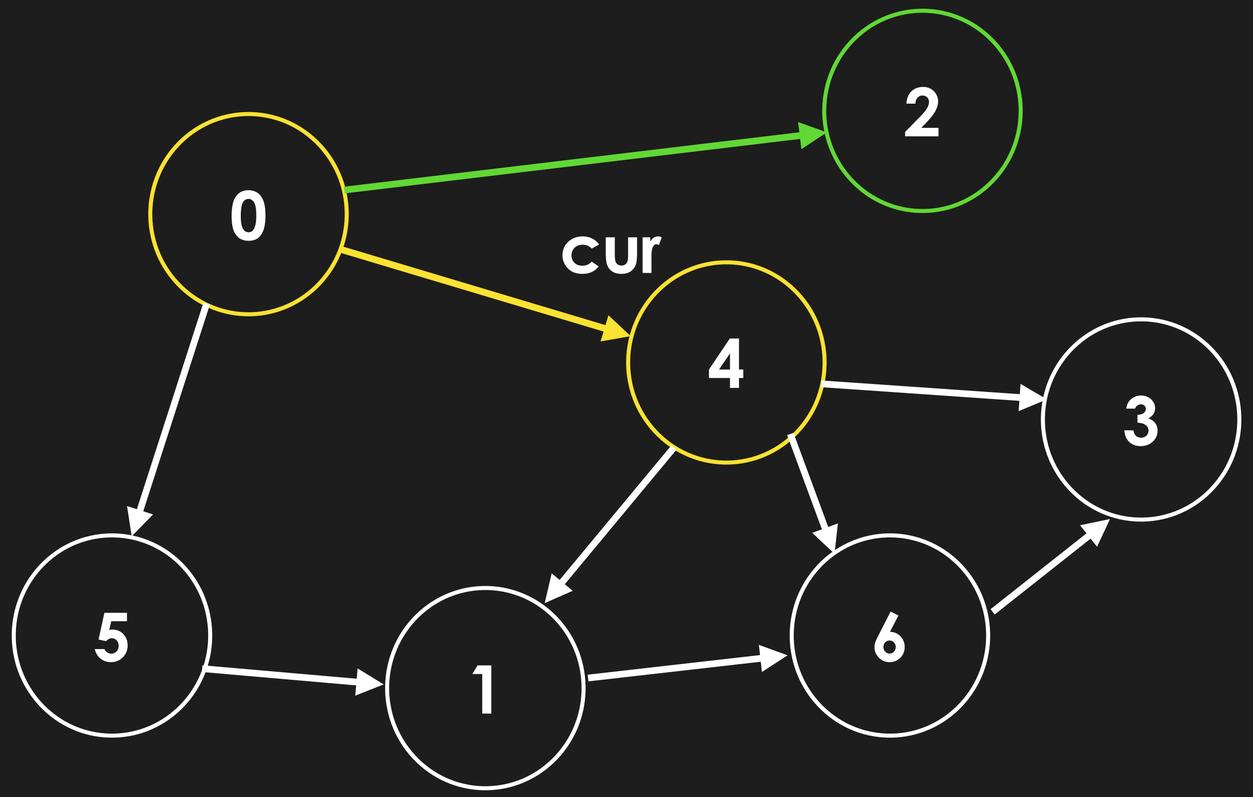
# Top Sort



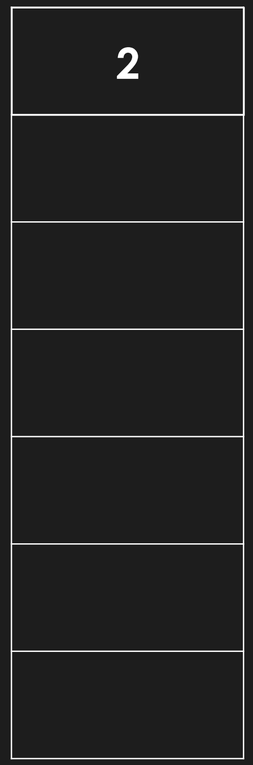

## Sorted Array



# Top Sort

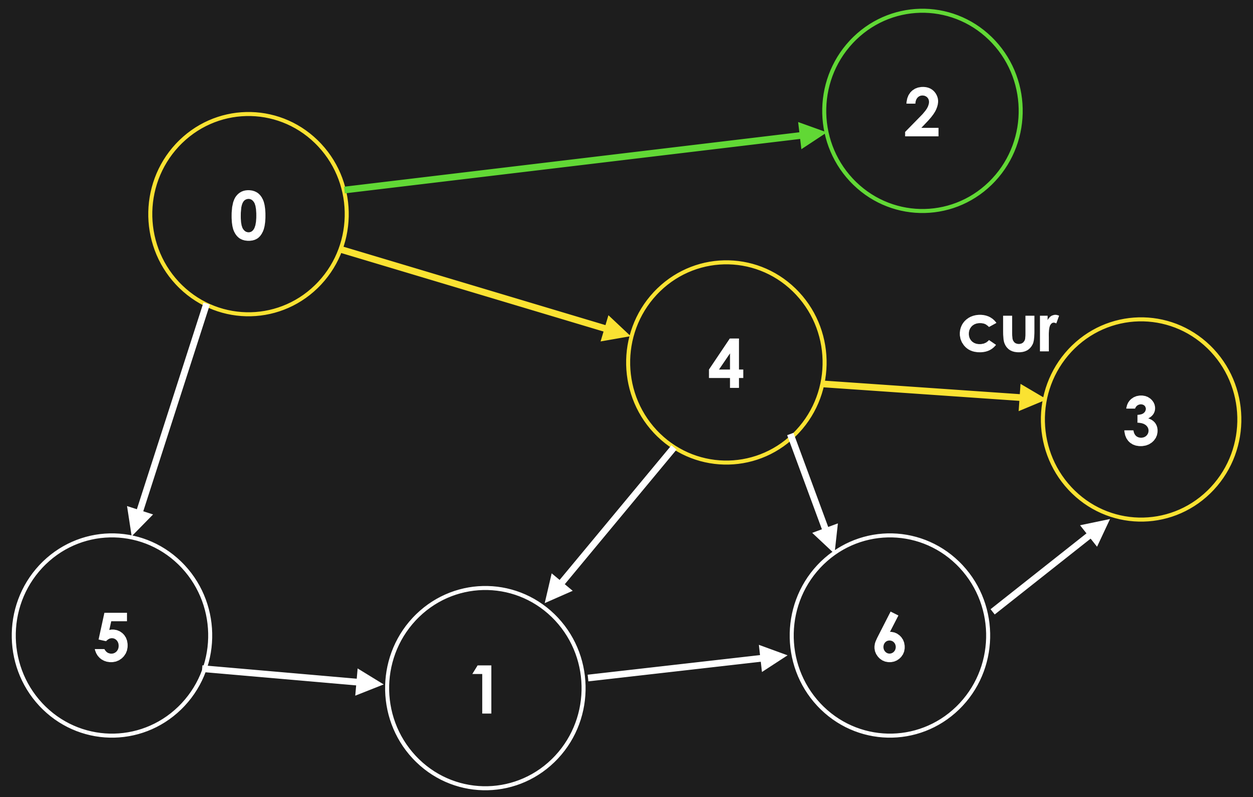



## Sorted Array

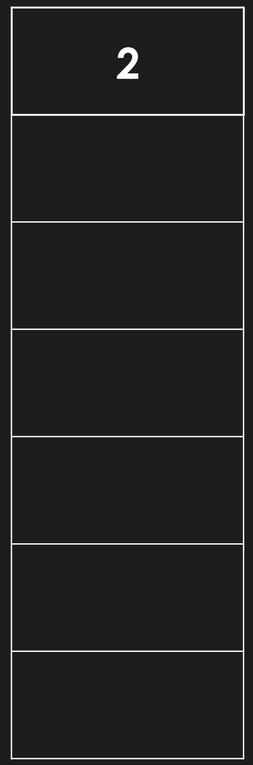



CodeIT

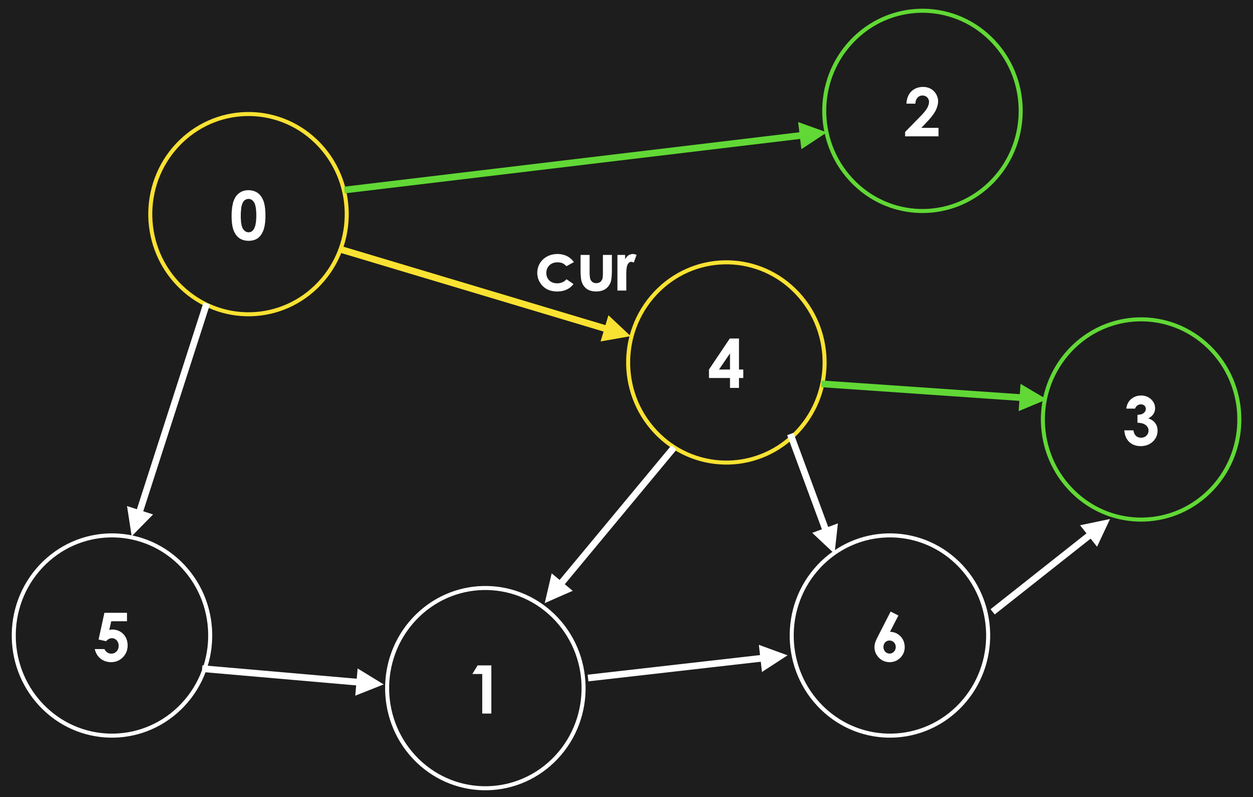
# Top Sort




## Sorted Array



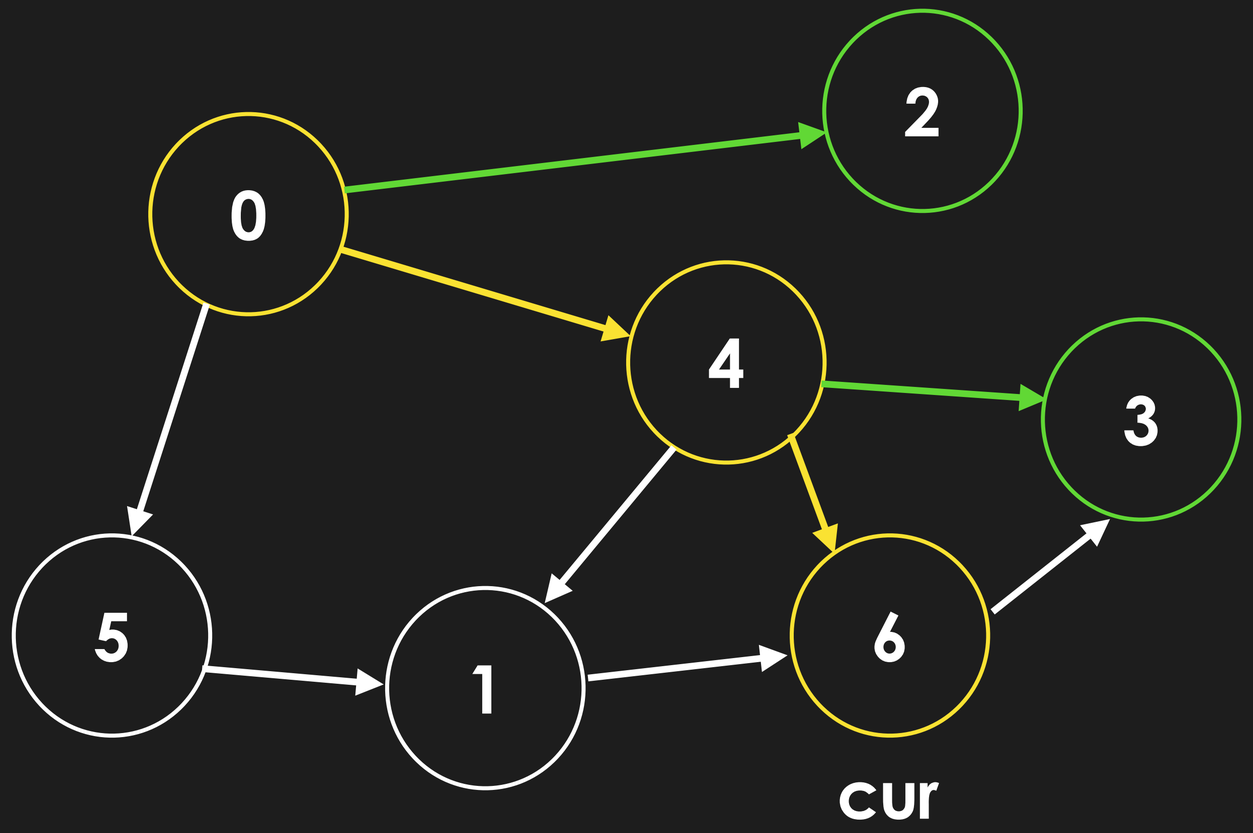
# Top Sort




## Sorted Array

| |
|---|
| 2 |
| 3 |
| |
| |
| |
| |
| |

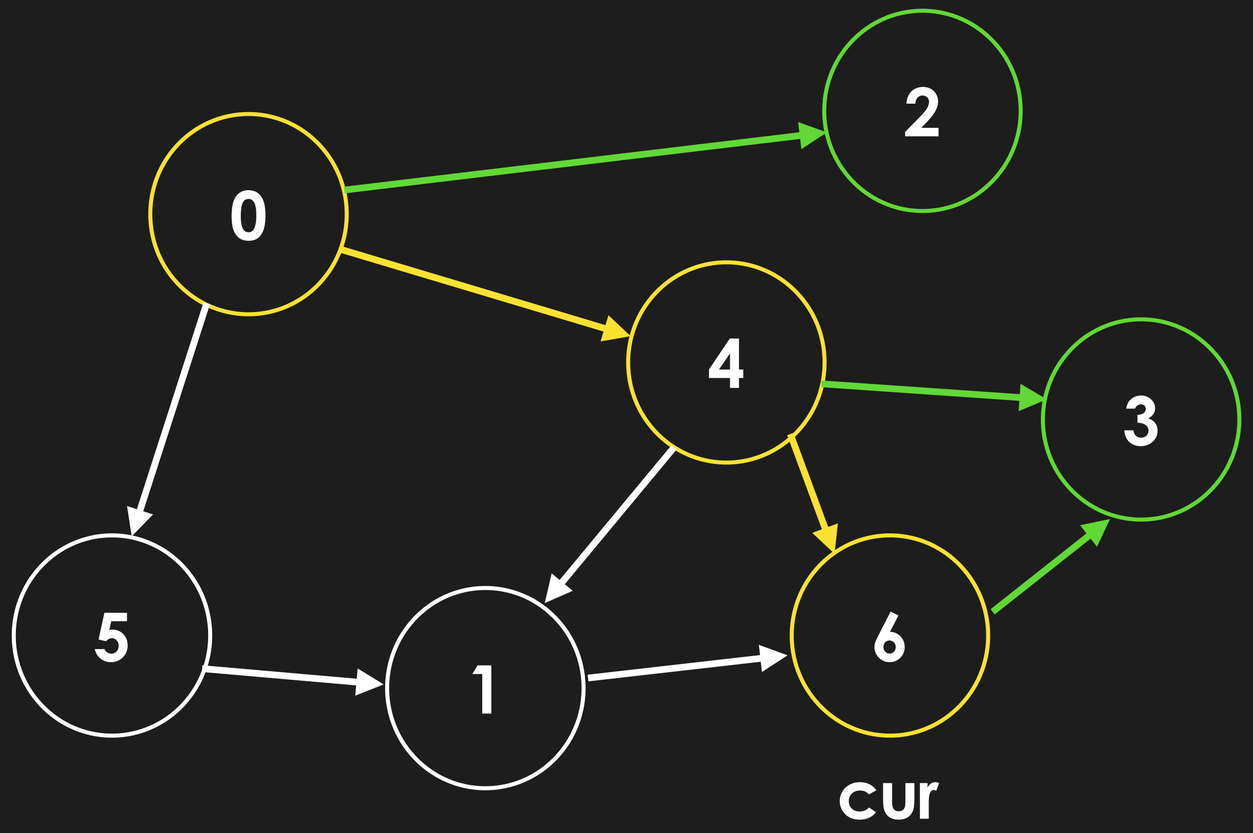
# Top Sort




## Sorted Array

| |
|---|
| 2 |
| 3 |
| |
| |
| |
| |
| |

# Top Sort

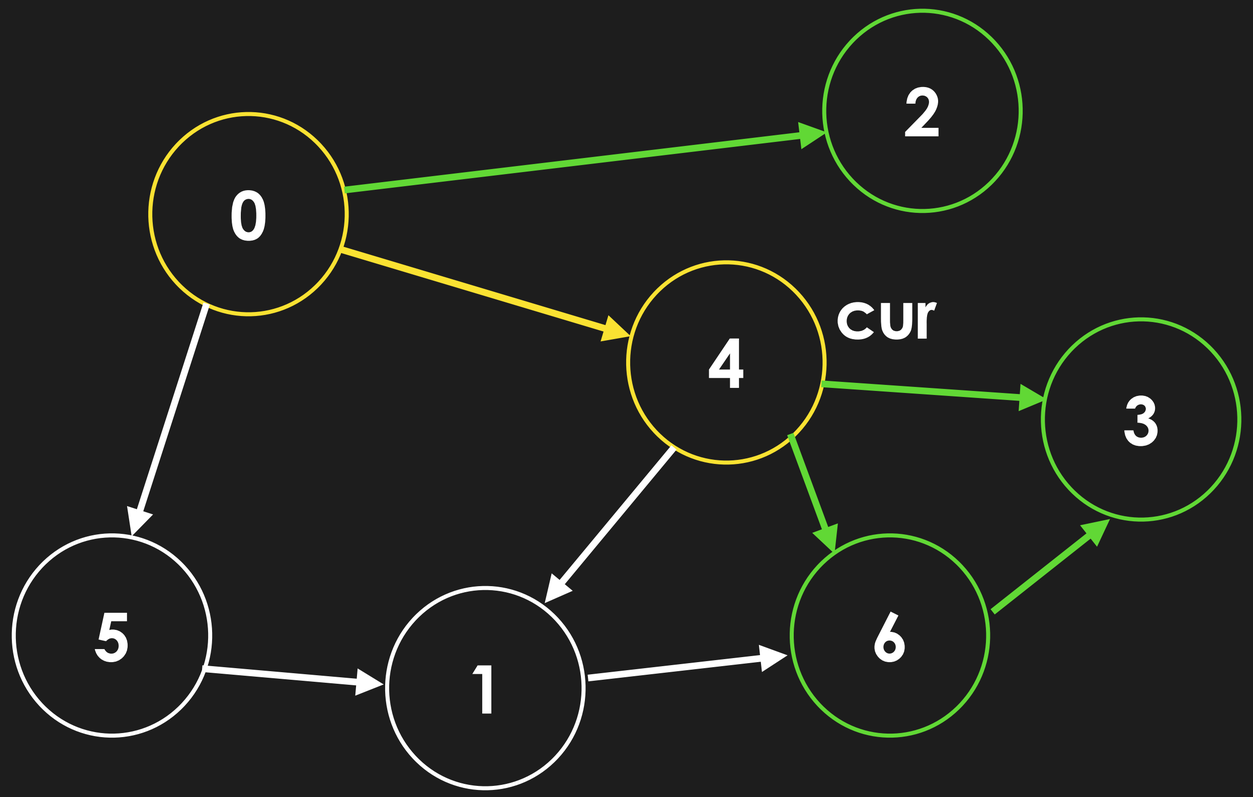



## Sorted Array

| |
|---|
| 2 |
| 3 |
| |
| |
| |
| |
| |

CodeIT

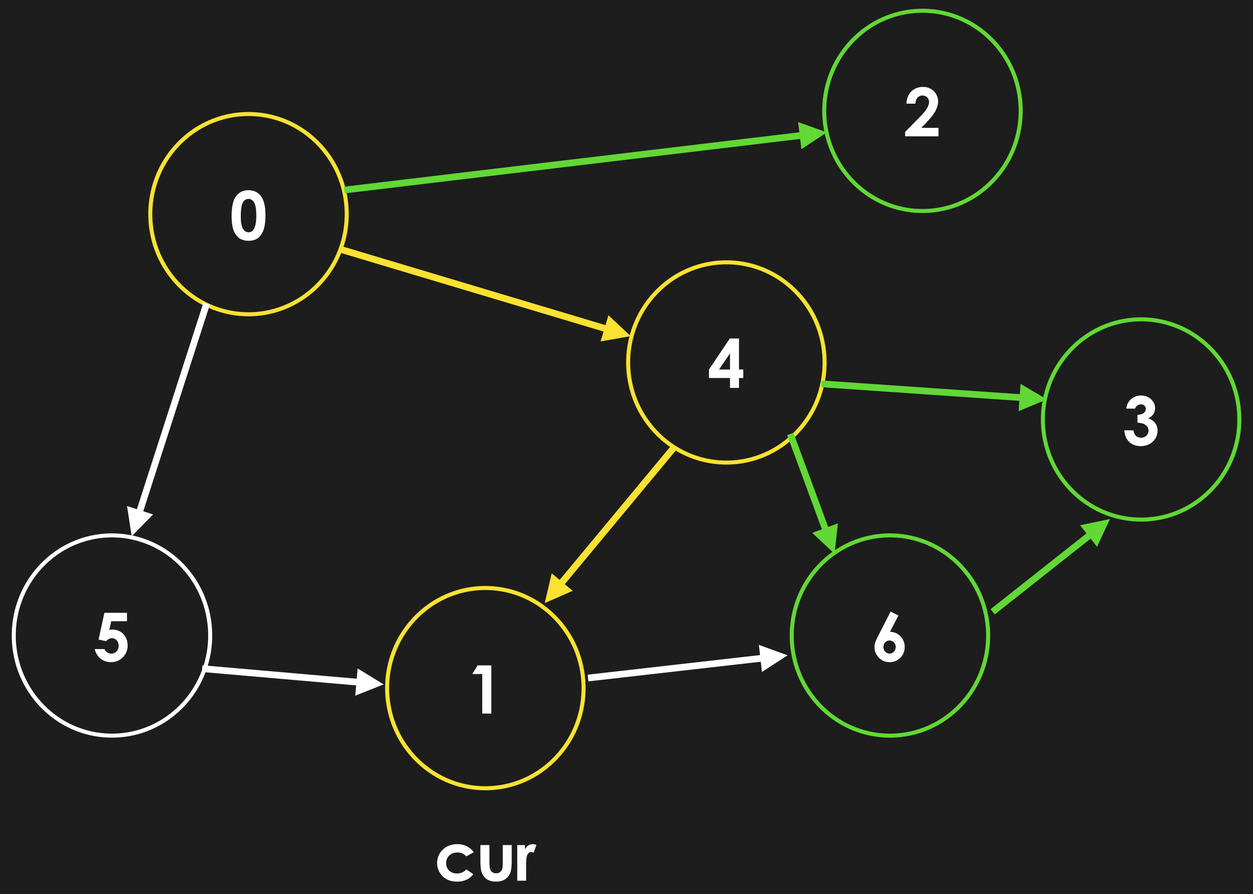
# Top Sort




## Sorted Array

| |
|---|
| 2 |
| 3 |
| 6 |
| |
| |
| |
| |

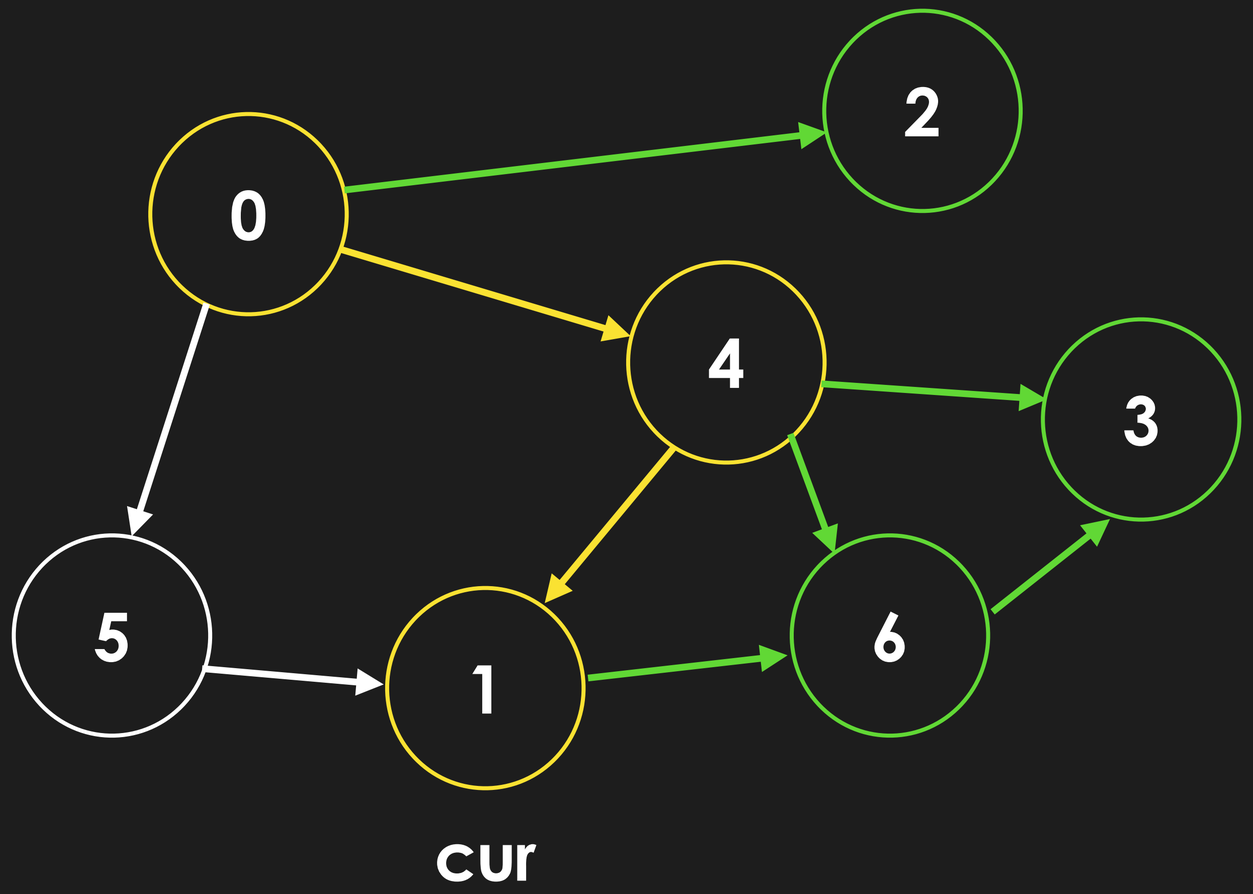
# Top Sort




## Sorted Array

| |
|---|
| 2 |
| 3 |
| 6 |
| |
| |
| |
| |

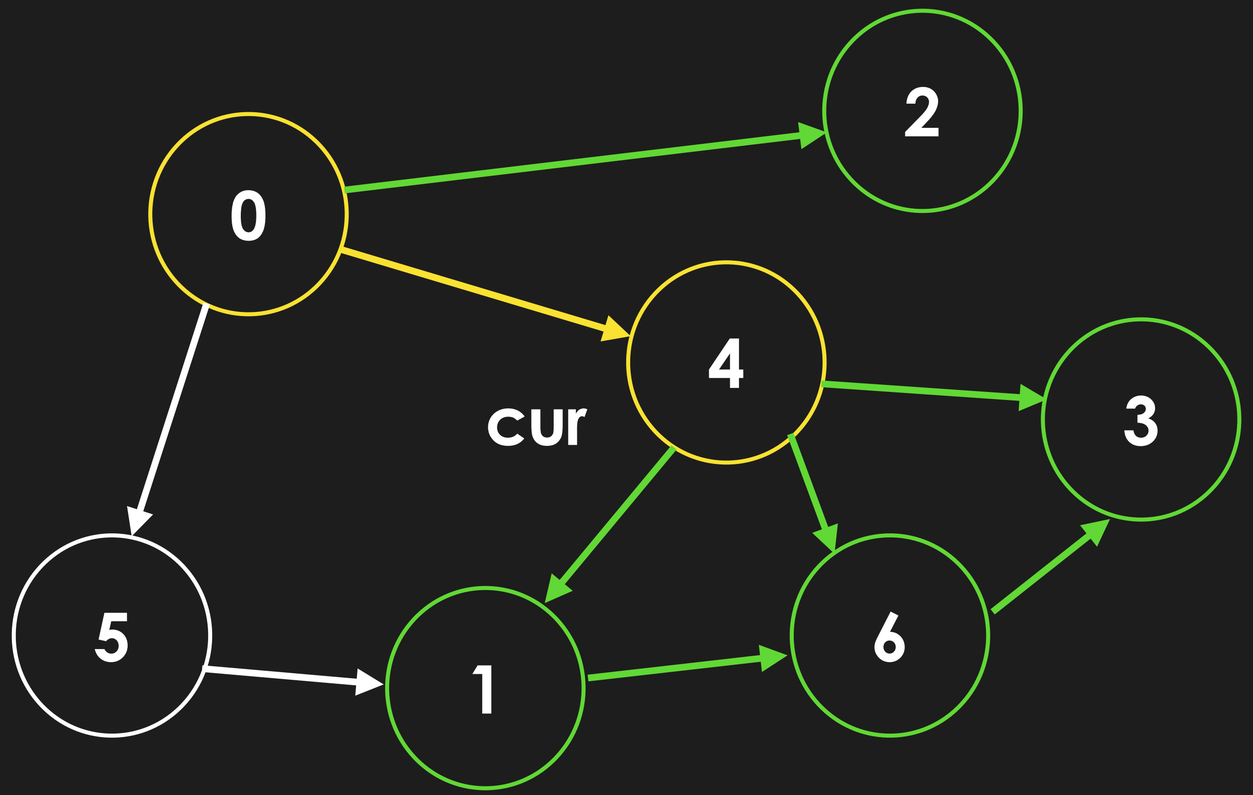
# Top Sort




## Sorted Array

| |
|---|
| 2 |
| 3 |
| 6 |
| |
| |
| |
| |

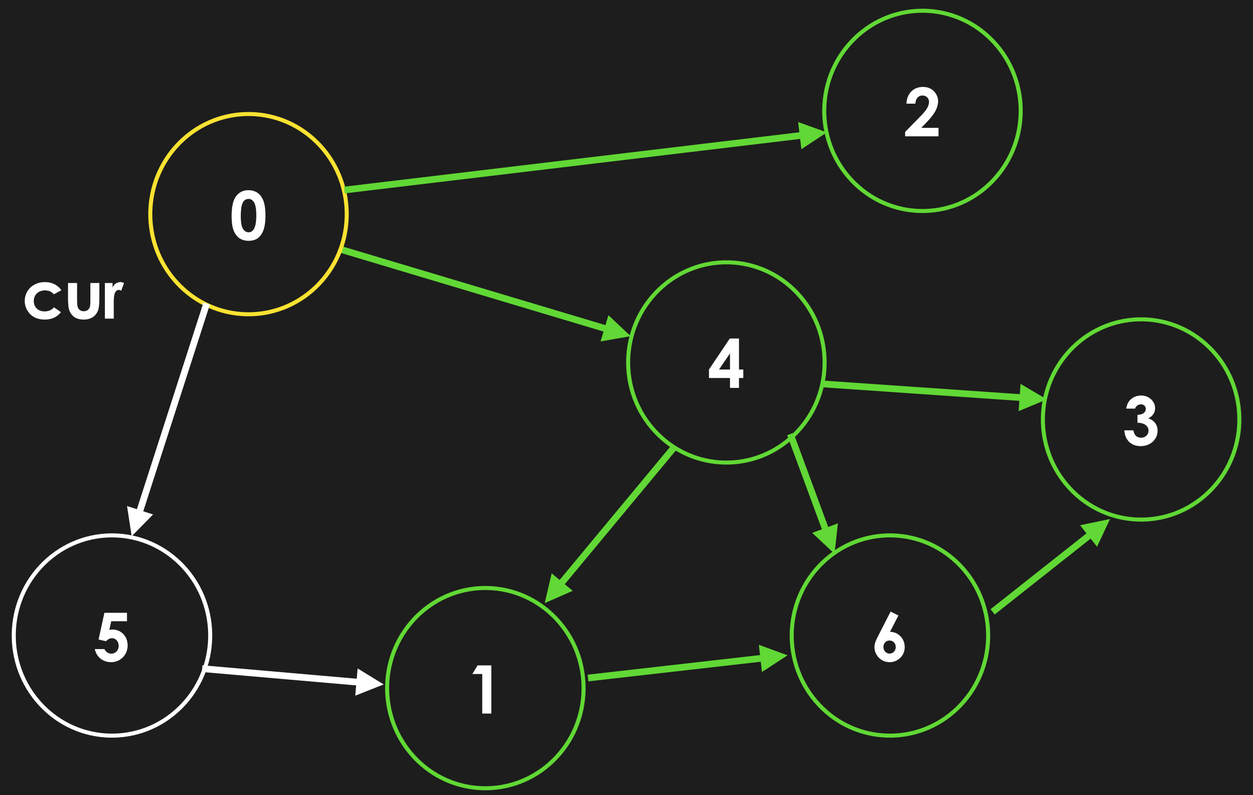
# Top Sort




## Sorted Array

| |
|---|
| 2 |
| 3 |
| 6 |
| 1 |
| |
| |
| |

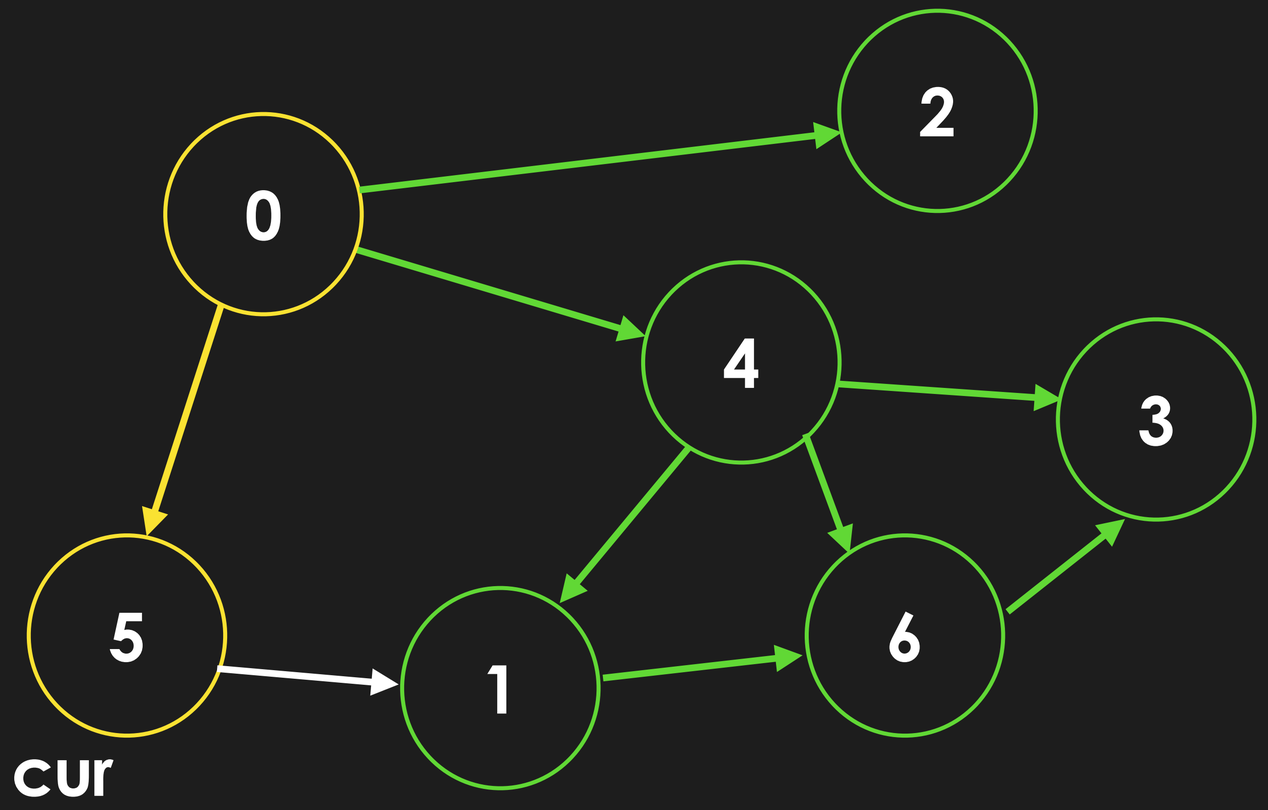
# Top Sort




## Sorted Array

| |
|---|
| 2 |
| 3 |
| 6 |
| 1 |
| 4 |
| |
| |

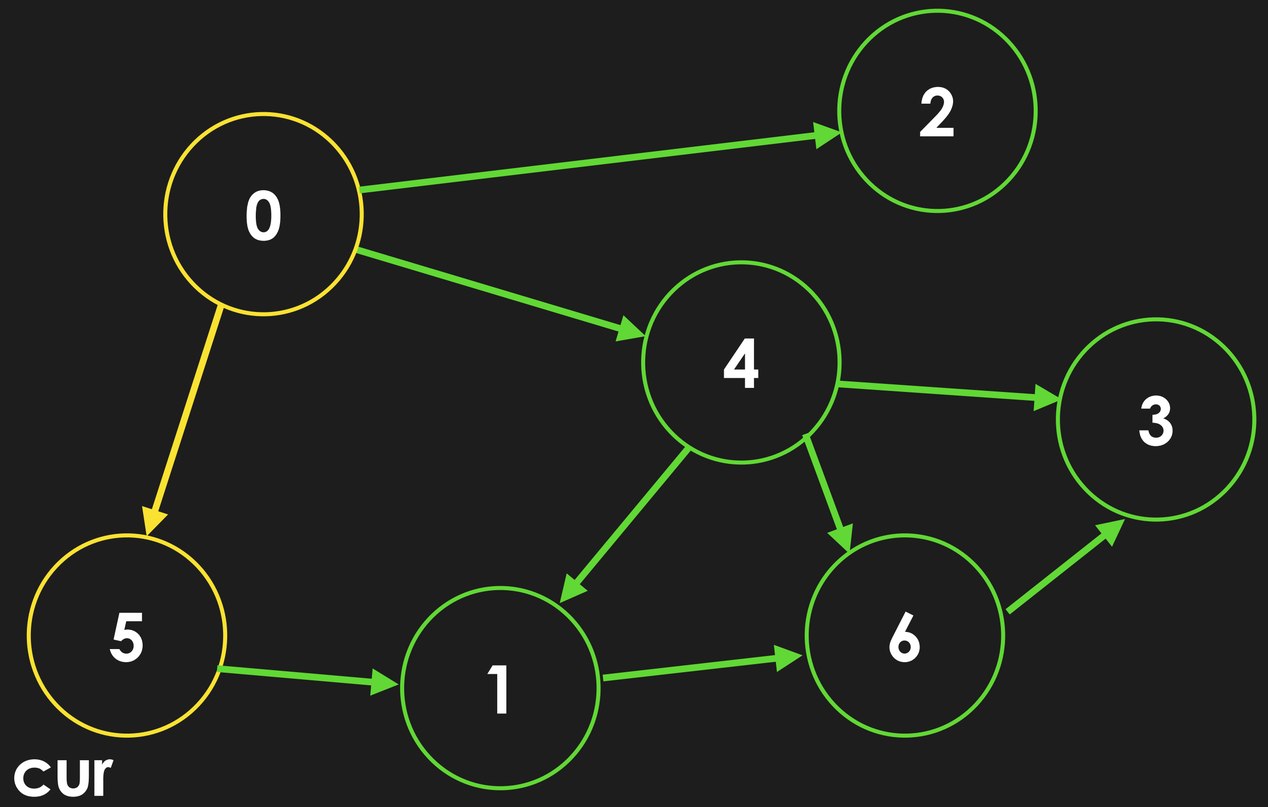
# Top Sort




## Sorted Array

| 2 |
|---|
| 3 |
| 6 |
| 1 |
| 4 |
| |
| |

# Top Sort

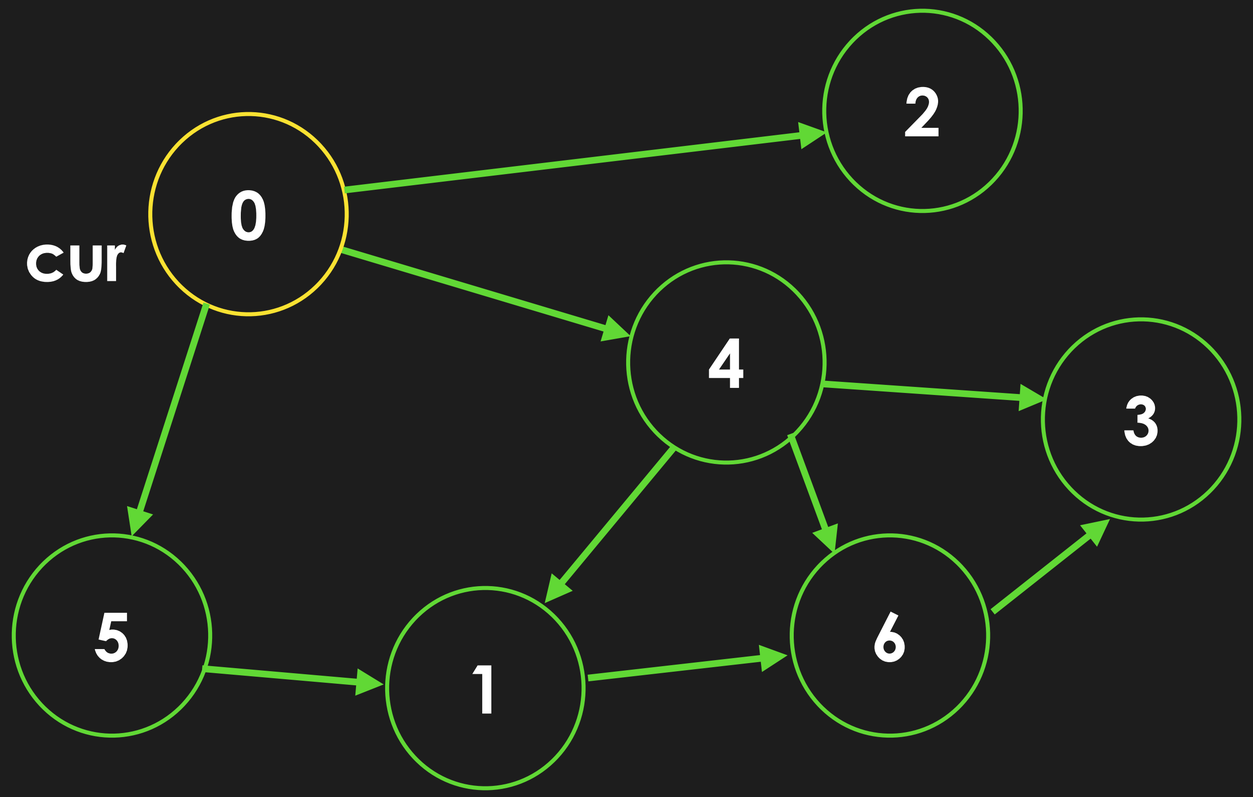



## Sorted Array

| 2 |
|---|
| 3 |
| 6 |
| 1 |
| 4 |
| |
| |

CodeIT

# Top Sort

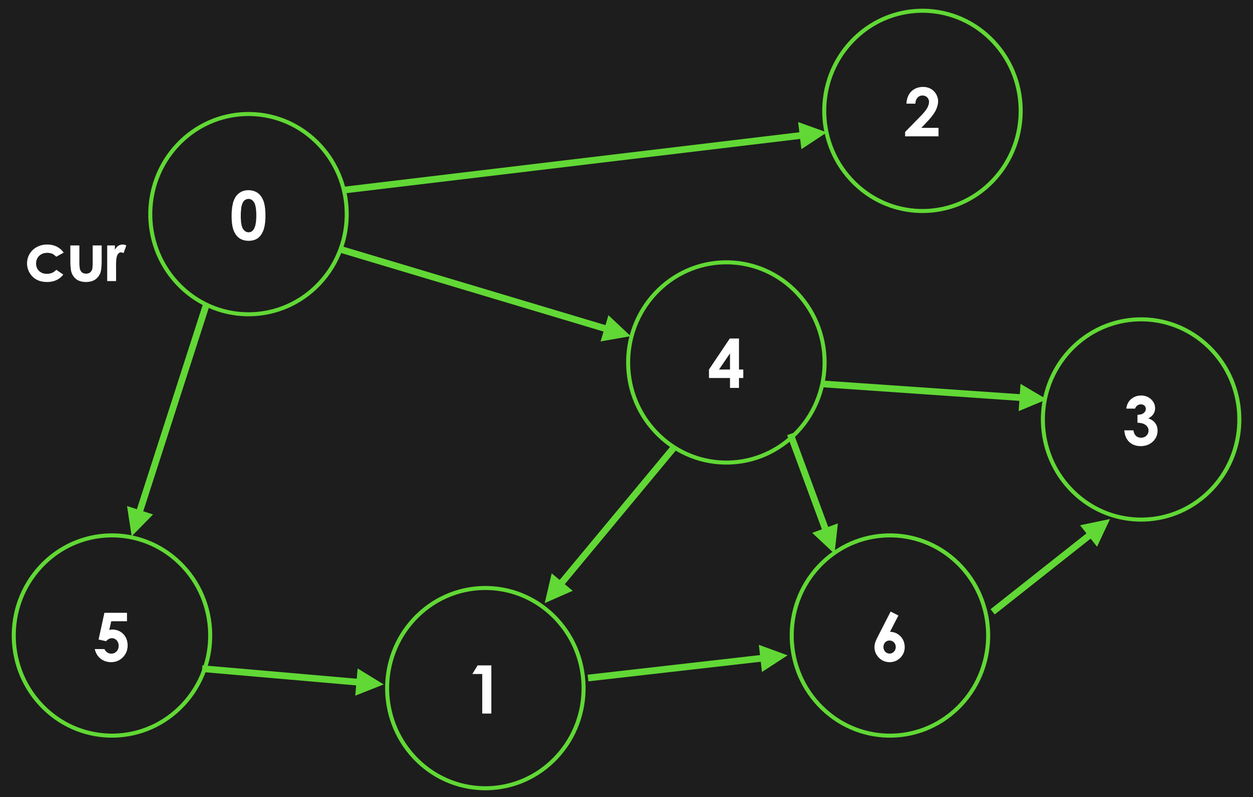



## Sorted Array

| |
|---|
| 2 |
| 3 |
| 6 |
| 1 |
| 4 |
| 5 |
| |

CodeIT

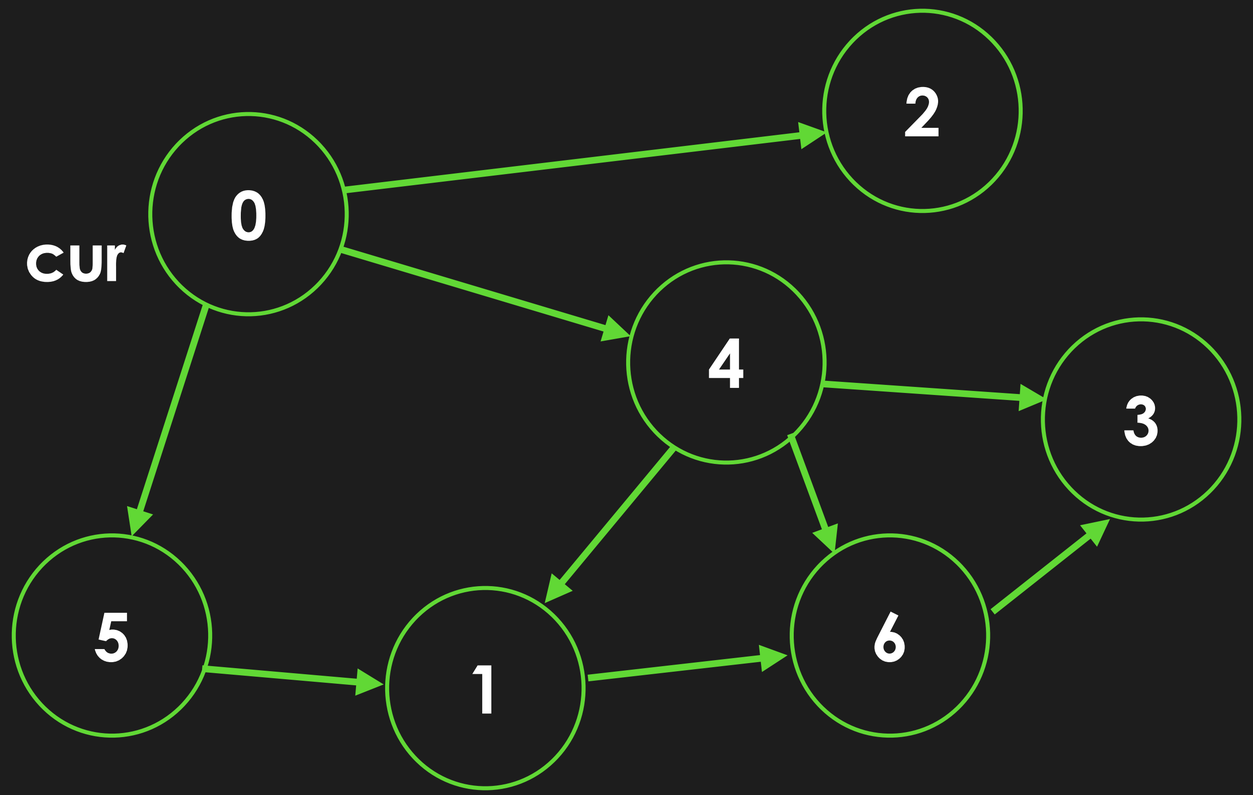
# Top Sort




## Sorted Array

| 2 |
|---|
| 3 |
| 6 |
| 1 |
| 4 |
| 5 |
| 0 |

# Top Sort




Sorted Array

| 2 |
|---|
| 3 |
| 6 |
| 1 |
| 4 |
| 5 |
| 0 |

Repeat for all unvisited vertices

CodeIT

# Sorted Array


CodeIT

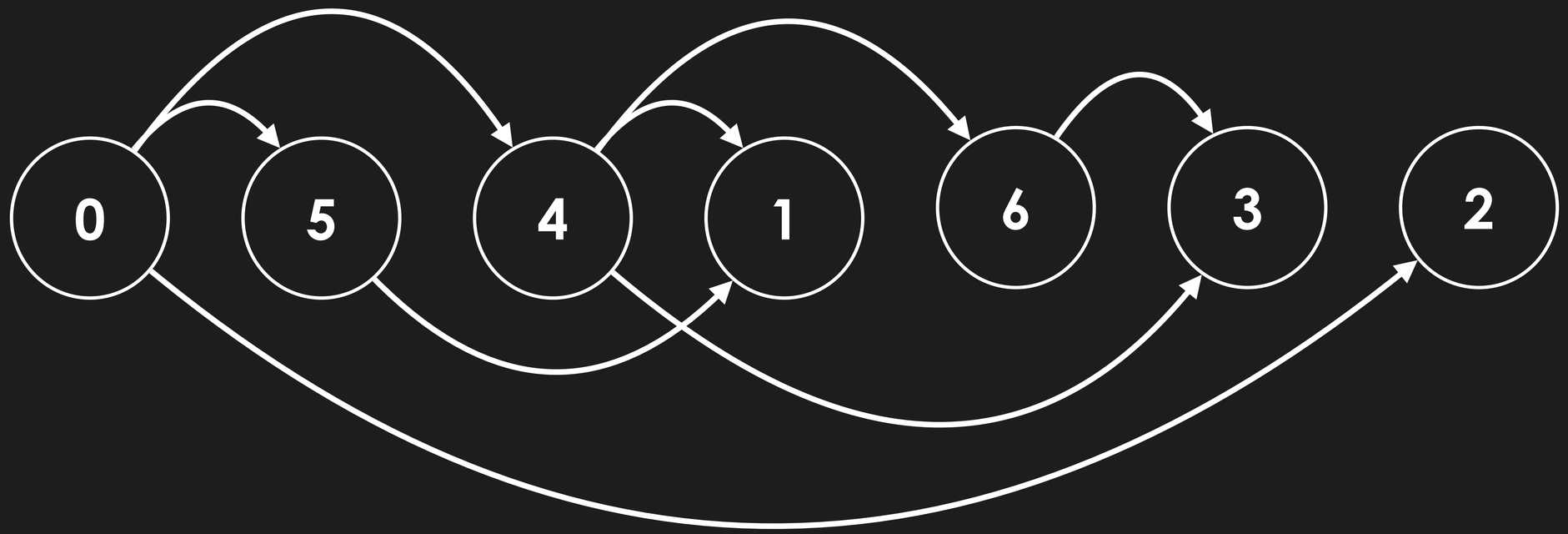
# Reverse Sorted Array

| 0 | 5 | 4 | 1 | 6 | 3 | 2 |
|---|---|---|---|---|---|---|

CodeIT

# Reverse Sorted Array

| 0 | 5 | 4 | 1 | 6 | 3 | 2 |
|---|---|---|---|---|---|---|




CodeIT

# Top Sort Implementation

# topSort

```
def dfsRecurse(digraph, v, result, visited):
    visited[v] = True

    for edge in digraph.adjList[v]:
        if not visited[edge.dest]:
            dfsRecurse(digraph, edge.dest, result, visited)

    result.append(v)


def topSort(digraph):
    visited = [False] * len(digraph.adjList)
    result = []

    for i in range(len(digraph.adjList)):
        if not visited[i]:
            dfsRecurse(digraph, i, result, visited)

    result.reverse()
    return result
```

CodeIT

```
V = 7
digraph = Digraph(V)
edges = [(0, 2), (0, 4), (0, 5), (4, 3), (4, 6),
            (4, 1), (6, 3), (1, 6), (5, 1)]
for edge in edges:
    digraph.addEdge(edge)

print(topSort(digraph, 0))
```

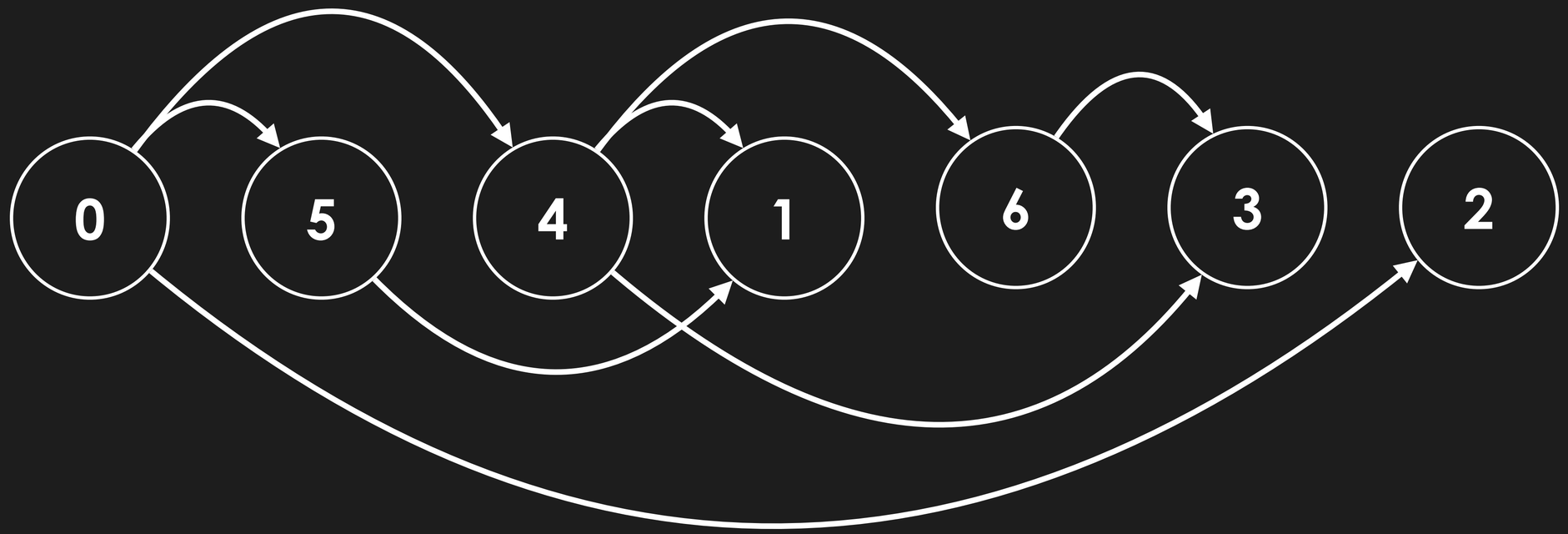
```
V = 7
digraph = Digraph(V)
edges = [(0, 2), (0, 4), (0, 5), (4, 3), (4, 6),
             (4, 1), (6, 3), (1, 6), (5, 1)]
for edge in edges:
    digraph.addEdge(edge)

print(topSort(digraph, 0))
```

```
[0, 5, 4, 1, 6, 3, 2]
```

CodeIT

```
[0, 5, 4, 1, 6, 3, 2]
```




CodeIT

# Lab Session 2

- In this lab session, you will be implementing **topsort.py**
- Your task is to implement the top sort algorithm on a Directed Graph
- In your function, you are to check for cycles in the graph. If it is acyclic, return a topologically sorted list of vertices, else return None
- To test, run `**python utils/traversal_test.py**`

**Digraph**

- This class represents an directed graph and has been implemented for you.
- This graph contains an attribute `adjList` where `adjList[v]` contains all adjacent DirectedEdges to vertex v. For instance, if `adjList[0] = [DirectedEdge(0, 1), DirectedEdge(0, 2), DirectedEdge(0, 3)]`, then vertex 0 has three **outgoing edges** to vertices 1, 2, & 3

# Helper Classes

```
class DirectedEdge:
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest

class Digraph:
    def __init__(self, V):
        self.adjList = [[] for i in range(V)]

    def addEdge(self, edge):
        src, dest = edge
        self.adjList[src].append(DirectedEdge(src, dest))

    def printGraph(self):
        for v in range(len(self.adjList)):
            print("vertex {}: ".format(v), end="")
            for edge in self.adjList[v]:
                print("({}, {})".format(edge.src, edge.dest), end=" ")
            print()
```

# topsort solution

```
def dfsRecurse(digraph, v, result, visited, stack):
    visited[v] = True
    stack[v] = True

    for edge in digraph.adjList[v]:
        if stack[edge.dest]:
            return True

        if not visited[edge.dest]:
            hasCycle = dfsRecurse(digraph, edge.dest, result, visited,
stack)
            if hasCycle:
                return True

    result.append(v)
    stack[v] = False
    return False
```

# topsort solution

```
def topsort(digraph):


    visited = [False] * len(digraph.adjList)
    stack = [False] * len(digraph.adjList)
    result = []

    for i in range(len(digraph.adjList)):
        if not visited[i]:
            hasCycle = dfsRecurse(digraph, i, result, visited, stack)
            if hasCycle:
                return

    result.reverse()

    return result
```

CodeIT