ALGORITHM LIST

- **Must-have import**

```
from typing import List, Tuple, Callable, Union, Optional
from time import time
import numpy as np
```

- **Bubble Sort**

```
def BubbleSort(array: Union[List, Tuple, np.ndarray], reverse: bool = False):
    # Implementation of Bubble Sort from Right to Left
    for x in range(0, len(array):
        swapped = False
        for y in range(len(array) – 1, x, -1):
            if array[y - 1] > array[y]:
                array[y - 1], array[y], swapped = array[y], array[y - 1], True
            if swapped is False:
                break
    # Implementation of Bubble Sort from Left to Right
    swap: bool = True
    num_iter: int = 0
    while swap is True:
        swap = False
        for x in range(0, len(arr) - num_iter - 1):
            if arr[x] > arr[x + 1]:
                arr[x], arr[x + 1] = arr[x + 1], arr[x]
                swap = True
        num_iter += 1
```

- ➔ For order from maximum to minimum, change > by <.
- ➔ In-place sorting, Stable sorting
- ➔ Worst-case & Average Case: O(N^2), Best-case: O(N) for optimized version and O(N^2) for non-optimized version

- **Selection Sort**

```
def SelectionSort(array, reverse: bool = False):
    def search(arr, search_maximum: bool = False):
        index = 0
        if search_maximum is False:
            for i in range(1, len(arr)):
                if arr[i] < arr[index]:
                    index = i
        else:
            for i in range(1, len(arr)):
                if arr[i] > arr[index]:
                    index = i
```

```
            return index

    if reverse is False:
        for current_index in range(0, len(array)):
            index = current_index + search(arr=array[current_index:], search_maximum=False)
            if index != current_index:
                array[current_index], array[index] = array[index], array[current_index]
    else:
        for current_index in range(0, len(array)):
            index = current_index + search(arr=array[current_index:], search_maximum=True)
            if index != current_index:
                array[current_index], array[index] = array[index], array[current_index]
```

- ➔ In-place sorting, Stable sorting. Real-time Complexity: O(N*(N-1)/2)
- ➔ Worst-case & Average Case: O(N^2) with O(N) swap, Best-case: O(N^2) with O(1) swap.

- **Insertion Sort**

```
def InsertionSort(arr, reverse: bool = False):
    if reverse is False:
        for index in range(1, len(array)):
            if array[index - 1] < array[index]:
                continue
            for i in range(0, index):
                if array[i] > array[index]:
                    key = array[index]
                    array[i + 1: index + 1] = array[i:index]
                    array[i] = key
                    break
    else:
        for index in range(1, len(array)):
            if array[index - 1] > array[index]:
                continue
            for i in range(0, index):
                if array[i] < array[index]:
                    key = array[index]
                    array[i + 1: index + 1] = array[i:index]
                    array[i] = key
                    break
```

- ➔ In-place sorting, Stable sorting.
- ➔ Worst-case & Average Case: O(N^2) with O(N^2) swap, Best-case: O(N) with O(1) swap.

```
def BinarySearch(array, value, start:int=0, end: int = None, reverse: bool=False):
    if end is None:
```

```
        end = len(array)

    if reverse is False:
        if start == end:
            if array[start] > value:
                return start
            else:
                return start + 1
    else:
        if start == end:
            if array[start] < value:
                return start
            else:
                return start + 1

    if start > end:  # Ensuring position can be found
        return start

    mid = (start + end) // 2
    if reverse is False:
        if array[mid] < value:
            return BinaryIndexing(array=array, value=value, start=mid + 1, end=end, reverse=reverse)
        elif array[mid] > value:
            return BinaryIndexing(array=array, value=value, start=start, end=mid - 1, reverse=reverse)
        else:
            return mid
    else:
        if array[mid] > value:
            return BinaryIndexing(array=array, value=value, start=mid + 1, end=end, reverse=reverse)
        elif array[mid] < value:
            return BinaryIndexing(array=array, value=value, start=start, end=mid - 1, reverse=reverse)
        else:
            return mid

def BinaryInsertionSort(array, reverse: bool = False):
    if reverse is False:
        for index in range(1, len(array)):
            if array[index - 1] < array[index]:
                continue
            i = BinaryIndexing(array=array[0:index], value=array[index], reverse=reverse)
            key_point = array[index]
            array[i + 1:index + 1] = array[i:index]
            array[i] = key_point
```

```
    else:
        for index in range(1, len(array)):
            if array[index - 1] > array[index]:
                continue
            i = BinaryIndexing(array=array[0:index], value=array[index], reverse=reverse)
            key_point = array[index]
            array[i + 1:index + 1] = array[i:index]
            array[i] = key_point

def binarySearch(array, value):
    start, end = 0, len(array)
    while start <= end:
        mid = (start + end) // 2
        if value < array[mid]:
            end = mid
        elif value > array[mid]:
            start = mid
        else:
            return mid
    return -1
```

- Linked List Section

```
class Node:
    def __init__(self, value: Union[int, float]):
        self.next: Node = None
        self.prev: Node = None
        self.value = value

class LL:
    def __init__(self, dtype):
        if dtype == "LL":
            return LinkedList()
        elif dtype == "CLL":
            return CircularLinkedList()
        elif dtype == "DLL":
            return DoubleLinkedList()
        elif dtype == "DCLL":
            return DoubleCircularLinkedList()
        else:
            raise ValueError("False Value")

    def insertFirst(self, value: Union[int, float]):
        pass
```

```
    def append(self, value: Union[int, float]):
        pass

    def delete(self, value: Union[int, float], delete_all: bool = False):
        pass

    def setMultiValues(self, valueArray: Union[List[int], Tuple[int], List[float], Tuple[float]]):
        pass

    def build_array(self, verbose: bool = False):
        pass

    def display(self):
        return self.build_array(verbose=True)

    def display_status(self):
        pass

class LinkedList:
    def __init__(self):
        self.head = None
        self.num_of_node = 0

    def insertFirst(self, value: Union[int, float]):
        if not isinstance(value, (int, float)):
            warning(" Your value is not fit")

        node = Node(value=value)
        if self.head is not None:
            node.next = self.head
        self.head = node
        self.num_of_node += 1

    def append(self, value: Union[int, float]):
        if not isinstance(value, (int, float)):
            warning(" Your value is not fit")

        if self.head is None:
            self.insertFirst(value=value)
        else:
            node = Node(value=value)
            current_node = self.head
            while current_node.next is not None:
                current_node = current_node.next
```

```
        current_node.next = node
        self.num_of_node += 1

def setMultiValues(self, valueArray: Union[List[int], Tuple[int], List[float], Tuple[float]]):
    if len(valueArray) != self.num_of_node:
        raise ValueError("The Array ({}) is not fit compared to the "
                         "Linked List ({})".format(len(valueArray), self.num_of_node))

    current_node = self.head
    for value in valueArray:
        current_node.value = value
        current_node = current_node.next

def delete(self, value: Union[int, float], delete_all: bool = False):
    if not isinstance(value, (int, float)):
        warning(" Your value is not fit")

    if self.head is not None:
        if self.head.value == value:
            self.head = self.head.next
            self.num_of_node -= 1
            if self.head.value == value and delete_all is True:
                self.delete(value=value)
        else:
            current_node = self.head
            while current_node.next is not None:
                if current_node.next.value != value:
                    current_node = current_node.next
                elif current_node.next.value == value:
                    if delete_all is True:
                        current_node.next = current_node.next.next
                        self.num_of_node -= 1
                    else:
                        break
            if current_node.next is not None and delete_all is False:
                if current_node.next.value == value:
                    current_node.next = current_node.next.next
                    self.num_of_node -= 1

def build_array(self, verbose: bool = False):
    if self.head is None:
        return []

    stack = []
```

```
        current_node = self.head
        while current_node.next is not None:
            if verbose is True:
                print(current_node.value, end=" ")
            stack.append(current_node.value)
            current_node = current_node.next
        stack.append(current_node.value)
        if verbose is True:
            print(current_node.value, end="\n")
        return stack

    def display(self):
        return self.build_array(verbose=True)

    def display_status(self):
        current_node = self.head
        while current_node.next is not None:
            print(current_node.value, current_node.next)
            current_node = current_node.next
        print(current_node.value, current_node.next)
```

class CircularLinkedList:

```
    def __init__(self):
        self.head = None
        self.num_of_node = 0
        self.last_node: Node = None

    def insert(self, value: Union[int, float], insert_at_begin: bool = False):
        if not isinstance(value, (int, float)):
            warning(" Your value is not fit")

        node = Node(value=value)
        if self.head is None:
            self.head = node
            self.head.next = self.head
            self.last_node = node
        else:
            self.last_node.next = node
            node.next = self.head
            if insert_at_begin is True:
                self.head = node
            else:
                self.last_node = node
        self.num_of_node += 1
```

```
def insertFirst(self, value: Union[int, float]):
    self.insert(value=value, insert_at_begin=True)

def append(self, value: Union[int, float]):
    self.insert(value=value, insert_at_begin=False)

def setMultiValues(self, valueArray: Union[List[int], Tuple[int], List[float], Tuple[float]]):
    if len(valueArray) != self.num_of_node:
        raise ValueError("The Array ({}) is not fit compared to the "
                         "Linked List ({})".format(len(valueArray), self.num_of_node))

    current_node = self.head
    for value in valueArray:
        current_node.value = value
        current_node = current_node.next

def delete(self, value: Union[int, float], delete_all: bool = False):
    if not isinstance(value, (int, float)):
        warning(" Your value is not fit")

    if self.head is not None:
        if self.head.value == value:
            self.last_node.next = self.head.next
            self.head = self.head.next
            self.num_of_node -= 1
            if self.head.value == value and delete_all is True:
                self.delete(value=value)
        else:
            current_node = self.head
            while current_node.next != self.head:
                if current_node.next.value != value:
                    current_node = current_node.next
                elif current_node.next.value == value:
                    if delete_all is True:
                        current_node.next = current_node.next.next
                        self.num_of_node -= 1
                        if current_node.next == self.last_node:
                            self.last_node = current_node
                    else:
                        break
            if current_node.next is not self.head and delete_all is False:
                if current_node.next.value == value:
                    current_node.next = current_node.next.next
```

```
                        if current_node.next == self.last_node:
                            self.last_node = current_node
                        self.num_of_node -= 1

    def build_array(self, verbose: bool = False):
        if self.head is None:
            return []

        stack = []
        current_node = self.head
        while current_node.next != self.head:
            if verbose is True:
                print(current_node.value, end=" ")
            stack.append(current_node.value)
            current_node = current_node.next
        stack.append(current_node.value)
        if verbose is True:
            print(current_node.value, end="\n")
        if len(stack) != self.num_of_node:
            warning(" Something is wrong with the number of values in Linked List")
        return stack

    def display(self):
        return self.build_array(verbose=True)

    def display_status(self):
        current_node = self.head
        while current_node.next != self.head:
            print(current_node.value, current_node.next)
            current_node = current_node.next
        print(current_node.value, current_node.next)

class DoubleLinkedList:
    def __init__(self):
        self.head = None
        self.num_of_node = 0

    def insertFirst(self, value: Union[int, float]):
        if not isinstance(value, (int, float)):
            warning(" Your value is not fit")

        if self.head is None:
            self.head = Node(value=value)
        else:
```

```
        node = Node(value=value)
        self.head.prev = node
        node.next = self.head
        self.head = node
    self.num_of_node += 1

def append(self, value):
    if not isinstance(value, (int, float)):
        warning(" Your value is not fit")

    if self.head is None:
        self.insertFirst(value=value)
    else:
        node = Node(value=value)
        current_node = self.head
        while current_node.next is not None:
            current_node = current_node.next
        current_node.next = node
        node.prev = current_node
    self.num_of_node += 1

def delete(self, value, delete_all: bool = False):
    if not isinstance(value, (int, float)):
        warning(" Your value is not fit")

    if self.head is not None:
        if self.head.value == value:
            self.head = self.head.next
            self.head.prev = None
            self.num_of_node -= 1
            if self.head.value == value and delete_all is True:
                self.delete(value=value)
        else:
            current_node = self.head
            while current_node.next is not None:
                if current_node.next.value != value:
                    current_node = current_node.next
                elif current_node.next.value == value:
                    if delete_all is True:
                        current_node.next = current_node.next.next
                        if current_node.next is not None:
                            current_node.next.prev = current_node
                        self.num_of_node -= 1
                    else:
```

```
                break

            if current_node.next is not None and delete_all is False:
                if current_node.next.value == value:
                    current_node.next = current_node.next.next
                    if current_node.next is not None:
                        current_node.next.prev = current_node
                    self.num_of_node -= 1

    def setMultiValues(self, valueArray: Union[List[int], Tuple[int], List[float], Tuple[float]]):
        if len(valueArray) != self.num_of_node:
            raise ValueError("The Array ({}) is not fit compared to the "
                             "Linked List ({})".format(len(valueArray), self.num_of_node))

        current_node = self.head
        for value in valueArray:
            current_node.value = value
            current_node = current_node.next

    def build_array(self, verbose: bool = False):
        if self.head is None:
            return []

        stack = []
        current_node = self.head
        while current_node.next is not None:
            if verbose is True:
                print(current_node.value, end=" ")
            stack.append(current_node.value)
            current_node = current_node.next
        stack.append(current_node.value)
        if verbose is True:
            print(current_node.value, end="\n")
        return stack

    def display(self):
        return self.build_array(verbose=True)

    def display_status(self):
        current_node = self.head
        while current_node.next is not None:
            print("{}, next: {}, prev: {}".format(current_node.value, current_node.next, current_node.prev))
            current_node = current_node.next
        print("{}, next: {}, prev: {}".format(current_node.value, current_node.next, current_node.prev))
```

```python
class DoubleCircularLinkedList:
    def __init__(self):
        self.head = None
        self.num_of_node = 0

    def insert(self, value: Union[int, float], insert_at_begin: bool = False):
        if not isinstance(value, (int, float)):
            warning(" Your value is not fit")

        node = Node(value=value)
        if self.head is None:
            self.head = node
            self.head.next = self.head
            self.head.prev = self.head.next
        else:
            last_node = self.head.prev

            last_node.next = node
            node.prev = last_node

            node.next = self.head
            self.head.prev = node
            if insert_at_begin is True:
                self.head = node

        self.num_of_node += 1

    def insertFirst(self, value: Union[int, float]):
        self.insert(value=value, insert_at_begin=True)

    def append(self, value: Union[int, float]):
        self.insert(value=value, insert_at_begin=False)

    def setMultiValues(self, valueArray: Union[List[int], Tuple[int], List[float], Tuple[float]]):
        if len(valueArray) != self.num_of_node:
            raise ValueError("The Array ({}) is not fit compared to the "
                             "Linked List ({})".format(len(valueArray), self.num_of_node))

        current_node = self.head
        for value in valueArray:
            current_node.value = value
            current_node = current_node.next
```

```
def delete(self, value: Union[int, float], delete_all: bool = False):
    if not isinstance(value, (int, float)):
        warning(" Your value is not fit")

    if self.head is not None:
        if self.head.value == value:
            self.head.prev.next = self.head.next
            self.head.next.prev = self.head.prev
            self.head = self.head.next
            self.num_of_node -= 1
            if self.head.value == value and delete_all is True:
                self.delete(value=value)
        else:
            current_node = self.head
            while current_node.next != self.head:
                if current_node.next.value != value:
                    current_node = current_node.next
                elif current_node.next.value == value:
                    if delete_all is True:
                        current_node.next = current_node.next.next
                        current_node.next.prev = current_node
                        self.num_of_node -= 1
                    else:
                        break
            if delete_all is False:
                if current_node.next.value == value:
                    current_node.next = current_node.next.next
                    current_node.next.prev = current_node
                    self.num_of_node -= 1

def build_array(self, verbose: bool = False):
    if self.head is None:
        return []

    stack = []
    current_node = self.head
    while current_node.next != self.head:
        if verbose is True:
            print(current_node.value, end=" ")
        stack.append(current_node.value)
        current_node = current_node.next
    stack.append(current_node.value)
    if verbose is True:
        print(current_node.value, end="\n")
```

```
        if len(stack) != self.num_of_node:
            warning(" Something is wrong with the number of values in Linked List")
        return stack

    def display(self):
        return self.build_array(verbose=True)

    def display_status(self):
        current_node = self.head
        while current_node.next != self.head:
            print("{}, next: {}, prev: {}".format(current_node.value, current_node.next, current_node.prev))
            current_node = current_node.next
        print("{}, next: {}, prev: {}".format(current_node.value, current_node.next, current_node.prev))
```

Test Case:

```
if __name__ == '__main__':
    LL = DoubleCircularLinkedList()
    LL.insertFirst(value=100)
    LL.insertFirst(value=200)
    LL.insertFirst(value=5000)
    LL.insertFirst(value=1000)
    LL.insertFirst(value=2000)
    LL.insertFirst(value=7000)
    LL.insertFirst(value=3000)
    LL.insertFirst(value=3000)

    LL.display_status()
    print(LL.num_of_node)
    print()

    LL.delete(3000, delete_all=True)
    LL.display()
    print()

    LL.delete(300)
    LL.display()
    print()

    LL.delete(7000)
    LL.display()
    print()

    LL.append(1e5)
```

```
    LL.append(1e5)
    LL.display()

    LL.delete(value=1e5)
    LL.display()
    print()

    LL.delete(value=1e5)
    LL.display()
    print()

    LL.append(1234)
    LL.append(1234)
    LL.display()
    print()
    LL.delete(value=1234, delete_all=True)
    LL.display()
    print()
```

- Implement Queue & Stack from Array

```
class Queue:
    def __init__(self, max_size: int):
        if not isinstance(max_size, int):
            raise ValueError("The size should be an integer")
        if max_size <= 0:
            raise ValueError("The size should be a positive")
        self.data = [0] * max_size

        self.max_size: int = max_size
        self.front = 0 # head
        self.rear = 0 # tail

    def enqueue(self, value):
        if self.rear >= self.max_size:
            raise MemoryError(" No extra memory can be added")
        self.data[self.rear] = value
        self.rear += 1

    def dequeue(self):
        if self.front >= self.rear:
            raise IndexError(" No data is remained")
        self.front += 1
        return self.data[self.front - 1]
```

```
    def display(self):
        value = self.data[self.front:self.rear]
        for idx in range(self.front, self.rear):
            print(self.data[idx], end=" ")
        return value


class Stack:
    def __init__(self, max_size: int):
        if not isinstance(max_size, int):
            raise ValueError("The size should be an integer")
        if max_size <= 0:
            raise ValueError("The size should be a positive")
        self.data = [0] * max_size

        self.max_size: int = max_size
        self.tail = 0

    def append(self, value):
        if self.tail >= self.max_size:
            raise MemoryError(" No extra memory can be added")
        self.data[self.tail] = value
        self.tail += 1

    def pop(self):
        if self.tail <= 0:
            raise IndexError(" No data is remained")
        self.tail -= 1
        return self.data[self.tail]

class StackByQueue:
    def __init__(self, max_size: int, boost_adding: bool = True):
        if not isinstance(max_size, int):
            raise ValueError("The size should be an integer")
        if max_size <= 0:
            raise ValueError("The size should be a positive")

        self.main_data: Queue = Queue(max_size=max_size)
        self.temp_data: Queue = Queue(max_size=max_size)

        self.max_size: int = max_size
        self.current_size: int = 0
        self.__boost_adding: bool = boost_adding
```

```
def append(self, value):
    if self.__boost_adding is True:
        self.__fast_append(value=value)
    else:
        self.__slow_append(value=value)

def pop(self):
    if self.__boost_adding is not True:
        return self.__fast_pop()
    else:
        return self.__slow_pop()

def __fast_append(self, value):
    if self.max_size <= self.current_size:
        raise ValueError(" No memory left")
    self.current_size += 1
    self.main_data.enqueue(value=value)

def __slow_append(self, value):
    if self.max_size <= self.current_size:
        raise ValueError(" No memory left")

    self.current_size += 1
    self.temp_data.enqueue(value=value)
    while not self.main_data.empty():
        self.temp_data.enqueue(value=self.main_data.dequeue())

    temp = self.main_data
    self.main_data = self.temp_data
    self.temp_data = temp

def __fast_pop(self):
    if self.main_data.empty() is True:
        raise IndexError(" No data is remained")

    self.current_size -= 1
    return self.main_data.dequeue()

def __slow_pop(self):
    if self.main_data.empty() is True:
        raise IndexError(" No data is remained")

    while self.main_data.size() != 1:
        self.temp_data.enqueue(value=self.main_data.dequeue())
```

```
        value = self.main_data.dequeue()
        self.current_size -= 1

        temp = self.main_data
        self.main_data = self.temp_data
        self.temp_data = temp

        return value

class QueueByStack:
    def __init__(self, max_size: int, boost_adding: bool = True):
        if not isinstance(max_size, int):
            raise ValueError("The size should be an integer")
        if max_size <= 0:
            raise ValueError("The size should be a positive")
        # You can use normal Python List instead but due to course requirement

        self.main_data: Stack = Stack(max_size=max_size)
        self.temp_data: Stack = Stack(max_size=max_size)

        self.max_size: int = max_size
        self.current_size: int = 0
        self.__boost_adding: bool = boost_adding

    def enqueue(self, value):
        if self.__boost_adding is True:
            self.__fast_enqueue(value=value)
        else:
            self.__slow_enqueue(value=value)

    def dequeue(self):
        if self.__boost_adding is not True:
            return self.__fast_dequeue()
        else:
            return self.__slow_dequeue()

    def __fast_enqueue(self, value):
        if self.max_size <= self.current_size:
            raise ValueError(" No memory left")
        self.current_size += 1
        self.main_data.append(value=value)

    def __slow_enqueue(self, value):
```

```
        if self.max_size <= self.current_size:
            raise ValueError(" No memory left")

        self.current_size += 1

        while not self.main_data.empty():
            self.temp_data.append(value=self.main_data.pop())
        self.main_data.append(value)
        while not self.temp_data.empty():
            self.main_data.append(value=self.temp_data.pop())

    def __fast_dequeue(self):
        if self.main_data.empty() is True:
            raise IndexError(" No data is remained")

        self.current_size -= 1
        return self.main_data.pop()

    def __slow_dequeue(self):
        if self.main_data.empty() is True and self.temp_data.empty() is True:
            raise IndexError(" No data is remained")

        elif self.temp_data.empty() is True and self.main_data.empty() is False:
            while self.main_data.empty() is False:
                self.temp_data.append(self.main_data.pop())
            return self.temp_data.pop()
        else:
            return self.temp_data.pop()

class CircularQueue:
    def __init__(self, max_size: int):
        if not isinstance(max_size, int):
            raise ValueError("The size should be an integer")
        if max_size <= 0:
            raise ValueError("The size should be a positive")
        self.max_size = max_size

        # initializing queue with none
        self.queue = [None for _ in range(max_size)]
        self.front = self.rear = -1

    def enqueue(self, data):
        if (self.rear + 1) % self.max_size == self.front:
            raise MemoryError("No memory left")
```

```
        if self.front == -1:
            self.front = 0
            self.rear = 0
        else:
            self.rear = (self.rear + 1) % self.max_size
        self.queue[self.rear] = data

    def dequeue(self):
        if self.front == -1:
            raise MemoryError("No memory left")

        temp = self.queue[self.front]
        if self.front == self.rear:
            self.front = -1
            self.rear = -1
        else:
            self.front = (self.front + 1) % self.max_size
        return temp

    def display(self):
        if self.front == -1:
            print("Queue is Empty")

        elif self.rear >= self.front:
            print("Elements in the circular queue are:", end=" ")
            for i in range(self.front, self.rear + 1):
                print(self.queue[i], end=" ")
            print()

        else:
            print("Elements in Circular Queue are:", end=" ")
            for i in range(self.front, self.max_size):
                print(self.queue[i], end=" ")
            for i in range(0, self.rear + 1):
                print(self.queue[i], end=" ")
            print()

        if (self.rear + 1) % self.max_size == self.front:
            print("Queue is Full")
```

➔ From here, please do by yourself, implement Stack by Queue or Queue by Stack by Test Request

- Recursion

```
def fibonacci(n):
```

```
    if n in (0, 1):
        return n
    else:
        return fibonacci(n=n-1) + fibonacci(n=n-2)



def fibonacci_iteration(n):
    if n in (0, 1):
        return n
    else:
        a, b = 0, 1
        for i in range(0, n):
            a, b = a + b, a
        return a

def tower_hanoi(number_of_disk: int, source: Tuple[List, str], intermediate: Tuple[List, str], target: Tuple[List, str]):
    if number_of_disk > 0:
        # Move from source to intermediate
        tower_hanoi(number_of_disk=number_of_disk - 1, source=source, intermediate=target, target=intermediate)
        if source[0] is not None:
            data = source[0].pop()
            print("MOVING {} at {} to {}".format(data, source[1], target[1]))
            target[0].append(data)
        # Move from intermediate to target
        tower_hanoi(number_of_disk=number_of_disk - 1, source=intermediate, intermediate=source, target=target)
        print(source[0], intermediate[0], target[0])

class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def printInorder(root):
    if root:
        printInorder(root.left)
        print(root.val),
        printInorder(root.right)

def printPostorder(root):
    if root:
        printPostorder(root.left)
        printPostorder(root.right)
        print(root.val),
```

```python
def printPreorder(root):
    if root:
        print(root.val),
        printPreorder(root.left)
        printPreorder(root.right)
```