Challenge 2: Implement Depth First Search

After the BFS algorithm, we will now tackle the implementation for Depth First Search.

We'll cover the following

- Problem statement
  - Input
  - Output
  - Sample input
  - Sample output
- Coding exercise

Problem statement

You have to implement the **Depth First Search** algorithm on a directed graph using the data structures which we have implemented in the previous sections.

> **Note:** Your solution should work for both connected and unconnected graphs.

Input

A directed graph in the form of an adjacency list and a starting vertex

Output

A string containing the vertices of the graph listed in the correct order of traversal.

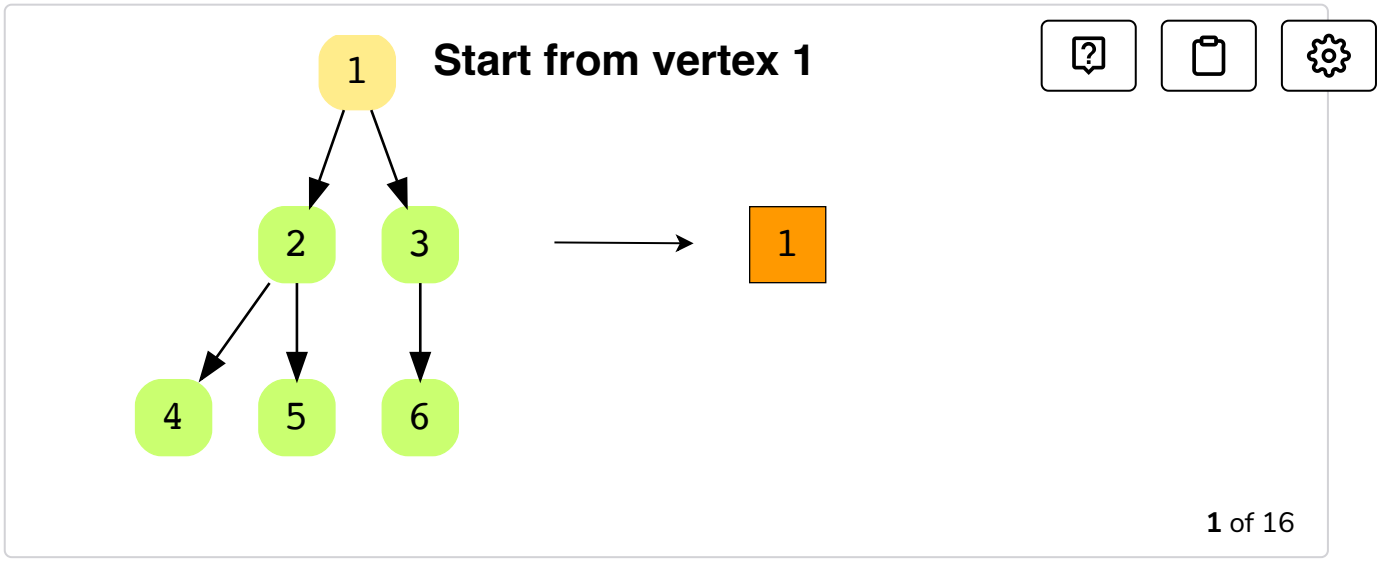
Sample input

Graph:

Vertex	Edges
0	None
1	3, 2
2	5, 4
3	6
4	None
5	None
6	None

Sample output

```
"1245360" or "1362540"
```



Coding exercise

Take a close look and design a step-by-step algorithm first before jumping on to the implementation. This problem is designed for your practice, so try to solve it on your own first. If you get stuck, you can always refer to the solution provided in the solution section.

Good luck!

main.py

Graph.py

Stack.py

Queue.py

LinkedList.py

Node.py

```
from Node import Node
```

```
class LinkedList:
    def __init__(self):
        self.head_node = None

    def get_head(self):
        return self.head_node

    def is_empty(self):
        if(self.head_node is None):  # Check whether the head is None
            return True
        else:
            return False

    def insert_at_head(self, dt):
        temp_node = Node(dt)
        if(self.is_empty()):
            self.head_node = temp_node
            return self.head_node
        temp_node.next_element = self.head_node
        self.head_node = temp_node
        return self.head_node

    # Inserts a value at the end of the list
    def insert_at_tail(self, value):
        # Creating a new node
        new_node = Node(value)
        # Check if the list is empty, if it is simply point head to new node
        if self.get_head() is None:
            self.head_node = new_node
            return
        # if list not empty, traverse the list to the last node
        temp = self.get_head()
        while temp.next_element is not None:
            temp = temp.next_element
        # Set the nextElement of the previous node to new node
        temp.next_element = new_node
        return

    def length(self):
        # start from the first element
        curr = self.get_head()
        length = 0
        # Traverse the list and count the number of nodes
        while curr is not None:
            length += 1
            curr = curr.next_element
        return length

    def print_list(self):
        if(self.is_empty()):
            print("List is Empty")
```

```
            return False
        temp = self.head_node
        while temp.next_element is not None:
            print(temp.data, end=" -> ")
            temp = temp.next_element
        print(temp.data, "-> None")
        return True

    def delete_at_head(self):
        # Get Head and firstElement of List
        first_element = self.get_head()
        # If List is not empty then link head to the
        # nextElement of firstElement.
        if (first_element is not None):
            self.head_node = first_element.next_element
            first_element.next_element = None
        return

    def delete(self, value):
        deleted = False
        if self.is_empty():  # Check if list is empty -> Return False
            print("List is Empty")
            return deleted
        current_node = self.get_head()  # Get current node
        previous_node = None  # Get previous node
        if current_node.data is value:
            self.delete_at_head()  # Use the previous function
            deleted = True
            return deleted
        # Traversing/Searching for Node to Delete
        while current_node is not None:
            # Node to delete is found
            if value is current_node.data:
                # previous node now points to next node
                previous_node.next_element = current_node.next_element
                current_node.next_element = None
                deleted = True
                break
            previous_node = current_node
            current_node = current_node.next_element
        return deleted

    def search(self, dt):
        if self.is_empty():
            print("List is Empty")
            return None
        temp = self.head_node
        while(temp is not None):
            if(temp.data is dt):
                return temp
            temp = temp.next_element
        print(dt, " is not in List!")
```

```
        return None

    def remove_duplicates(self):
        if self.is_empty():
            return
        # If list only has one node, leave it unchanged
        if self.get_head().next_element is None:
            return
        outer_node = self.get_head()
        while outer_node:
            inner_node = outer_node  # Iterator for the inner loop
            while inner_node:
                if inner_node.next_element:
                    if outer_node.data == inner_node.next_element.data:
                        # Duplicate found, so now removing it
                        new_next_element = inner_node.next_element.next_element
                        inner_node.next_element = new_next_element
                    else:
                        # Otherwise simply iterate ahead
                        inner_node = inner_node.next_element
                else:
                    # Otherwise simply iterate ahead
                    inner_node = inner_node.next_element
            outer_node = outer_node.next_element
        return
```


Back

Solution Review: Implement Breadth F...

Next

Solution Review: Implement Depth Fir...

Report an Is