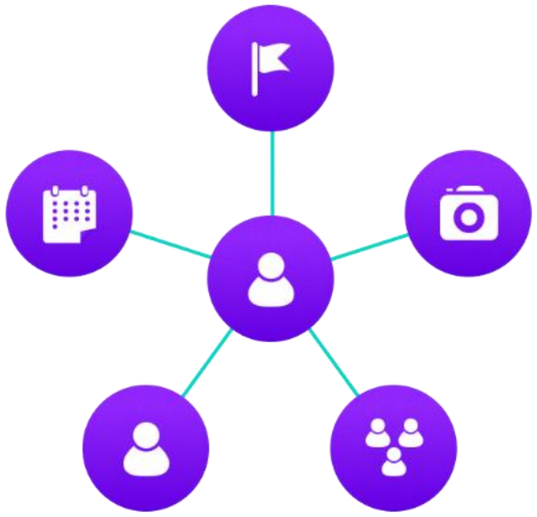
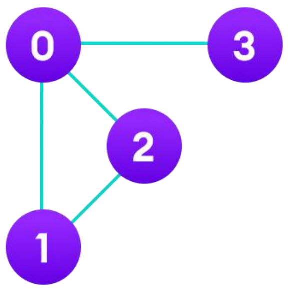
Graph Data Stucture

A graph data structure is a collection of nodes that have data and are connected to other nodes.



More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)



Vertices and edges

In the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

Graph Terminology

- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Graph Representation

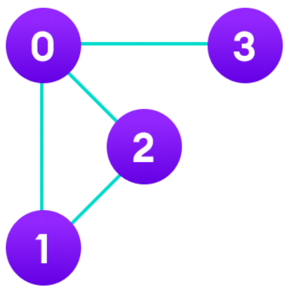
Graphs are commonly represented in two ways:

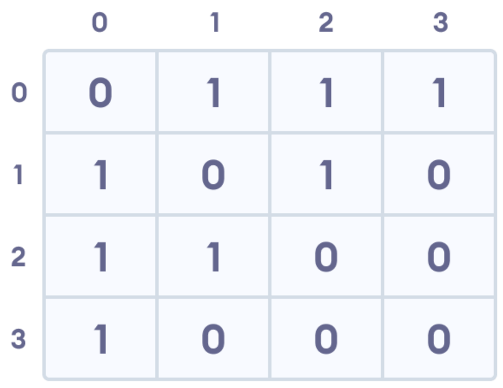
1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is





	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Graph adjacency matrix

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

Python Implementation

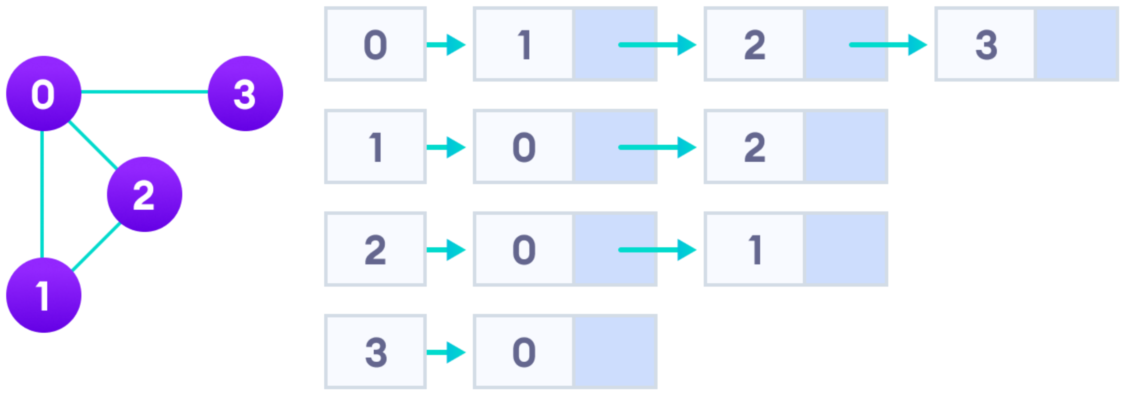
```
def create_graph_matrix(v, e, edges):
    graph = [[str(0)] * (v + 1) for i in range(v + 1)]
    for edge in edges:
        u, v, w = edge
        graph[u][v] = str(w)
    return graph
```

2. Adjacency List

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.

The adjacency list for the graph we made in the first example is as follows:



Adjacency list representation

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

Python Implementation

```
def create_graph_list(v, e, edges):

    graph = [None] * (v + 1)

    for edge in edges:

        u, v, w = edge

        if not graph[u]:

            graph[u] = []

        graph[u].append((v, w))

    return graph
```

Graph Operations

The most common graph operations are:

- Check if the element is present in the graph
- Graph Traversal
- Add elements (vertex, edges) to graph
- Finding the path from one vertex to another

Graph Traversal

BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of **O(V + E)**, where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

Python Implementation

```
def BFS (graph, s, vertex):
    Q = []
    state = [0] * (vertex + 1)
    Q.append(s)
    state[s] = 1
    while Q:
        u = Q.pop(0)
        print(u, end=" ")
        if graph[u]:
            state[u] = 1
            for v in graph[u]:
                if state[v] == 0:
                    Q.append(v)
                    state[v] = 1
            state[u] = 2
```

Other Implementations:

#Shortest Path and Time

```
def BFS(graph, s, vertex):
    Q = []
    state = [0] * (vertex + 1)
    distance = [0] * (vertex + 1)
    parent = [0] * (vertex + 1)
    Q.append(s)
    state[s] = 1
    while Q:
        u = Q.pop(0)
        print(u, end=" ")
        if graph[u]:
            state[u] = 1
            for v in graph[u]:
                if state[v] == 0:
                    Q.append(v)
                    state[v] = 1
                    distance[v] = distance[u] + 1
                    parent[v] = u
            state[u] = 2
    parent[s] = -1
    print()
    return parent, distance


def find_path(parent, D):
    if parent[D] == -1:
        print("Shortest Path:", D, end=" ")
        return
    else:
        find_path(parent, parent[D])
        print(D, end=" ")
```

# Bipartite Check:

```
def BFS(graph, s, vertex):
    Q = []
    state = [0] * (vertex + 1)
    distance = [0] * (vertex + 1)
    color = [0] * (vertex + 1)
    vertexes = []
    Q.append(s)

    state[s] = 1
    while Q:
        u = Q.pop(0)
        vertexes.append(u)
        if graph[u]:
            state[u] = 1
            for v in graph[u]:
                if state[v] == 0:
                    Q.append(v)
                    state[v] = 1
                    distance[v] = distance[u] + 1
                    color[v] = distance[v] % 2
            state[u] = 2
    return color, vertexes


def bipartite(color, edges):
    for u, v in edges:
        if color[u] == color[v]:
            return False
    return True


edges = [(1, 3), (3, 2), (1, 4)]
graph = create_graph(4, 3, edges)
color, vertexes = BFS(graph, 1, len(graph) - 1)

print(bipartite(color, edges))
```

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of `O (V + E)`, where `V` is the number of nodes and `E` is the number of edges.
The space complexity of the algorithm is `O(V)`.

Python Implementation:

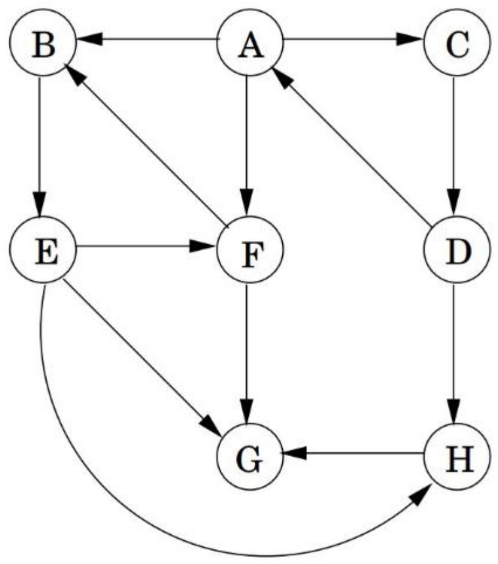
```
graph = create_graph(7, 7, [(1, 3), (3, 2), (1, 4), (2, 5), (5,
6), (1, 7), (7, 6)])

state = [0] * (len(graph) + 1)

def DFS(s):

    state[s] = 1

    print(s, end=" ")

    for v in graph[s]:

        if state[v] == 0:

            DFS(v)
```

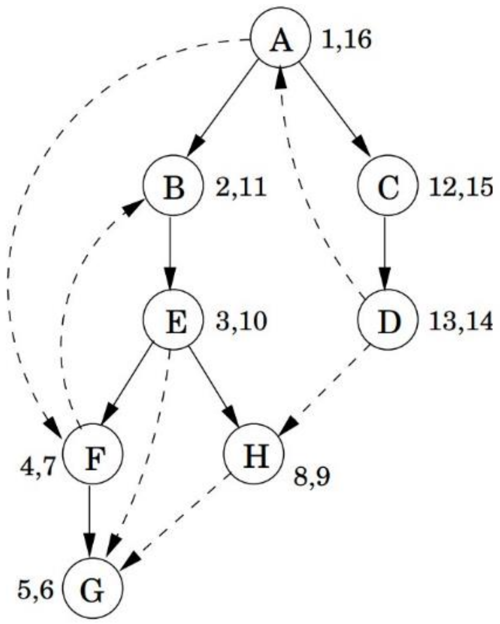
# Other Implementation

Cycle Detection

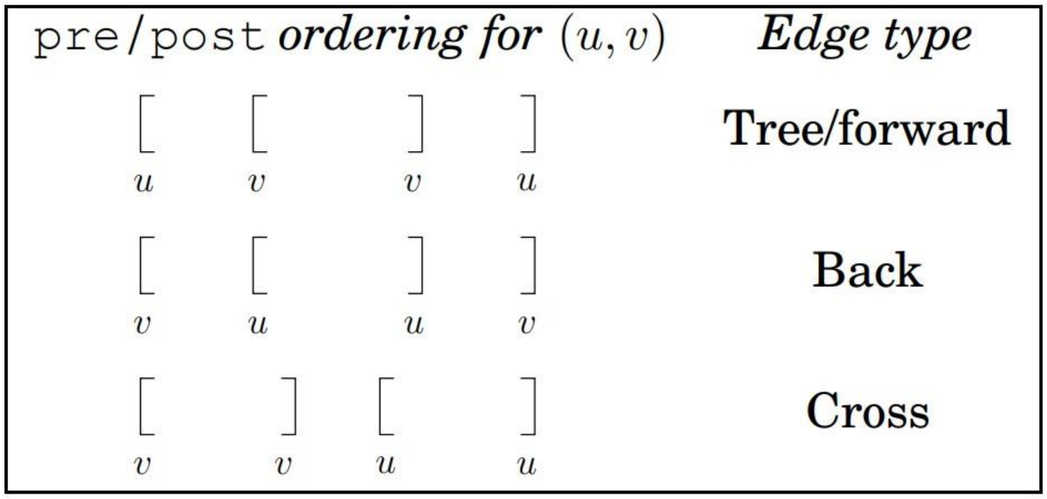
```
def cycle_detect(s, state, graph, parent=-1):

    state[s] = 1

    if graph[s]:

        for v in graph[s]:

            if state[v] == 0 and cycle_detect(v, state, graph, s)
== "YES":

                return "YES"

            elif state[v] == 1 and parent != v:

                return "YES"

    state[s] = 2

    return "NO"
```

Edge Classification





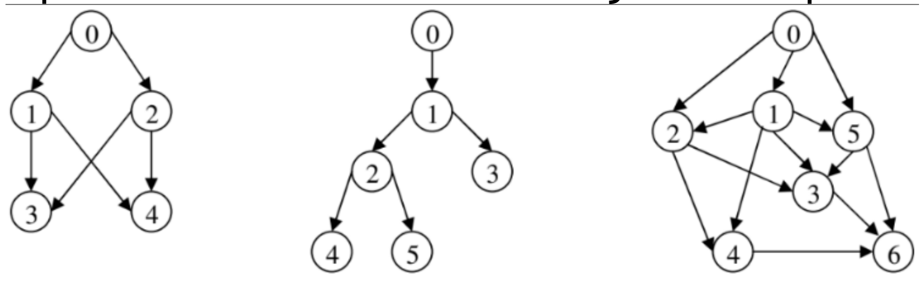
Edge Type (u, v)	**Pre/Post-Order**
Tree/forward	$\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$
Back	$\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$
Cross	$\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$



pre/post *ordering for* (u, v)
Edge type
$[_u \quad [_v \quad]_v \quad]_u$
Tree/forward
$[_v \quad [_u \quad]_u \quad]_v$
Back
$[_v \quad]_v \quad [_u \quad]_u$
Cross

Topological Sorting

- The Graph has to be a Directed Acyclic Graph.

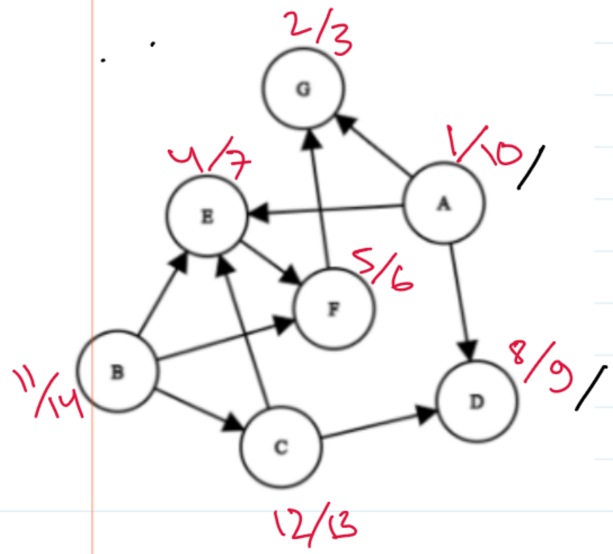


0, 1, 2, 3, 4	0, 1, 3, 2, 4, 5	0, 1, 2, 4, 5, 3, 6
0, 1, 2, 4, 3	0, 1, 3, 2, 5, 4	0, 1, 2, 5, 3, 4, 6
0, 2, 1, 3, 4	0, 1, 2, 3, 4, 5	0, 1, 2, 5, 4, 3, 6
0, 2, 1, 4, 3	0, 1, 2, 3, 5, 4	0, 1, 5, 2, 3, 4, 6
	0, 1, 2, 4, 3, 5	0, 1, 5, 2, 4, 3, 6
	0, 1, 2, 4, 5, 3	
	0, 1, 2, 5, 3, 4	
	0, 1, 2, 5, 4, 3	

Figure 4.3 : Possible topological sorting results for three DAGS.

1. Topological Sorting Using DFS:
   - Call DFS and compute finish times
   - As each vertex is finished, insert in to the front of linked list.
   - Return the linked list
   - Time Complexity O(V+E)

# Cycles can be detected like normal DFS.



14	13	10	9	7	6	3
B	C	A	D	E	F	G

Python Implementation:
```
def get_prereq(G, vertex, s=1):
    state = [0] * (vertex + 1)
    stack = []
    for v in range(1, vertex + 1):
        if state[v] == 0:
            DFS(v, state, G, stack)
    while stack:
        output.write(f"{stack.pop(-1)} ")


def DFS(s, state, G, stack, parent=-1):
    state[s] = 1
    for v in G[s]:
        if state[v] == 0:
            DFS(v, state, G, stack, s)
        elif state[v] == 1 and parent != v:
            output.write("IMPOSSIBLE")  # Cycle detected
            exit()
    stack.append(s)
    state[s] = 2
```
2. Kahn's Algorithm:

- Create an indegree array with vertexes and corresponding indegree of the vertex. [(indegree1, vertex1), (indegree1, vertex1)].
- Append the vertex with indegree 0 in to a Priority Queue.
- Now run a loop while the Queue is not empty. Pop the vertex with the least indegree and decrease the indegrees of adjacent vertexes by 1 and append them to the priority queue.
- When each vertex is popped append them in a list (that's the ordering part).
- Time Complexity: O(V+E)

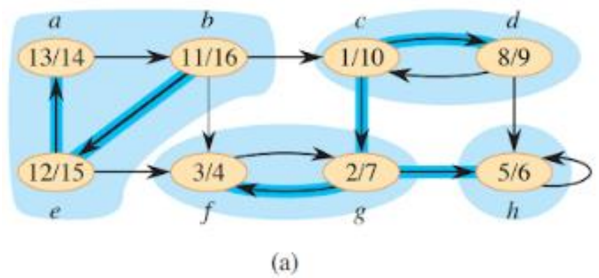
# To see if there is a cycle or not, just check if after running the algorithm the sum of indegree is zero or not. if not, then there is a cycle.

Python Implementation:

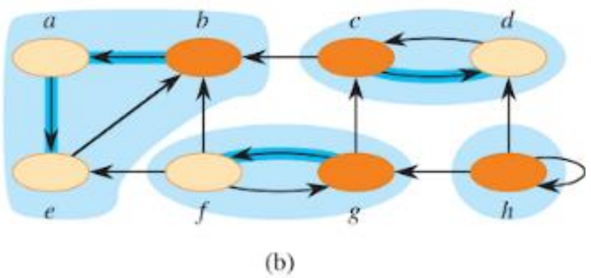
```python
def get_prereq(G, indegree):
    out = ""
    stack = PriorityQueue()
    for v in range(1, vertex + 1):
        if indegree[v] == 0:
            stack.put(v)
    while not stack.empty():
        v = stack.get()
        out += f"{v} "
        for u in G[v]:
            indegree[u] -= 1
            if indegree[u] == 0:
                stack.put(u)
    if sum(indegree) > 0:  # Cycle detected
        return "IMPOSSIBLE"

    return out
```

# Strongly Connected Components (Kosaraju's Algorithm)
Strongly Connected Components are the set of vertices such that each vertex can be visited from the other vertex.

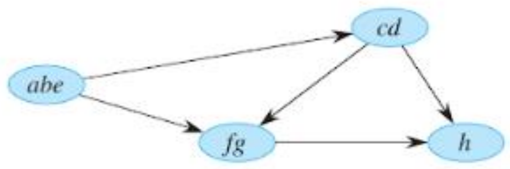
- Call DFS(G) to compute finish times (Get topologically sorted vertices)
- Call DFS on GT→ Transpose of G.
- Each time the DFS stops append the visited vertices in that run as one component and run DFS again in order of topological order.
- Output the vertices of each tree as separate strongly connected component
- Time Complexity O(V+E)



(a)



(b)

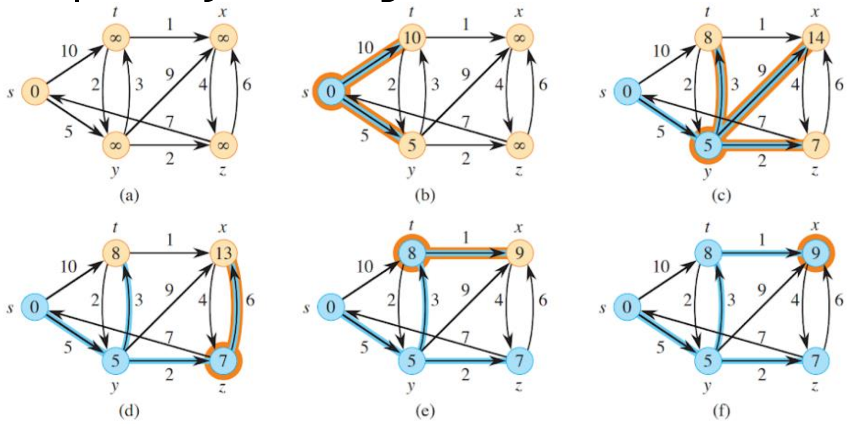


Python Implementation:

```
def dfs(s, graph, state, s1=[]):
    state[s] = 1
    for v in graph[s]:
        if state[v] == 0:
            dfs(v, graph, state, s1)
    s1.append(s)
    return s1


def get_strongly_connected_components(graph, graphT,
vertex):
    components = []
    state = [0] * (vertex + 1)
    sorted_vertices = []
    for i in range(1, vertex + 1):
        if state[i] == 0:
            sorted_vertices = dfs(i, graph, state,
sorted_vertices)
    state = [0] * (vertex + 1)
    while sorted_vertices:
        v = sorted_vertices.pop(-1)
        if state[v] == 0:
            component = dfs(v, graphT, state, [])
            components.append(component)
    return components
```

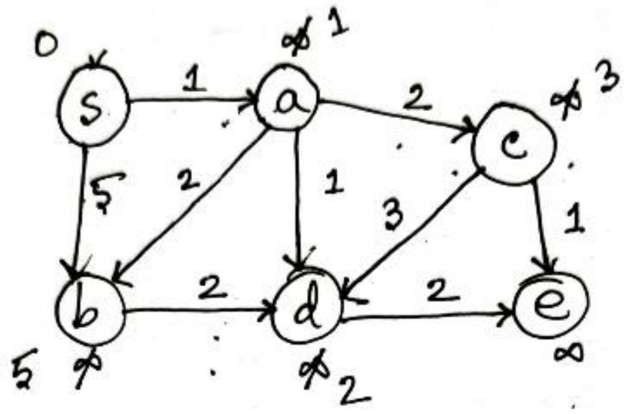
Single Source Shortest Path Algorithm

1. Djikstra Algorithm:
   - Djikstra won't work on negative weighted graphs.
   - The Graph has to be a DAG with no negative cycle.
   - Time Complexity O(E.logV)



Python Implementation:

```
def dijkstra(G, s, vertex):
    # Creating Distance and parent lists
    distance = [float('inf') for _ in range(vertex + 1)]
    pi = [None for _ in range(vertex + 1)]
    # Set distance of source to 0 and
    distance[s] = 0
    Q = []
    heapq.heappush(Q, (distance[s], s))

    while Q:
        u = Q.pop(0)[1]
        for v, d in G[u]:
            if distance[v] > distance[u] + d:
                distance[v] = distance[u] + d
                pi[v] = u
                heapq.heappush(Q, (distance[v], v))
    return distance, pi
```



$\boxed{d}$	s	a	b	c	d	e
	0	~~∞~~ 1	~~∞~~ ~~5~~ 3	~~∞~~ 3	~~∞~~ 2	~~∞~~ 4

$\boxed{\Pi}$	s	a	b	c	d	e
	nil	~~nil~~ s	~~nil~~ a	~~nil~~ a	~~nil~~ a	~~nil~~ d

priority queue

s → popped

a → popped

d → popped

c → popped

b → popped

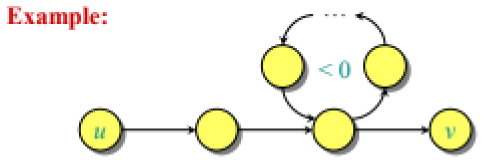
e → popped

$\boxed{d}$: 0, 1, 3, 3, 2, 4

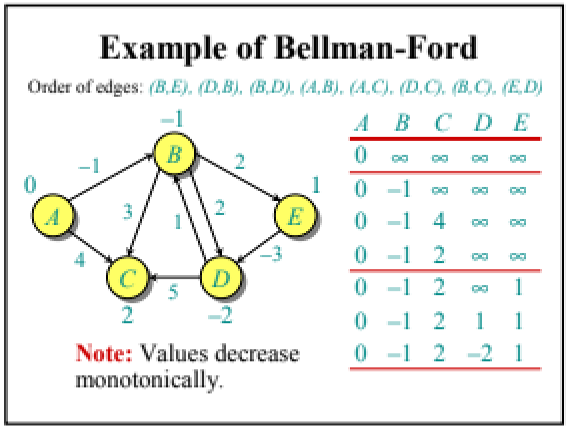
$\boxed{\Pi}$: nil , s, a, a, a, d.

2. Bellman Ford Algorithm:

- Works on negative weight graphs.
- Negative-weight cycles:
  Recall: If a graph G = (V, E) contains a negative- weight cycle, then some shortest paths may not exist.
  Example:



  Bellman-Ford algorithm: Finds all shortest-path lengths from a source s ∈ V to all v ∈ V or determines that a negative-weight cycle exists.
- Bellman-Ford algorithm is designed for directed graphs.



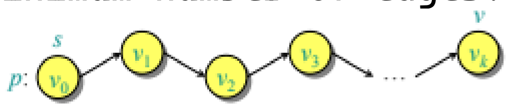
Example of Bellman-Ford

Order of edges: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)

A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

Note: Values decrease monotonically.

- Correctness Theorem: If G = (V, E) contains no negative-weight cycles, then after the Bellman-Ford algorithm executes, d[v] = δ(s, v) for all v ∈ V. Proof. Let v ∈ V be any vertex, and consider a shortest path p from s to v with the minimum number of edges.



Initially, d[v] = 0 = δ (s, v), and d[s] is unchanged by subsequent relaxations (because of the lemma from last lecture that d[v]≥δ (s, v) and δ (s, s) ≥0.

- After 1 pass through E, we have d [v] = δ (s, v).
- After 2 passes through E, we have d [v] = δ (s, v).
- After k passes through E, we have d [v] = δ (s, v).

Since G contains no negative-weight cycles, p is simple. Longest simple path has ≤ |V| − 1 edges.

- Detection of negative-weight cycles Corollary. If a value d[v] fails to converge after |V| − 1 passes, there exists a negative-weight cycle in G reachable from s.
- Time Complexity O(V.E).

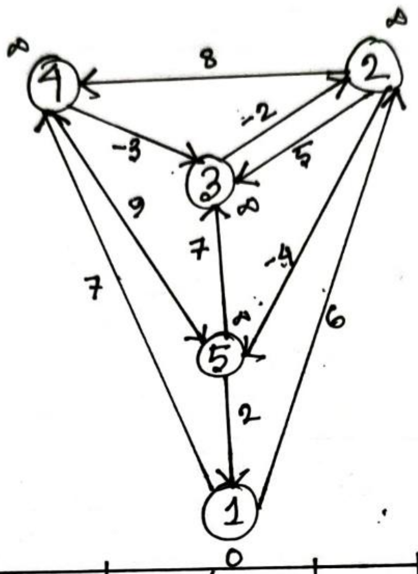
Python Implementation

```python
def bellman_ford(G, s):
    # Initialize distances and predecessors
    d = {v: float('inf') for v in G}
    pi = {v: None for v in G}
    d[s] = 0
    # Relax edges repeatedly
    for _ in range(len(G) - 1):
        for u in G:
            for v in G[u]:
                if d[v] > d[u] + G[u][v]:
                    d[v] = d[u] + G[u][v]
                    pi[v] = u
    # Check for negative-weight cycles
    for u in G:
        for v in G[u]:
            if d[v] > d[u] + G[u][v]:
                return "Negative-weight cycle detected"
    return d, pi
```

Shortest Path Decision Table

Graph Criteria	BFS $O(V+E)$	Dijkstra's $O((V+E)\log V)$	Bellman-Ford $O(VE)$	Floyd-Warshall $O(V^3)$
Max Size	$V+E \leq 100M$	$V+E \leq 1M$	$VE \leq 100M$	$V \leq 450$
Unweighted	Best	Ok	Bad	Bad in general
Weighted	WA	Best	Ok	Bad in general
Negative weight	WA	Modified Ok	Ok	Bad in general
Negative cycle	Cannot detect	Cannot detect	Can detect	Can detect
Small graph	WA if weighted	Overkill	Overkill	Best

Shortest Paths Algorithm Decision Table



Edge list

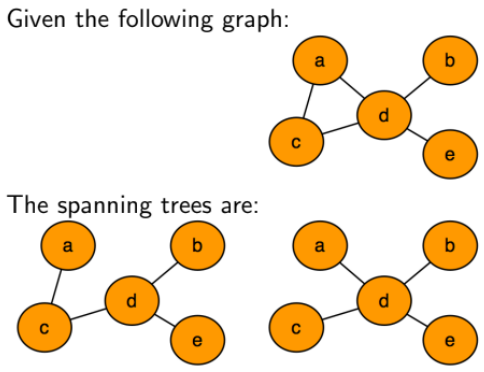
[(1, 2),
(1, 4),
(2, 3),
(2, 5),
(2, 4),
(3, 2),
(4, 3),
(4, 5),
(5, 3),
(5, 1)]

	1	2	3	4	5
1st iteration	0	$\cancel{\infty}$ 6^{1}	$\cancel{\infty}$ $\cancel{11}^{2}$ 4^{4}	$\cancel{\infty}$ 7^{1}	$\cancel{\infty}$ 2^{2}
2nd	0	$\cancel{6}^{1}$ 2^{3}	4^{4}	7^{1}	2^{2}
3rd	0	2^{3}	4^{4}	7^{1}	$\cancel{2}^{2}$ -2^{2}
4th	0	2^{3}	4^{4}	7^{1}	-2^{2}

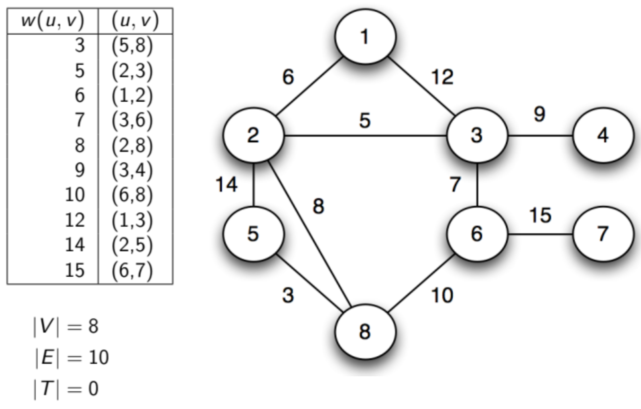
	1	2	3	4	5
d =	0	2	4	7	-2
π =	Nil	3	4	1	2

Minimum Spanning Tree (MST)

- A subgraph T of an undirected graph G = (V, E) is a spanning tree of G if it is a tree and contains every vertex of G.
- The minimum-cost spanning tree of a graph A spanning tree T of a undirected graph G = (V, E) is a minimum-cost spanning tree of G if the total weight, $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.



1. Prim's Algorithm:
   ##
2. Kruskal's Algorithm:
   - Parent of each node is set to itself.
   - The edges are sorted in order of increasing weight.
   - Then for each edge if the vertexes in that edge have different parent the edge is taken and their parents are set to be the same. Path compression can be used to speed up this process.
   - This runs until all the vertexes are included.
   - Time Complexity: O(ElgE)

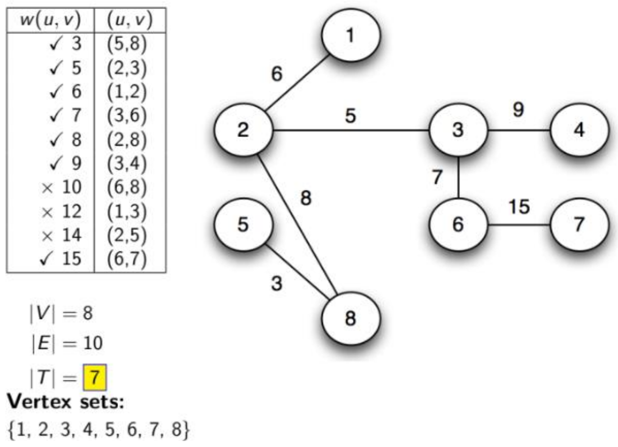


$w(u,v)$	(u,v)
3	(5,8)
5	(2,3)
6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)

$|V| = 8$

$|E| = 10$

$|T| = 0$



$w(u,v)$	(u,v)
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
✓ 9	(3,4)
× 10	(6,8)
× 12	(1,3)
× 14	(2,5)
✓ 15	(6,7)

$|V| = 8$

$|E| = 10$

$|T| = 7$

Vertex sets:

{1, 2, 3, 4, 5, 6, 7, 8}

Python Implementation:

```python
def MST(n, edges):
    parent = [i for i in range(n + 1)]
    edges.sort(key=lambda x: x[2])
    size = [1 for i in range(n + 1)]
    c = 0
    total_cost = 0
    for u, v, w in edges:
        if find_parent(parent, u) != find_parent(parent,
v):
            c += 1
            union(size, parent, u, v)
            total_cost += w
            if c == n - 1:
                return total_cost
    return total_cost


def find_parent(parent, i):
    if parent[i] == i:
        return i
    parent[i] = find_parent(parent, parent[i])
    return parent[i]


def union(size, parent, u, v):
    parent_u = find_parent(parent, u)
    parent_v = find_parent(parent, v)
    if size[parent_u] > size[parent_v]:
        parent[parent_v] = parent_u
    else:
        parent[parent_u] = parent_v
    total_size = size[parent_u] + size[parent_v]
    size[parent[u]] = total_size
```

Huffman Coding

1. Scan text to be compressed and tally occurrence of all characters.
2. Sort or prioritize characters based on number of occurrences in text and same freq→ based on appearance.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.

Eerie eyes seen near lake.

- What characters are present?

E e r i space

y s n a r l k .

Eerie eyes seen near lake.

- What is the frequency of each character in the text?

Char	Freq.	Char	Freq.	Char	Freq.
E	1	y	1	k	1
e	8	s	2	.	1
r	2	n	2		
i	1	a	2		
space	4	l	1		

- The queue after inserting all nodes

- Uses binary tree nodes

Building a Tree

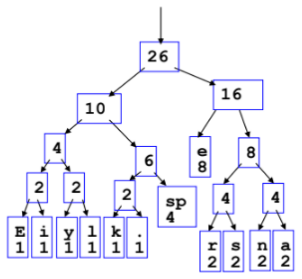
- While priority queue contains two or more nodes
  - Create new node
  - Dequeue node and make it left subtree
  - Dequeue next node and make it right subtree
  - Frequency of new node equals sum of frequency of left and right children
  - Enqueue new node back into queue

ght;

Encoding the File

Traverse Tree for Codes

Char	Code
E	0000
i	0001
y	0010
l	0011
k	0100
.	0101
space	011
e	10
r	1100
s	1101
n	1110
a	1111



Python Implementation:

```python
class HuffmanNode:
    ID = 0

    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
        HuffmanNode.ID += 1
        self.ID = HuffmanNode.ID

    def __lt__(self, other):
        if self.freq == other.freq:
            return self.ID < other.ID
        return self.freq < other.freq

def huffman_encoding(pq):
    while pq.qsize() > 1:
        left = pq.get()
        right = pq.get()
        new_node = HuffmanNode(None, left.freq + right.freq)
        new_node.left = left
        new_node.right = right
        pq.put(new_node)
    return pq.get()

def export_codes(node, prefix="", codes={}):
    if node.char is not None:
        codes[node.char] = prefix
        return
    export_codes(node.left, prefix + "0", codes)
    export_codes(node.right, prefix + "1", codes)
    return codes

def create_freq_dict(S):
    chars = defaultdict(int)
    for c in S:
        chars[c] += 1
    chars = sorted(chars.items(), key=lambda x: x[1])
    pq = PriorityQueue()
    for char, freq in chars:
        pq.put(HuffmanNode(char, freq))
    return pq

def decode_huffman(exported_codes, encoded):
    decoded = ""
    while encoded:
        for char, code in expored_codes.items():
            if encoded.startswith(code):
                decoded += char
                encoded = encoded[len(code):]
    return decoded
```

Dynamic Programming

- Build up the solution by computing solutions to the
- subproblems.
- Don't solve the same subproblem twice, but rather save the
- solution so it can be re-used later on.
- Often used for a large class to optimization problems.
- Unlike Greedy algorithms, implicitly solve all subproblems.
- Motivating the case for DP with Memoization – a top-down technique, and then moving on to Dynamic Programming – a bottom-up technique.

1. LCS :
   - O(nm)

	0	A	B	A	Z	D	C
0	0	0	0	0	0	0	0
B	0	↑ 0	↖ 1	← 1	← 1	← 1	← 1
A	0	↖ 1	↑ 1	↖ 2	← 2	← 2	← 2
C	0	- ↑ 1	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3
B	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 2	↑ 3
A	0	↖ 1	↑ 2	↖ 3	- ← 3	← 3	← 3
D	0	↑ 1	↑ 2	↑ 3	↑ 3	↖ 4	← 4

LCS: ABAD

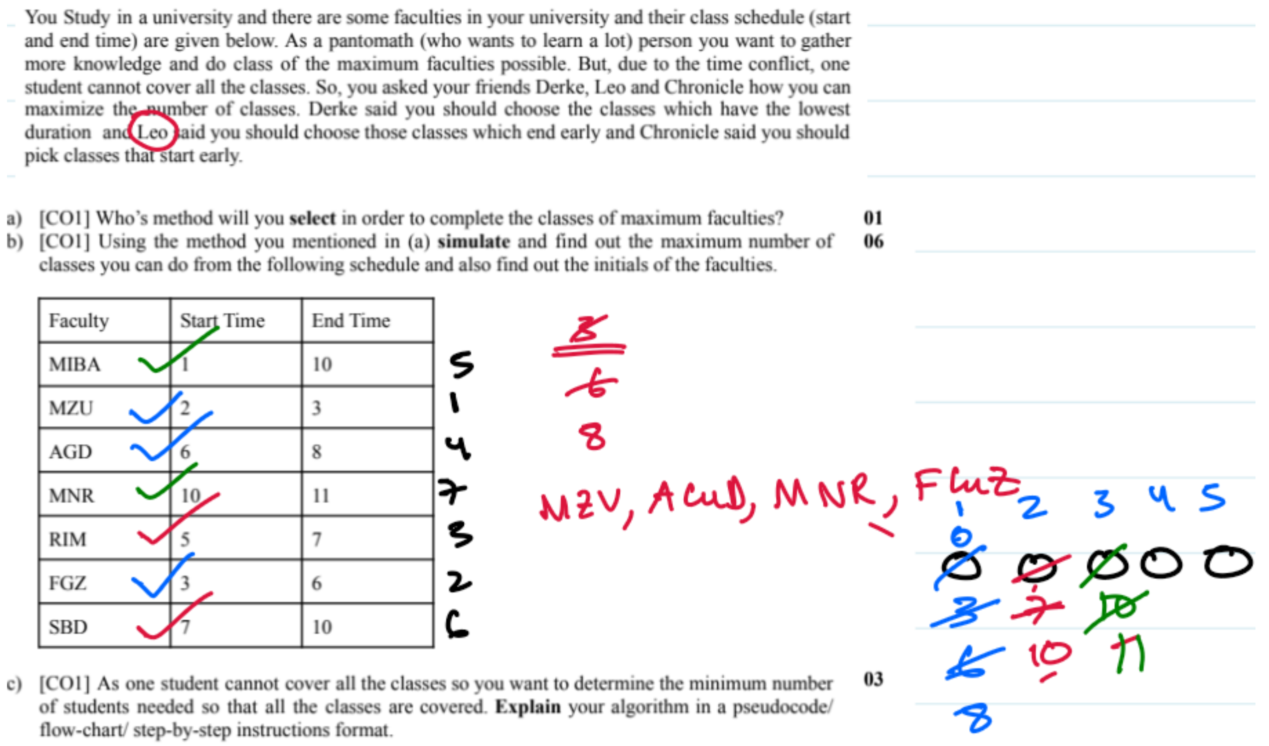
2. 0/1 KNAPSACK:

- O(nW)

Objects	Weight (Kg)	Profit ($)	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0	0	0
Jewelry	3	5	0	0	0	5	5	5	5	5	5
Sculpture	5	9	0	0	0	5	5	9	9	9	14
Painting	4	5	0	0	0	5	5	9	9	10	14
Book	1	4	0	4	4	5	9	9	13	13	14
Mummy	12	6	0	4	4	5	9	9	13	13	14
MAX WEIGHT	8		So:	Jewellery and Sculpture							

Greedy

1. Time Scheduling:



You Study in a university and there are some faculties in your university and their class schedule (start and end time) are given below. As a pantomath (who wants to learn a lot) person you want to gather more knowledge and do class of the maximum faculties possible. But, due to the time conflict, one student cannot cover all the classes. So, you asked your friends Derke, Leo and Chronicle how you can maximize the number of classes. Derke said you should choose the classes which have the lowest duration and Leo said you should choose those classes which end early and Chronicle said you should pick classes that start early.

a) [CO1] Who's method will you **select** in order to complete the classes of maximum faculties? 01

b) [CO1] Using the method you mentioned in (a) **simulate** and find out the maximum number of classes you can do from the following schedule and also find out the initials of the faculties. 06

Faculty	Start Time	End Time	
MIBA ✓	1	10	5
MZU ✓	2	3	1
AGD ✓	6	8	4
MNR ✓	10	11	7
RIM ✓	5	7	3
FGZ ✓	3	6	2
SBD ✓	7	10	6

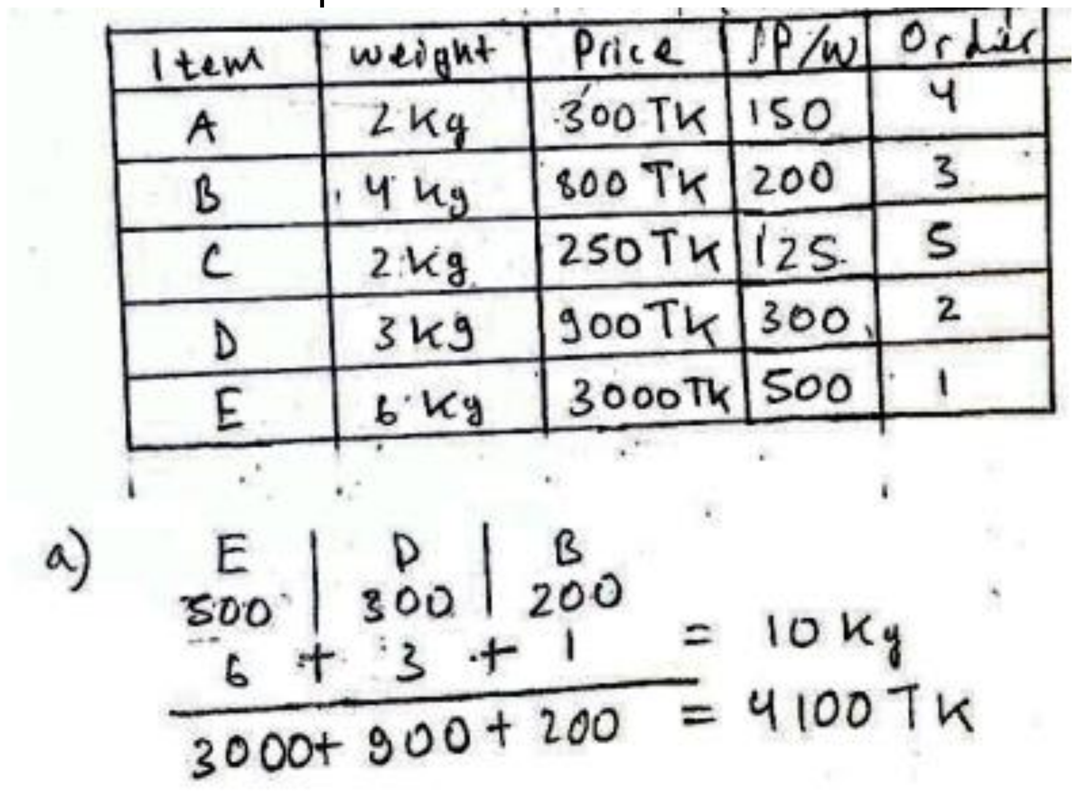
~~8~~
~~6~~
8

MZU, AGD, MNR, FGZ

1 2 3 4 5
0 0 0 0 0
~~3~~ ~~7~~ ~~10~~
~~6~~ 10 11
8

c) [CO1] As one student cannot cover all the classes so you want to determine the minimum number of students needed so that all the classes are covered. **Explain** your algorithm in a pseudocode/ flow-chart/ step-by-step instructions format. 03

2. Fractional Knapsack:



Item	weight	Price	P/W	Order
A	2 kg	300 Tk	150	4
B	4 kg	800 Tk	200	3
C	2 kg	250 Tk	125	5
D	3 kg	900 Tk	300	2
E	6 kg	3000 Tk	500	1

a)
$$\begin{array}{c|c|c} E & D & B \\ 500 & 300 & 200 \end{array}$$
$$6 + 3 + 1 = 10 \text{ kg}$$
$$3000 + 900 + 200 = 4100 \text{ Tk}$$