Assignment N*5

Name: Harlee Ramos **Due Date:** 07/21/2024

1. **In Lecture 13, we solved the Fractional Knapsack problem (Example 2) using a greedy algorithm ("iteratively choosing the item with largest unit value") and we proved it can yield an optimal solution. Describe this greedy algorithm formally in pseudo-code.**

The following pseudocode, FRACTIONAL_KNAPSACK, implements the greedy algorithm approach of iteratively selecting the item with the highest unit value. It accepts four arguments: the number of items (n), knapsack capacity (B), a list of weights (s), and a list of values (v). The pseudocode begins with the construction of a list of items, each of which is a tuple holding the value, weight, and value-to-weight ratio.

On line 2, the function GET_RATIO is used to compute the value-to-weight ratio for each item tuple. The MERGE_SORT method uses this ratio to order the items.Line 3 uses the merge sort to sort the elements in decreasing order depending on their value-to-weight ratio. MERGE (A, p, q, r): Merges two sorted subarrays of A with MERGE_SORT(A, p, r): Recursively divides the array into subarrays, sorts them, and then merges them back together.

Lines 4–13: Iterate through the sorted list of objects, adding whole or fractional items until the knapsack is full or all items are considered. The operation ends at line 14 with the return of the knapsack's total value.

MERGE(A, p, q, r)

```
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
  L[i] = A[p + i - 1]
for j = 1 to n2
  R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
  if L[i] <= R[j]
    A[k] = L[i]
    i = i + 1
else A[k] = R[j]
    j = j + 1
```

MERGE_SORT(A, p, r)

```
if p < r
  q = (p + r) / 2
  MERGE-SORT(A, p, q)
  MERGE-SORT(A, q + 1, r)
  MERGE(A, p, q, r)
```

GET_RATIO(item)

```
Define function GET_RATIO(item)
return item[2] // item[2] is the value-to-weight ratio
```

FRACTIONAL_KNAPSACK(n, B, s, v)

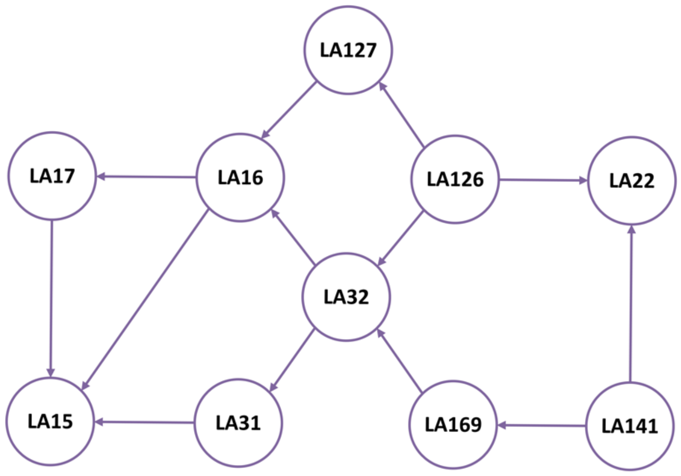
```
1. items = [(v[i], s[i], v[i] / s[i]) for i in range(n)]        // Create list of items with (value, weight, ratio)
2. Define a function to extract the ratios: GET_RATIO(item) -> item[2]
3. MERGE_SORT(items, 0, n - 1) using get_ratio to compare the ratios
4. total_value = 0
5. remaining_capacity = B
6. for i in range(n):
7.     if items[i][1] <= remaining_capacity:            // items[i][1] is the weight
8.         total_value = total_value + items[i][0]              // items[i][0] is the value
9.         remaining_capacity = remaining_capacity - items[i][1]
10.    else:
11.        fraction = remaining_capacity / items[i][1]
12.        total_value = total_value + items[i][0] * fraction
13.        break
14. return total_value
```

Time Complexity Analysis: Sorting the items using MERGE_SORT takes O(nlogn), most of the operations of FRACTIONAL_KNAPSACK consist of Creating the list of items and iterating through them takes a time of O(n). then considering the weight of the times, this pseudocode ensures a time complexity of O(nlogn), because an efficient sorting algorithm is used to sort the items by their value-to-weight ratio before applying the greedy strategy to solve the Fractional Knapsack problem.

2. Bob loves foreign languages and plans to take one language course each semester. He wants to plan his course schedule to take the following 10 language courses. Find a sequence of courses that allows Bob to satisfy all the prerequisites and show how you find this sequence.

Course	Prerequisites
$LA15$	None
$LA16$	$LA15$, $LA17$
$LA17$	$LA15$
$LA22$	None
$LA31$	$LA15$
$LA32$	$LA16$, $LA31$
$LA126$	$LA22$, $LA32$, $LA127$
$LA127$	$LA16$
$LA141$	$LA22$, $LA169$
$LA169$	$LA32$

A directed graph, considering all courses and their prerequisites, could depict the problem's situation as follows:



To find the solution to this problem, I will explain the steps needed to reach the sequence of courses.

1. **Construction of adjacency matrix.**

In the following pseudocode, GENERATE-ADJACENCY-MATRIX, we take as arguments a dictionary where each key is a course and the value is a list of prerequisite courses, and the quantity of all courses/nodes (n); then, those arguments must be represented by each course as a node and each prerequisite relationship as a directed edge from the prerequisite course to the dependent course. The objective of the function is to return a conversion of the prerequisites dictionary into an adjacency matrix A.

```
GENERATE-ADJACENCY-MATRIX(prerequisites, n)
let A be a new n x n matrix
for i from 0 to n-1
    for j from 0 to n-1
        A[i][j] = 0
for each (course, prereqs) in prerequisites
    for each prereq in prereqs
        A[course][prereq] = 1
return A
```

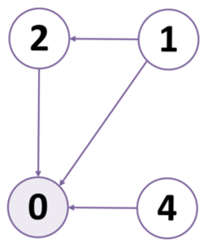
Calculations

Creation of the table, every spot is fill with 0.

```
for i from 0 to n-1
for j from 0 to n-1
A[i][j] = 0
```

j / i		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	0	0	0	0	0	0	0	0	0	0
2	***LA17***	0	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	0	0	0	0	0	0	0	0	0	0
5	***LA32***	0	0	0	0	0	0	0	0	0	0
6	***LA126***	0	0	0	0	0	0	0	0	0	0
7	***LA127***	0	0	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	0	0	0	0	0	0	0
9	***LA169***	0	0	0	0	0	0	0	0	0	0

0 row

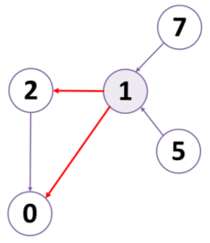


***At this point there no prerequisites/edges.**

A[0][0…9] = 0 at each spot,
Remains unchanged.

course / prereq		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	0	0	0	0	0	0	0	0	0	0
2	***LA17***	0	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	0	0	0	0	0	0	0	0	0	0
5	***LA32***	0	0	0	0	0	0	0	0	0	0
6	***LA126***	0	0	0	0	0	0	0	0	0	0
7	***LA127***	0	0	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	0	0	0	0	0	0	0
9	***LA169***	0	0	0	0	0	0	0	0	0	0

1st row

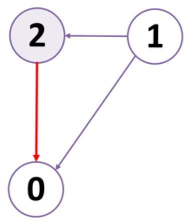


***At this point there exists some prerequisites/edges.**

```
for each (course, prereqs) in prerequisites
    for each prereq in prereqs
        A[course][prereq] = 1
        A[0][1] = 1
        A[2][1] = 1
```

prereq / course		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	0	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	0	0	0	0	0	0	0	0	0	0
5	***LA32***	0	0	0	0	0	0	0	0	0	0
6	***LA126***	0	0	0	0	0	0	0	0	0	0
7	***LA127***	0	0	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	0	0	0	0	0	0	0
9	***LA169***	0	0	0	0	0	0	0	0	0	0

2nd row

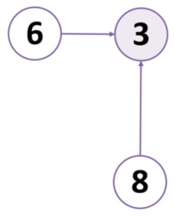


***At this point there exists some prerequisites/edges.**

```
for each (course, prereqs) in prerequisites
    for each prereq in prereqs
        A[course][prereq] = 1
        A[0][2] = 1
```

prereq / course		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	1	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	0	0	0	0	0	0	0	0	0	0
5	***LA32***	0	0	0	0	0	0	0	0	0	0
6	***LA126***	0	0	0	0	0	0	0	0	0	0
7	***LA127***	0	0	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	0	0	0	0	0	0	0
9	***LA169***	0	0	0	0	0	0	0	0	0	0

3rd row

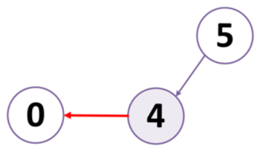


***At this point there no prerequisites/edges.**

0 at each spot,
Remains unchanged.

prereq / course		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	1	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	0	0	0	0	0	0	0	0	0	0
5	***LA32***	0	0	0	0	0	0	0	0	0	0
6	***LA126***	0	0	0	0	0	0	0	0	0	0
7	***LA127***	0	0	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	0	0	0	0	0	0	0
9	***LA169***	0	0	0	0	0	0	0	0	0	0

4th row

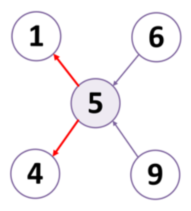


***At this point there exists some prerequisites/edges.**

```
for each (course, prereqs) in prerequisites
    for each prereq in prereqs
        A[course][prereq] = 1
        A[0][4] = 1
```

prereq / course		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	1	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	1	0	0	0	0	0	0	0	0	0
5	***LA32***	0	0	0	0	0	0	0	0	0	0
6	***LA126***	0	0	0	0	0	0	0	0	0	0
7	***LA127***	0	0	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	0	0	0	0	0	0	0
9	***LA169***	0	0	0	0	0	0	0	0	0	0

5th row

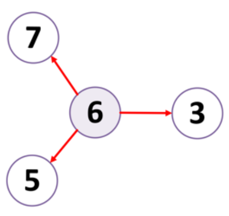


***At this point there exists some prerequisites/edges.**

```
for each (course, prereqs) in prerequisites
    for each prereq in prereqs
        A[course][prereq] = 1
A[1][5] = 1
A[4][5] = 1
```

prereq / course		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	1	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	1	0	0	0	0	0	0	0	0	0
5	***LA32***	0	1	0	0	1	0	0	0	0	0
6	***LA126***	0	0	0	0	0	0	0	0	0	0
7	***LA127***	0	0	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	0	0	0	0	0	0	0
9	***LA169***	0	0	0	0	0	0	0	0	0	0

6th row

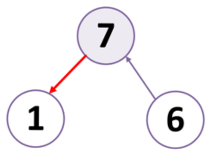


***At this point there exists some prerequisites/edges.**

```
for each (course, prereqs) in prerequisites
    for each prereq in prereqs
        A[course][prereq] = 1
A[3][6] = 1
A[5][6] = 1
A[7][6] = 1
```

prereq / course		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	1	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	1	0	0	0	0	0	0	0	0	0
5	***LA32***	0	1	0	0	1	0	0	0	0	0
6	***LA126***	0	0	0	1	0	1	0	1	0	0
7	***LA127***	0	0	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	0	0	0	0	0	0	0
9	***LA169***	0	0	0	0	0	0	0	0	0	0

7th row

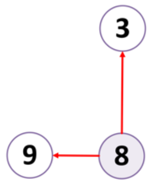


***At this point there exists some prerequisites/edges.**

```
for each (course, prereqs) in prerequisites
    for each prereq in prereqs
        A[course][prereq] = 1
A[1][7] = 1
```

prereq / course		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	1	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	1	0	0	0	0	0	0	0	0	0
5	***LA32***	0	1	0	0	1	0	0	0	0	0
6	***LA126***	0	0	0	1	0	1	0	1	0	0
7	***LA127***	0	1	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	0	0	0	0	0	0	0
9	***LA169***	0	0	0	0	0	0	0	0	0	0

8th row

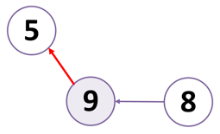


***At this point there exists some prerequisites/edges.**

```
for each (course, prereqs) in prerequisites
    for each prereq in prereqs
        A[course][prereq] = 1
A[3][8] = 1
A[9][8] = 1
```

prereq / course		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	1	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	1	0	0	0	0	0	0	0	0	0
5	***LA32***	0	1	0	0	1	0	0	0	0	0
6	***LA126***	0	0	0	1	0	1	0	1	0	0
7	***LA127***	0	1	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	1	0	0	0	0	0	1
9	***LA169***	0	0	0	0	0	0	0	0	0	0

9th row



***At this point there exists some prerequisites/edges.**

```
for each (course, prereqs) in prerequisites
    for each prereq in prereqs
        A[course][prereq] = 1
A[5][9] = 1
```

prereq / course		0	1	2	3	4	5	6	7	8	9
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	1	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	1	0	0	0	0	0	0	0	0	0
5	***LA32***	0	1	0	0	1	0	0	0	0	0
6	***LA126***	0	0	0	1	0	1	0	1	0	0
7	***LA127***	0	1	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	1	0	0	0	0	0	1
9	***LA169***	0	0	0	0	0	1	0	0	0	0

2. Construction of adjacency list:

To construct the adjacency list, we take the final matrix A from the last step of GENERATE-ADJACENCY-MATRIX and insert it at CONSTRUCT-ADJACENCY-LIST. This pseudocode will accept the A matrix as an argument, along with the n quantity of nodes (10 courses).

```
CONSTRUCT-ADJACENCY-LIST(A, n)
let adj_list be a new dictionary
for i from 0 to n-1
    adj_list[i] = []
for i from 0 to n-1
    for j from 0 to n-1
        if A[i][j] == 1
            adj_list[i].append(j)
return adj_list
```

Matrix A		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
		LA15	***LA16***	***LA17***	***LA22***	***LA31***	***LA32***	***LA126***	***LA127***	***LA141***	***LA169***
0	***LA15***	0	0	0	0	0	0	0	0	0	0
1	***LA16***	1	0	1	0	0	0	0	0	0	0
2	***LA17***	1	0	0	0	0	0	0	0	0	0
3	***LA22***	0	0	0	0	0	0	0	0	0	0
4	***LA31***	1	0	0	0	0	0	0	0	0	0
5	***LA32***	0	1	0	0	1	0	0	0	0	0
6	***LA126***	0	0	0	1	0	1	0	1	0	0
7	***LA127***	0	1	0	0	0	0	0	0	0	0
8	***LA141***	0	0	0	1	0	0	0	0	0	1
9	***LA169***	0	0	0	0	0	1	0	0	0	0

For better understand the pseudocode and the process, I will explain the calculations

The process begins with the initialization of an empty dictionary named adj_list. This dictionary will store the adjacency list representation of the graph. Each key is a node, and the value will be a list of nodes that the key node has edges to.

```
adj_list = {}
```

At line 2, there is a loop that iterates over each node in the graph. The variable i represents the index of the current node, ranging from 0 to n-1.

```
for i from 0 to n-1
adj_list[i] = []
```

*** At this point an empty list is created for each node:**

```
adj_list = {
 0: [],  // LA15
 1: [],  // LA16
 2: [],  // LA17
 3: [],  // LA22
 4: [],  // LA31
 5: [],  // LA32
 6: [],  // LA126
 7: [],  // LA127
 8: [],  // LA141
 9: []   // LA169
}
```

Matrix A's contents will fill the list from lines 4 through 7. The loop with i iterates over the nodes, and the loop with j iterates over the nodes.

The if condition checks if there is an edge from node i to node j by examining the value in the adjacency matrix A at position [i][j].

```
for i from 0 to n-1
    for j from 0 to n-1
        if A[i][j] == 1
```

If A[i][j] equals 1, it means there is a directed edge from i to j, then the j node number will we appended at the the by following this line adj_list[i].append(j).

Matrix		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A		***LA*15**	***LA*16**	***LA*17**	***LA*22**	***LA*31**	***LA*32**	***LA*126**	***LA*127**	***LA*141**	***LA*169**
0	***LA*15**	0	0	0	0	0	0	0	0	0	0
1	***LA*16**	1	0	1	0	0	0	0	0	0	0
2	***LA*17**	1	0	0	0	0	0	0	0	0	0
3	***LA*22**	0	0	0	0	0	0	0	0	0	0
4	***LA*31**	1	0	0	0	0	0	0	0	0	0
5	***LA*32**	0	1	0	0	1	0	0	0	0	0
6	***LA*126**	0	0	0	1	0	1	0	1	0	0
7	***LA*127**	0	1	0	0	0	0	0	0	0	0
8	***LA*141**	0	0	0	1	0	0	0	0	0	1
9	***LA*169**	0	0	0	0	0	1	0	0	0	0

At row 0, there is not 1. adj_list[0] = []

At the 1st row.
A[1][0] == 1 and A[1][2] == 1 adj_list[1] = [0, 2]

At 2nd row
A[2][0] == 1 adj_list[2] = [0]

At 3rd row, there is not 1. adj_list[3] = []

At the 4th row.
A[1][0] == 1. adj_list[4] = [0]

At the 5th row
A[5][1] == 1 and A[5][4] == 1 adj_list[5] = [1, 4]

At the 6th row
A[6][3] == 1 , A[6][5] == 1
A[6][7] == 1 adj_list[6] = [3,5,7]

At the 7th row
A[7][1] == 1 adj_list[7] = [1]

At the 8th row
A[8][3] == 1 , A[8][9] == 1 adj_list[8] = [3,9]

At the 9th row
A[9][5] == 1 adj_list[9] = [5]

The final adjacency list is the following

```
adj_list = {
 0: [],        // LA15
 1: [0,2],     // LA16
 2: [0],       // LA17
 3: [],        // LA22
 4: [0],       // LA31
 5: [1,4],     // LA32
6: [3,5,7],    // LA126
7: [1],        // LA127
8: [3,9],      // LA141
9: [5]         // LA169
}
```

3. Topological sort.

For the topological sort, the taken approach was via the implementation of depth-first search with the data structure stack. The stack operations are the following:

The STACK-EMPTY function takes as argument S: The stack. The function verifies whether stack S is empty. It returns TRUE if the stack is empty; otherwise, it returns FALSE.

```
STACK-EMPTY(S)
if S.top == 0
    return TRUE
else
    return FALSE
```

The PUSH function pushes an element x (in this case, a course index) onto the stack S. The PUSH function works by increasing the stack's top index by 1 and then repositioning element x to the new top position.

```
PUSH(S, x)
S.top = S.top + 1
S[S.top] = x
```

POP function pops the top element from Stack S. It checks if the stack is empty using STACK-EMPTY(s). If the stack is empty, raise an "underflow" error. Otherwise, decrement the stack's top index by 1. Finally, it returns the element that was at the top of the stack.

```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else
    S.top = S.top - 1
    return S[S.top + 1]
```

The pseudocode DFS-VISIT is based on the books with some modifications to use the stack. It takes as arguments a graph G, vertices u, color, and a stack S. The adj_list, which we calculated in the previous step, represents the graph. Graph theory commonly uses the notation G, u, v. We use u and v to represent vertices (or nodes) in the graph. This function ensures that every node is visited by checking if each node has been visited or not, calling DFS-VISIT for unvisited nodes, and then coloring the nodes (white is for unvisited, gray is for node exploration, and black is for explored and putted on stack nodes). This function effectively handles nodes without prerequisites.

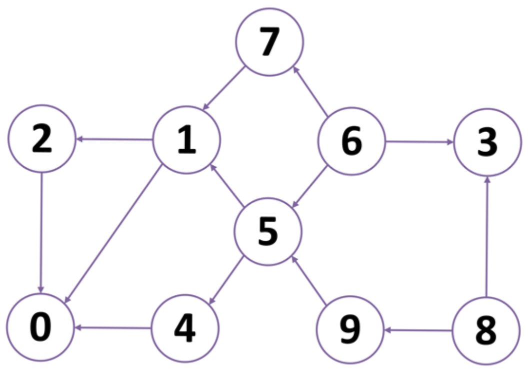
The DFS-TOPOLOGICAL-SORT pseudocode takes the adj_list = G as an argument and performs a linear ordering of a DAG graph, ensuring that for every directed edge u -> v, vertex u comes before vertex v in the ordering. In this version, we start by marking all nodes as white. Next, we establish a stack for storage. Moreover, our in-depth search ensures that lines 3 to 5 visit every node, even if the graph remains disconnected. Line 6 then creates a list to store the stack elements for subsequent pops. Finally, we reverse the list L to obtain the correct topological order. We added nodes to the stack postorder (after processing all descendants of a node), so reversing L gives the correct order from the start to the end.

```
DFS-VISIT(G, u, color, S)
color[u] = GRAY
for each v in G[u]
    if color[v] == WHITE
        DFS-VISIT(G, v, color, S)
color[u] = BLACK
PUSH(S, u)
```

```
DFS-TOPOLOGICAL-SORT(G)
1. let color be a dictionary with all keys in G
initialized to WHITE
2. let S be an empty stack
3. for each node u in G
4.     if color[u] == WHITE
5.         DFS-VISIT(G, u, color, S)
6. let L be an empty list
7. while not STACK-EMPTY(S)
8.     node = POP(S)
9.     append node to L
10. reverse L
11. return L
```

Explanation. The initialization for DFS-VISIT looks like this:

Same graph represented by their indexation number

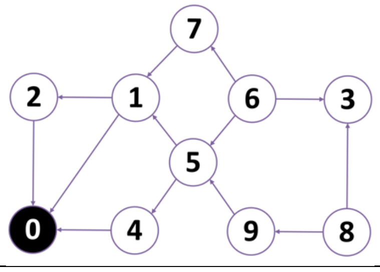


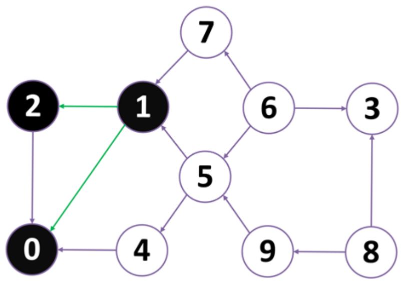
```
G = adj_list
adj_list = {
 0: [],           // LA15
 1: [0,2],        // LA16
 2: [0],          // LA17
 3: [],           // LA22
 4: [0],          // LA31
 5: [1,4],        // LA32
6: [3,5,7],      // LA126
7: [1],          // LA127
8: [3,9],        // LA141
9: [5]           // LA169
      }
```

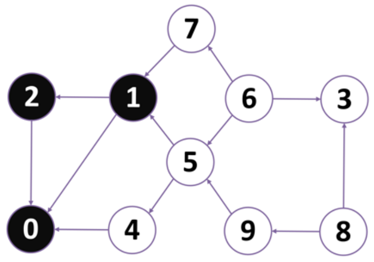
```
u current node

color = {
0: WHITE,
1: WHITE,
2: WHITE,
3: WHITE,
4: WHITE,
5: WHITE,
6: WHITE,
7: WHITE,
8: WHITE,
9: WHITE}

S = [] (empty stack)
```

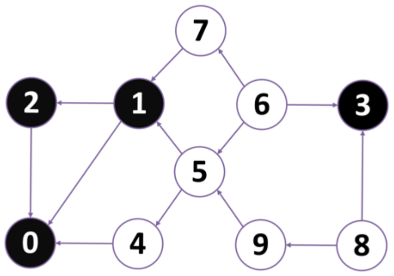
In the following section, I depicted each graph with the current state of the nodes to provide a more detailed description of how the pseudocode works.

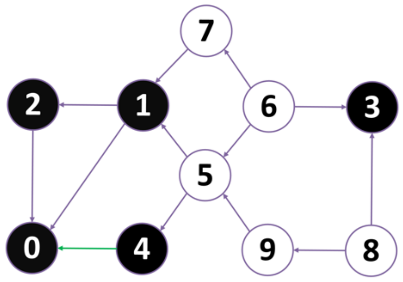
Node	Iterations

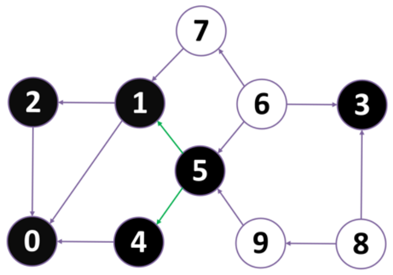
	For u = 0 (LA15):
	color[0] = WHITE, so call DFS-VISIT(0)
	color[0] = GRAY
	No adjacent nodes
	color[0] = BLACK
	PUSH(S, 0)
	Stack S = [0]

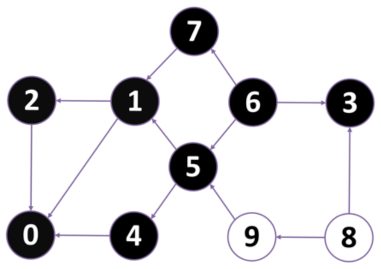
	For u = 1 (LA16):
	color[1] = WHITE, so call DFS-VISIT(1)
	color[1] = GRAY
	For v = 0, color is BLACK
	For v = 2, call DFS-VISIT(2)
	    color[2] = GRAY
	    For v = 0, color is BLACK
	    color[2] = BLACK
	    PUSH(S, 2)
	Stack S = [0, 2]
	color[1] = BLACK
	PUSH(S, 1)
	Stack S = [0, 2, 1]

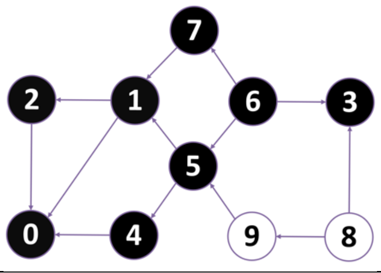
	For u = 2 (LA17):
	color[2] = BLACK (already visited)

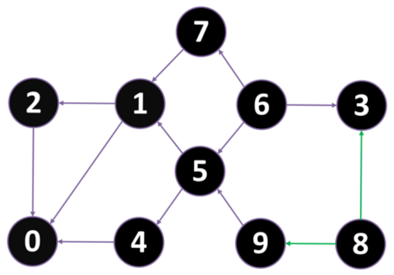
	For u = 3 (LA22):
	color[3] = WHITE, so call DFS-VISIT(3)
	color[3] = GRAY
	No adjacent nodes
	color[3] = BLACK
	PUSH(S, 3)
	Stack S = [0, 2, 1, 3]

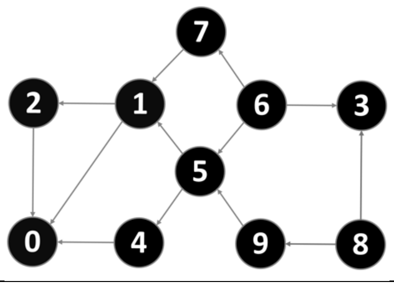
	For u = 4 (LA31):
	color[4] = WHITE, so call DFS-VISIT(4)
	color[4] = GRAY
	For v = 0, color is BLACK
	color[4] = BLACK
	PUSH(S, 4)
	Stack S = [0, 2, 1, 3, 4]

	For u = 5 (LA32):
	color[5] = WHITE, so call DFS-VISIT(5)
	color[5] = GRAY
	For v = 1, color is BLACK
	For v = 4, color is BLACK
	color[5] = BLACK
	PUSH(S, 5)
	Stack S = [0, 2, 1, 3, 4, 5]

	For u = 6 (LA126):
	color[6] = WHITE, so call DFS-VISIT(6)
	color[6] = GRAY
	For v = 3, color is BLACK
	For v = 5, color is BLACK
	For v = 7, call DFS-VISIT(7)
	    color[7] = GRAY
	    For v = 1, color is BLACK
	    color[7] = BLACK
	    PUSH(S, 7)
	Stack S = [0, 2, 1, 3, 4, 5, 7]
	color[6] = BLACK
	PUSH(S, 6)
	Stack S = [0, 2, 1, 3, 4, 5, 7, 6]

	For u = 7 (LA127):
	color[7] = BLACK (already visited)

	For u = 8 (LA141):
	color[8] = WHITE, so call DFS-VISIT(8)
	color[8] = GRAY
	For v = 3, color is BLACK
	For v = 9, call DFS-VISIT(9)
	    color[9] = GRAY
	    For v = 5, color is BLACK
	    color[9] = BLACK
	    PUSH(S, 9)
	Stack S = [0, 2, 1, 3, 4, 5, 7, 6, 9]
	color[8] = BLACK
	PUSH(S, 8)
	Stack S = [0, 2, 1, 3, 4, 5, 7, 6, 9, 8]

	For u = 9 (LA169):
	color[9] = BLACK (already visited)

```
Final Stack:
Stack S = [0, 2, 1, 3, 4, 5, 7, 6, 9, 8]
Initialize an empty list L = []
Pop Elements from Stack and Append to List:    S = []
L = [8, 7, 6, 3, 9, 5, 4, 1, 2, 0]
Reverse L to get the correct sequence   L = [0, 2, 1, 4, 5, 9, 3, 6, 7, 8]
```

4. Map the course names

The function MAP-INDICES-TO-COURSE-NAMES accepts a list of sorted indices (in this case is L) and a list of courses with their associated indices and returns a list of course names that correspond to the sorted indices. It compares each index in sorted_indices to the index in courses and extracts the relevant course titles.

MAP-INDICES-TO-COURSE-NAMES(L, courses)

```
1. let course_names be an empty list
2. for each index in L
3. for each (course, i) in courses
4. if i == index
5. append course to course_names
6. return course_names
```

Resulting sequence of courses that allows Bob to satisfy all the prerequisites:
course_names = [LA15, LA17, LA16, LA127, LA31, LA32, LA169, LA22, LA141, LA126]

3. **Given a simple undirected graph $G = (V, E)$, present an algorithm that determines whether G contains a cycle in $O(|V|)$ time. Analyze the time complexity of your algorithm to show why it finishes in the required running time.**

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v in G.Adj[u]
5      if v.color == WHITE
6          v.pi = u
7          DFS-Visit(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

```
DFS(G)
1  for each vertex u in G.V
2      u.color = WHITE
3      u.pi = NIL
4  time = 0
5  for each vertex u in G.V
6      if u.color == WHITE
7          DFS-Visit(G, u)
```

```
CONTAINS_CYCLE(G)
1  initialize color array of size |V| to WHITE
2  function DFS(u, parent, color):
3        color[u] = GRAY
4        for each v in adjacency list of u:
5            if color[v] == WHITE:
6                if DFS(v, u, color):
7                    return TRUE
8            else if v != parent and color[v] == GRAY:
9                return TRUE
10       color[u] = BLACK
11       return FALSE
12    for each node u in V:
13       if color[u] == WHITE:
14           if DFS(u, -1, color):
15               return TRUE
16    return FALSE
```

Time complexity analysis:

- **DFS-VISIT (G, u)** This is a helper function that visits a single node u and explores its neighbors recursively. This function accepts two arguments: a graph G and a node u. Its stats by incrementing time and assigns a discovery time to u before marking it as GRAY. This action takes O(1) time for each vertex.

  Moving to line four, for each vertex v adjacent to u: If v is WHITE, recursively invoke DFS-VISIT (G, v). The loop traverses all adjacent vertices of u. The total number of such iterations across all DFS-VISIT calls is O(|E|), as each edge is considered precisely once. This complexity includes all processes preceding and including line 7.

  However, on lines 8-10, the time complexity becomes O (1) for each vertex because, once all nearby vertices of u have been processed, u is marked as BLACK and the finishing time is set.

  The overall time complexity over all calls and taking the worst-case scenario is O(|V|+|E|). This is because each vertex u is visited exactly once, and all its surrounding vertices are considered precisely once.

- **DFS(G)** This function uses a recursive DFS to explore the graph G. It should maintain track of each vertex's parent to distinguish between a back edge (representing a cycle) and a tree edge.

  It begins by setting all vertices to WHITE color (u.color = WHITE) and u.pi to NIL. It takes O(|V|) time to process each vertex exactly once.

  The loop in line 5 passes through each vertex u in G.V. If u.color == WHITE, the method DFS-VISIT(G, u) will be called. The loop's body (lines 6-7) will call DFS-VISIT for each vertex, taking the same time complexity as the DSF-VISIT function O(|V|+|E|).

- **CONTAINS_CYCLE(G)** This function determines whether the graph G contains a cycle. It uses the DFS traversal approach to look for back edges, which are evidence of a cycle in the graph. If a back edge is identified, it indicates that the graph contains a cycle, and the method returns TRUE. If no back edges are identified (i.e., if a DFS traversal of the graph yields no back edges), the function returns FALSE, indicating that the graph lacks a cycle.

  The pseudocode begins by defining and initializing the color array to WHITE; this operation takes O(|V|).

  Lines 2-11 use the modified DFS function, like DFS-VISIT. Each vertex u is visited once, but all nearby vertices are also investigated, resulting in a total time complexity of O(|E|) iterations across all edges.

  Lines 12-16 include the pseudocode that will detect the cycle. This loop will pass through each vertex u. If u is WHITE, use the modified DFS function. In worst-case scenarios, these activities result in O(|V|) DFS calls as each vertex is only processed once.

The time complexity of CONTAINS_CYCLE(G) is O(|V|+|E|). This is due to the use of an array color of size |V| to control vertex states. This eliminates the need for separate arrays or flags for visited and exploration states. Each vertex is categorized as WHITE (unvisited), GRAY (currently being explored), or BLACK (fully processed). Furthermore, the approach discovers cycles in an undirected graph by utilizing the DFS traversal and looking for back edges.

The pseudocode uses a standard cycle detection method, but it's not possible to find cycles in a simple undirected graph in O(|V|) time because it requires examining all edges. This is because any cycle detection algorithm must inspect all edges in the graph to ensure proper detection. In a connected undirected graph G=(V,E), the number of edges varies from at least |V|-1 (in a tree structure and forest) to up to |V|(|V|-1)/2 in a full graph. This means that any cycle detection method must inspect all edges at least once to verify that no back edges (indicators of cycles) are overlooked. This requires O(|E|) time. As a result, the time complexity of finding cycles is determined by both the number of edges and the number of vertices.

4. **Given a simple undirected graph $G = (V, E)$, present an algorithm that determines whether G is bipartite in $O(|V| + |E|)$ time. A graph $G = (V, E)$ is bipartite, by its definition (on page 1172 in the 3 rd edition of CLRS), if and only if V can be decomposed into L and R, where $|L| + |R| = |V|$ and $L \cap R = \emptyset$, such that for each edge $e \in E$, one endpoint is in L and the other is in R. In other words, $G = (V, E)$ is bipartite if and only if G contains no cycles of odd length.**

Queue functions

The ENQUEUE (Q, x) function, which assigns a value to an array at a specific index, is a constant-time operation. Next, it checks if two values are equal, which is also a constant-time operation. Lastly, the condition in line 2 determines the execution of lines 3 and 4, both of which are constant-time operations due to their use of basic arithmetic and assignment. This means that regardless of the size of the queue Q, the ENQUEUE (Q, x) function will take the same amount of time to execute, so it's a constant time O(1).

ENQUEUE(Q, x)
```
1  Q[Q.tail] = x
2 if Q.tail == Q.length
3    Q.tail = 1
4 else Q.tail = Q.tail + 1
```

The DEQUEUE(Q) function starts by accessing an element in an array at a specific index, which is a constant-time operation. Next, it checks if two values are equal, which is also a constant-time operation. Like ENQUE on Lines 3 and 4, the execution of either line depends on the condition in Line 2, and both involve basic arithmetic and assignment, making them constant-time operations. It concludes by returning a value, which is a constant-time O(1) operation.

DEQUEUE(Q)
```
1 x = Q[Q.head]
2 if Q.head == Q.length
3     Q.head = 1
4 else Q.head = Q.head + 1
5 return x
```

BFS-BIPARTITE: This function takes a graph G and a starting vertex s. It first initializes the starting vertex s by setting its color to GRAY (meaning it's visited but not fully explored), its distance d to 0, and its predecessor pi to NIL. Then it performs a BFS on the graph. For each vertex v adjacent to the current vertex u, if v is WHITE, it sets v's color to GRAY, its distance d to u.d + 1, its predecessor pi to u, and enqueues v. If v is GRAY and its distance d is equal to u.d, it means the graph contains an odd-length cycle, so it returns FALSE. After exploring all adjacent vertices of u, it sets u's color to BLACK (meaning it's fully explored). If no odd-length cycles are found, it returns TRUE.

IS-BIPARTITE: This function checks each component of the graph. For each vertex u in the graph G, if u is WHITE, it calls BFS-BIPARTITE(G, u). If BFS-BIPARTITE(G, u) returns FALSE, it means the graph is not bipartite, so it returns FALSE. If all components are bipartite, it returns TRUE.

BFS-BIPARTITE (G,s)
```
1. Initialize each vertex in the graph:
2   For each vertex u in G.V:
3       Set u.color to WHITE
4       Set u.d to INFINITY
5       Set u.pi to NIL
6  Set the starting vertex s:
7   Set s.color to GRAY
8   Set s.d to 0
9   Set s.pi to NIL
10 Initialize a queue Q with s:
11.  Q <- {s}
12 Perform BFS:
13   While Q is not empty:
14     DEQUEUE the first element u from Q
15      For each vertex v in G.Adj[u]:
16        If v.color is WHITE:
17          Set v.color to GRAY
18          Set v.d to u.d + 1
19          Set v.pi to u
20           ENQUEUE(Q, v)      // Add v to the queue Q
21         Else if v.color is GRAY:
22           If v.d is equal to u.d:
23           Return FALSE (graph contains an odd-length cycle)
24      Set u.color to BLACK
25 Return TRUE if no odd-length cycles are found:
26   Return TRUE
```

IS-BIPARTITE(G)
```
1 Initialize each vertex in the graph:
2   For each vertex u in G.V:
3       Set u.color to WHITE
4 Check each component of the graph:
5   For each vertex u in G.V:
6       If u.color is WHITE:
7           If BFS-BIPARTITE(G, u) returns FALSE:
8               Return FALSE
9 Return TRUE if all components are bipartite:
10   Return TRUE
```

IS-BIPARTITE() runs in O(|V|+|E|) time, where |V| is the number of vertices and |E| is the number of edges in the graph, because each vertex and each edge is visited exactly once. The algorithm uses a queue for the BFS, and the enqueue and dequeue operations are both O(1).