DATA STRUCTURES AND ALGORITHMS.

BY GAURAV AMARNANI.

TOPIC 1: SEARCHING AND SORTING.

SORTING: Arranging data in increasing or decreasing order or some other parameter for example number of factors.

Ascending order: 1, 2, 3, 4, 5, 6,....
Descending order: 9, 8, 7, 6, 5,....

Number of Factors: 1, 2, 3, 7, 4, 9, 6,....
On the surface it looks like it isn't sorted.
But if you look at the number of factors for each of the element,
1 -> 2 -> 3 -> 7 -> 4 -> 9 -> 6. (Elements)
1 -> 1 -> 2 -> 2 -> 3 -> 3 -> 4. (Number of factors)

So it is important to clarify beforehand about the criteria of the sorting.

Sorting Techniques:

A. Bubble Sort. O(N^2)
B. Insertion Sort. O(N^2)
C. Selection Sort. O(N^2)

D. Quick Sort. O(N * log N)
E. Merge Sort. O(N * log N)

So we will look into one technique with O(N^2) and another with O(N * log N) time complexity.

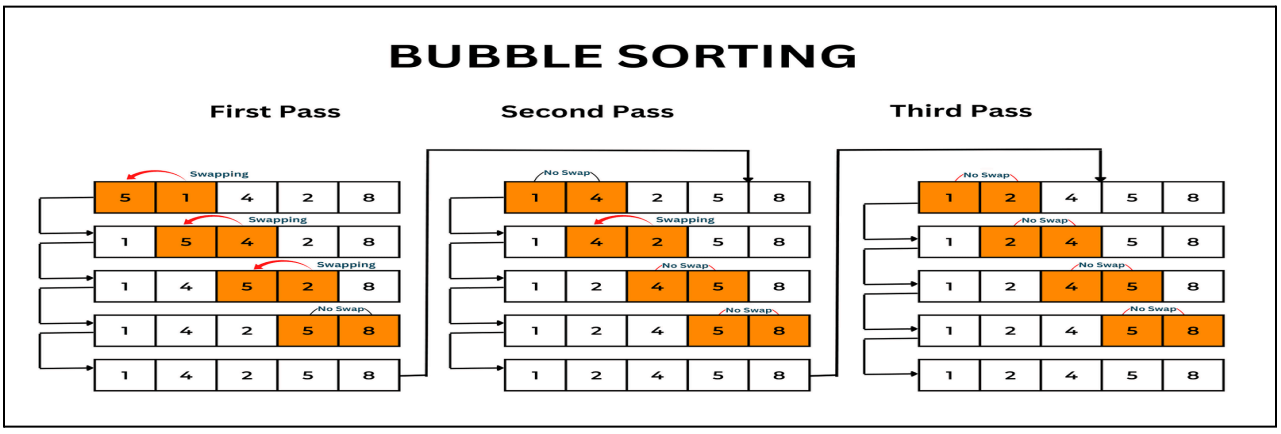
A. Bubble Sort:

In this we check if the current element is greater than the next element or not. If it is then we swap it and if it's not then we increase the pointer.
It essentially finds the last element in the array with each iteration.
So we need to have N-1 iterations as the 0th index will be sorted by the end.

Code:

```
class DSA {
      pubic int[] bubbleSort(int[] arr) {
            //Loop for N-1 iterations.
            for(int i = 0; i < arr.length - 1; i++) {
                  //Iterate through elements.
                  for(int j = 0; j < arr.length - 1 - i; j++) {
                        //Swap if the current element is bigger.
                        if(arr[j] > arr[j+1]) {
                              int temp = arr[j];
                              int arr[j] = arr[j+1];
                              int arr[j+1] = temp;
                        }
                  }
            }
            return arr;
      }
}
```

After each iteration the **(n-i-1)th element** will be in its correct place:



BY GAURAV AMARNANI.

Now the issue with O(N Log N) Time Complexity algorithm is that Merge Sort works better with huge datasets as it is able to give a Time Complexity of O(N Log N) even in the worst case scenario whereas the Quick Sort is only able to perform at O(N^2) Time Complexity in worst case.

So we have different scenarios where we have to use each one of them.

In interviews if the interviewer is asking the question without any Leetcode or any other platform involvement then it is safe to use Quick Sort because the dataset size is not huge and Quick Sort will perform better than Merge Sort.
But if any platform is used, then Merge Sort should be used because they pass huge datasets to check the performance of our algorithm against it.

So let's start with Merge Sort first:

B. Merge Sort:

Process of Merge Sorting: It is Divide and Conquer Recursive calling.

Let's assume that the array has 9 elements: [8, 7, 6, 5, 4, 3, 2, 1, 0]

Now we have to use the following Pseudo Code as driving function:

```
class DSA {
     public void mergeSort(int[] A, int lB, int uB) {
          if(lB < uB) {
               int mid = (lB = uB)/2;
               mergeSort(A, lB, mid);
               mergeSort(A, mid+1, uB);
               merge(A, lB, mid, uB);
          }
     }
}
```

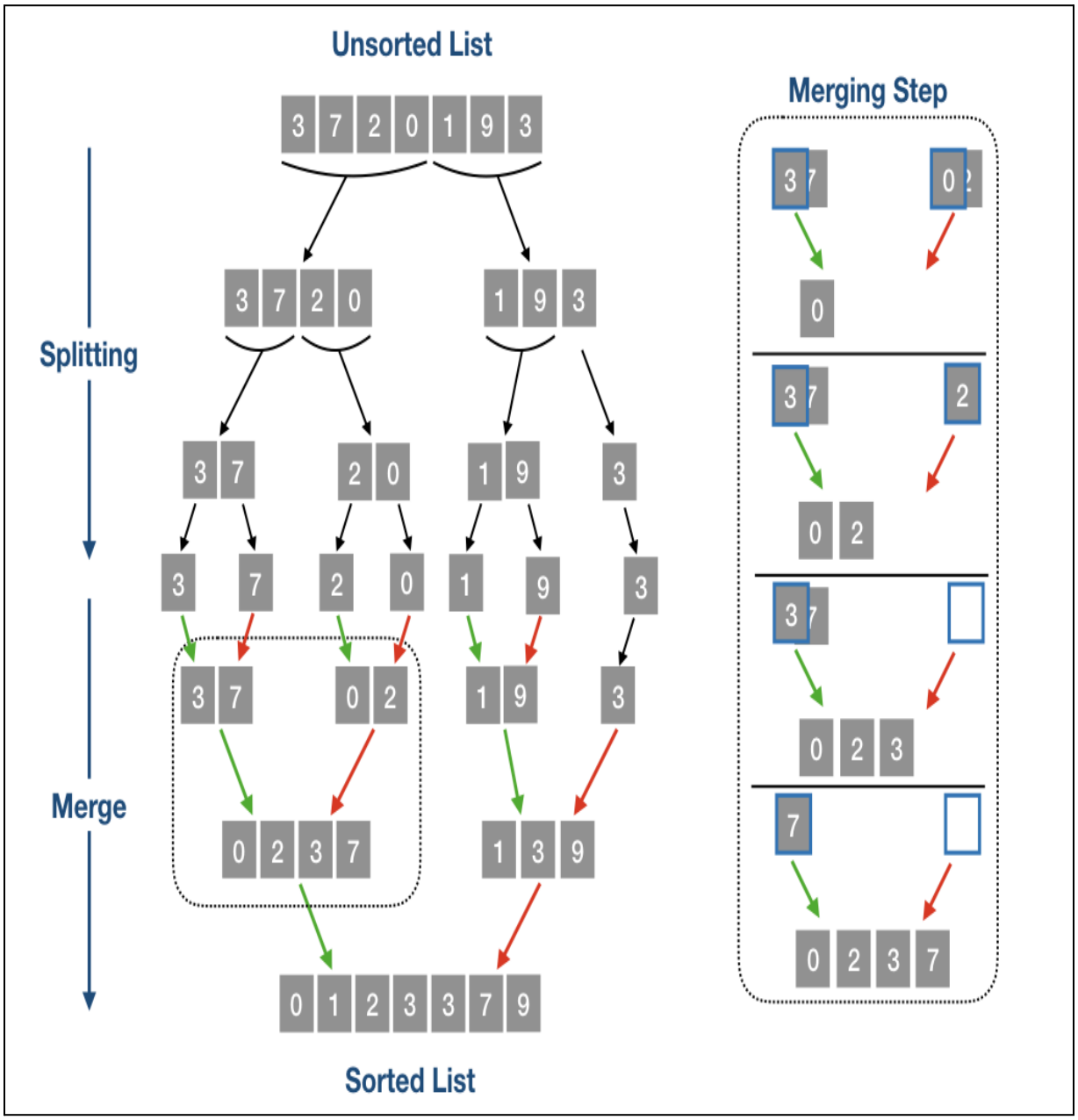
Through this code the following will the be function calls:

ms(0, 8)	Values Passed: [8, 7, 6, 5, 4, 3, 2, 1, 0].
ms(0, 4)	Values Passed: [8, 7, 6, 5, 4].
ms(0, 2)	Values Passed: [8, 7, 6].
ms(0, 1)	Values Passed: [8, 7].
ms(0, 0)	Values Passed: [8].
ms(1, 1)	Values Passed: [7].
merge(0, 0, 1)	From [8, 7] to [7, 8].
ms(2, 2)	Values Passed: [6].
merge(0, 1, 2)	From [7, 8, 6] to [6, 7, 8].
ms(3, 4)	Values Passed: [5, 4].
ms(3, 3)	Values Passed: [5].
ms(4, 4)	Values Passed: [4].
merge(3, 3, 4)	From [5, 4] to [4, 5].
merge(0, 2, 4)	From [6, 7, 8, 4, 5] to [4, 5, 6, 7, 8].
ms(5, 8)	Values Passed: [3, 2, 1, 0].
ms(5, 6)	Values Passed: [3, 2].
ms(5, 5)	Values Passed: [3].
ms(6, 6)	Values Passed: [2].
merge(5, 5, 6)	From [3, 2] to [2, 3].
ms(7, 8)	Values Passed: [1, 0].
ms(7, 7)	Values Passed: [1].
ms(8, 8)	Values Passed: [0].
merge(7, 7, 8)	From [1, 0] to [0, 1].
merge(5, 6, 8)	From [2, 3, 0, 1] to [0, 1, 2, 3].
merge(0, 4, 8)	From [4, 5, 6, 7, 8, 0, 1, 2, 3] to [0, 1, 2, 3, 4, 5, 6, 7, 8].

The following is the merge code which will basically check if the incoming array's elements with the already existing array's elements one by one until one list is over.

```
class DSA {
      public void merge(int[] A, int l, int mid, int u) {
            int i = l;
            int j = mid + 1;
            int k = l;

            while(i <= mid && j <= u) {
                  if(a[i] <= a[j]) {
                        b[k] = a[i];
                        i++;
                  }
                  else {
                        b[k] = a[j];
                        j++;
                  }
                  k++;
            }

            while(i <= mid) {
                  b[k] = a[i];
                  i++;
                  k++;
            }

            while(j <= u) {
                  b[k] = a[j];
                  j++;
                  k++;
            }

            for(k = l; k < u; k++) {
                  a[k] = b[k];
            }
      }
}
```

The following is the visual representation of working of Merge Sort:



C. Quick Sort:

This approach also follows a divide and conquer approach.
It is different from Merge Sort as it uses two pointers to put the pivot element at its correct location during each iteration.

It starts by selecting a pivot element (generally we use the first element or the lower bound as the pivot element), after picking the pivot we start by comparing it with the start and end pointers inside the array.

The conditions are that the start pointer's element should be smaller than the pivot and the end pointer's element should be greater than the pivot.

We keep iterating until that's not the case for both the conditions and after that we swap them.

This all happens until start < end. Once they overlap each other the pivot element is swapped with the end pointer's element and now that is the final location of the pivot element inside that array.

It now repeats the process from start to mid and mid +1 to end.
Following is the driver code for this:

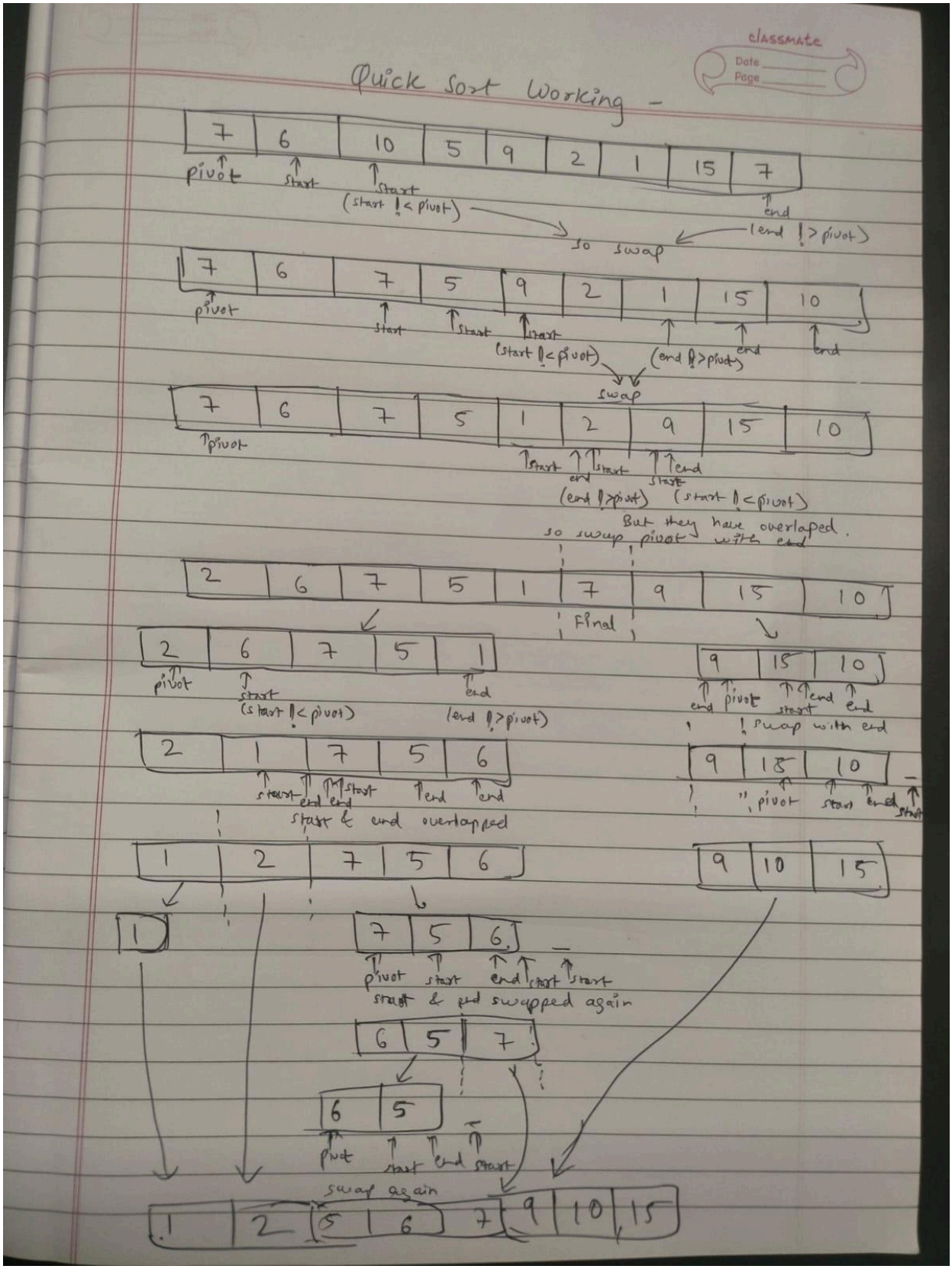
```
class DSA {
      public void sort(int[] A, int l, int u){
            if(l < u) {
                  int location = partition(A, l, u);
                  partition(A, l, location - 1);
                  partition(A, location + 1, u);
            }
      }
}
```

This code calls the partition method recursively until all the elements are in their final positions.

The main code for Quick Sort:

```
class DSA {
	public void partition(int[] A, int l, int u) {
		int pivot = A[l];
		int start = l;
		int end = u;

		while(start < end) {
			while(A[start] < pivot) {
				start++;
			}
			while(A[end] > pivot) {
				end--;
			}
			if(start < end) {
				int temp = A[start];
				A[start] = A[end];
				A[end] = temp;
			}
		}
		int temp = A[l];
		A[l] = A[end];
		A[end] = temp;
	}
}
```

The following is the visual representation to explain the internal working of Quick Sort Algorithm:


Quick Sort Working -
pivot
start
(start !< pivot)
(end !> pivot)
so swap
But they have overlaped.
so swap pivot with end
Final
swap with end
start & end overlapped
start & end swapped again
swap again

SEARCHING: In Searching we have two methods, one is linear search which is basically iterating through all the elements of the array and finding the one we are looking for, it gives us O(N) Time Complexity.

The alternative to this is the Binary Search Algorithm, which uses Divide and Conquer approach. It only works on sorted arrays, it uses the midpoint of the array as the starting point and goes from there.

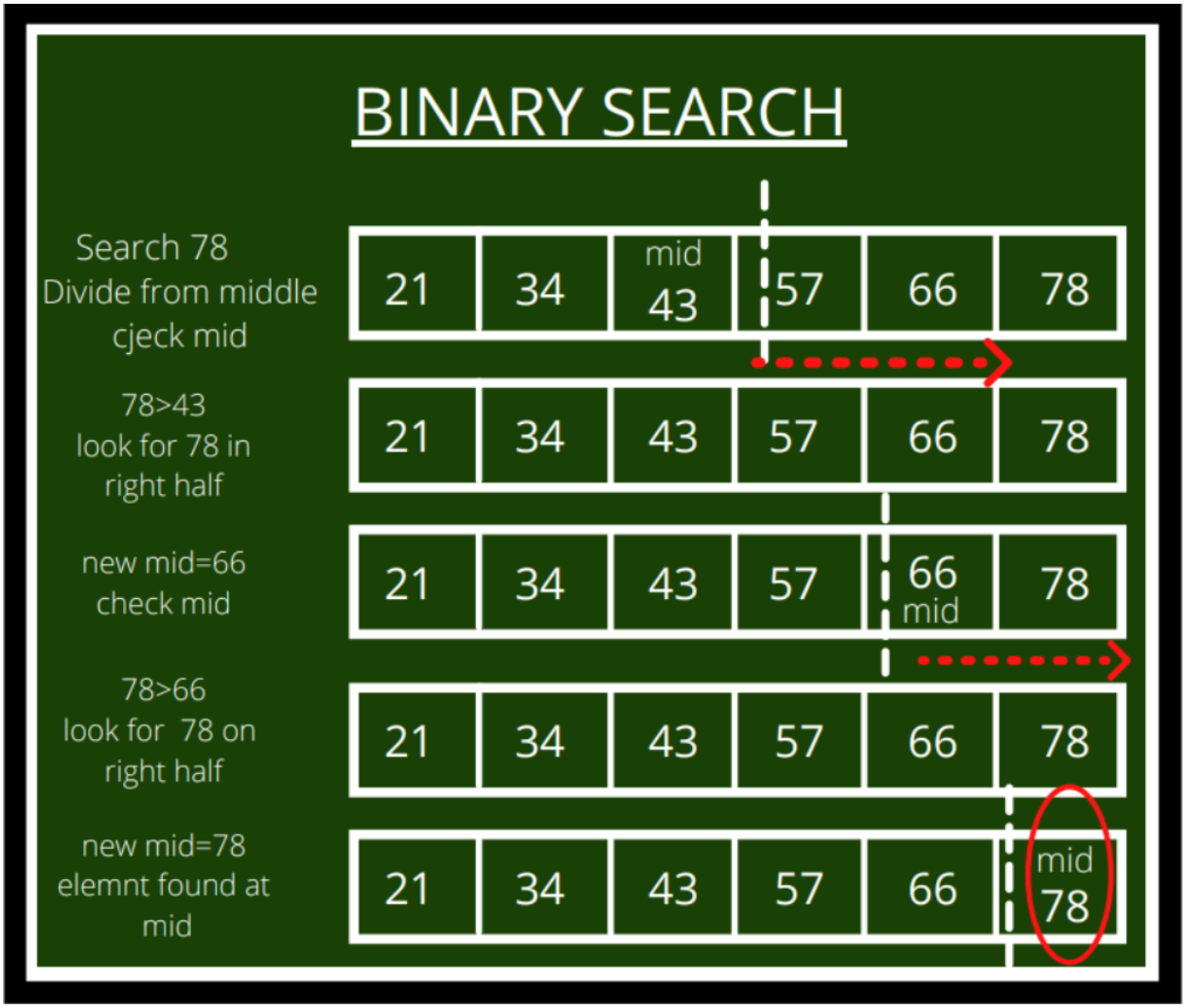
It first checks if the element on the midpoint is the element we are looking for or not. If it is not then it checks whether it is smaller or bigger.

If it is bigger then the right side of the midpoint is discarded as the element we are looking for cannot exist on the right side as it is sorted. Same for left, if the element is smaller then the left side is discarded.

Here is the Pseudo Code for the above explanation:

```
class DSA {
	//Array has to be sorted for Binary Search to work.
	public void binarySearch(int[] A, int element) {
		int left = 0;
		int right = A.length - 1;
		while(left <= right) {
			int mid = (left + right)/2;
			if(A[mid] == element) {
				return mid; //Found at mid.
			}
			else if(A[mid] > element) {
				right = mid - 1;
			}
			else {
				left = mid + 1;
			}
		}
		return -1; //Not Found.
	}
}
```

Now for the visual representation of Binary Search Algorithm:



Binary Search has a Time Complexity of O(Log N) and it does not work for unsorted arrays.

XXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXX

TOPIC 2: ARRAYS.

1D Arrays are pretty simple so will just solve some common problems.

The following is the list of questions we are gonna solve:

1. Find Sum.
2. Find Max.
3. Change element through function.
4. Swap array indexes.
5. Reverse an Array.
6. Reverse partial Array.
7. Rotate partial Array.
8. Greater Element Problem.
9. Two Sum Problem I.
10. Two Sum Problem II.

The following is the driver code:

```
public static void main(String...args) {

        ArrayQuestion arrayQuestion = new ArrayQuestion();

        int[] arr = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, 16, 17, 18, 19, 20};

        arrayQuestion.printArray(arr, "The array:");

        int sum = arrayQuestion.sum(arr);
        out.println("\nSum of the array is: " + sum);

        int max = arrayQuestion.max(arr);
        out.println("\nMax of the array is: " + max);

        arrayQuestion.changesWithFunction(arr, 0, 10);
        arrayQuestion.printArray(arr, "\n\nArray after changing 1st
element to 10:");
```

```
        arrayQuestion.reverseArray(arr);
        arrayQuestion.printArray(arr, "Array after reversing:");

        arrayQuestion.reversePartOfArray(arr, 9);
        arrayQuestion.printArray(arr, "\n\nArray after reversing only
the first 10 elements:");

        arrayQuestion.rotateArray(arr, 10);
        arrayQuestion.printArray(arr, "\n\nArray after rotating 10
elements:");

        arrayQuestion.reversePartOfArray(arr, 9);
        arrayQuestion.printArray(arr, "\n\nReversing part of array
again to get back to original.");

        arrayQuestion.changesWithFunction(arr, 0, 1);
        arrayQuestion.printArray(arr, "\n\nArray after changing 1st
element back to 1:");

        int countOfGreaterElement =
arrayQuestion.greaterThanItself(arr);
        out.println("\n\nCount of Greater Elements is: " +
countOfGreaterElement);

        arrayQuestion.twoSum(arr, 39);
        arrayQuestion.twoSum2(arr, 39);
}
```

The following are function code snippets for each question:

Sum:

```
    public int sum(int[] arr) {
        int sum = 0;
        for(int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
        return sum;
    }
```

Max:

```
    public int max(int[] arr) {
        int max = Integer.MIN_VALUE;
        for(int i = 0; i < arr.length; i++) {
            if(max < arr[i])
                max = arr[i];
        }
        return max;
    }
```

Change value with function:

```
    public void changesWithFunction(int[] arr, int position, int
newValue) {
        arr[position] = newValue;
    }
```

Swap indexes:

```
    public void swapTwoIndexes(int[] arr, int pos1, int pos2) {
        int temp = arr[pos1];
        arr[pos1] = arr[pos2];
        arr[pos2] = temp;
    }
```

Reverse Array:

```
    public void reverseArray(int[] arr) {
        int left = 0;
        int right = arr.length - 1;
        while(left < right) {
            swapTwoIndexes(arr, left, right);
            left++;
            right--;
        }
    }
```

Reverse Partial Array:

```
    public void reversePartOfArray(int[] arr, int k) {
        int left = 0;
        int right = k;
        while (left < right) {
            swapTwoIndexes(arr, left, right);
            left++;
            right--;
        }
    }
```

Two Sum I:

```
    public void twoSum(int[] arr, int target) {
        Map<Integer, Integer> numMap = new HashMap<>();
        for(int i = 0; i < arr.length; i++) {
            int complement = target - arr[i];
            if(numMap.containsKey(complement)) {
                int[] temp = new int[] {arr[numMap.get(complement)],
arr[i] };
                printArray(temp, "\nTwo Sum for " + target + ": ");
            } else {
                numMap.put(arr[i], i);
            }
        }
    }
```

Rotate Partial Array:

```
    public void rotateArray(int[] arr, int k) {
        k %= arr.length;
        int[] arr2 = new int[k];
        int count = 0;
        //Copy the elements to be rotated into another array.
        for(int i = 0; i < k; i++) {
            arr2[i] = arr[i];
        }

        //Put the left elements into the front.
        for(int i = k; i < arr.length; i++) {
            arr[count] = arr[i];
            count++;
        }
        count = 0;
        //Insert the rotated elements into the original array.
        for(int i = k; i < arr.length; i++) {
            arr[i] = arr2[count];
            count++;
        }
    }
```

Greater Than Max Element:

```
    public int greaterThanItself(int[] arr) {
        int max = Integer.MIN_VALUE;
        int count = 0;
        for(int i = 0; i < arr.length; i++) {
            if(max < arr[i]) {
                max = arr[i];
                count = 1;
            }
            else if(max == arr[i])
                count++;
        }
        return arr.length - count;
    }
```

Results for reference:

```
The array:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20]

Sum of the array is: 210

Max of the array is: 20

Array after changing 1st element to 10:
[10, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20]

Array after reversing:
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
10]

Array after reversing only the first 10 elements:
[11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 10, 9, 8, 7, 6, 5, 4, 3, 2,
10]

Array after rotating 10 elements:
[10, 9, 8, 7, 6, 5, 4, 3, 2, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20]

Reversing part of array again to get back to original.
[10, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20]

Array after changing 1st element back to 1:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20]

Count of Greater Elements is: 19

Two Sum for 39:
[19, 20]
```

2D Array Questions:

1. Print Matrix row wise.
2. Print Matrix column wise.
3. Print Matrix in wave form.

The following is the driver code:

```
class DSA {
	public static void main(String...args) {
		int[][] arr = new int[4][4];
		for(int i = 0; i < arr.length; i++) {
			for(int j = 0; j < arr[i].length; j++) {
				arr[i][j] = j;
			}
		}
		DSA dSA = new DSA();
		dSA.columnWise(arr);
		dSA.rowWise(arr);
		dSA.waveform(arr);
	}
}
```

Column Wise:

```
class DSA {
	public void columnWise(int[] arr) {
		out.println("Column Wise Array: ");
		for(int i = 0; i < arr.length; i++) {
			for(int j = 0; j < arr[i].length; j++)
				out.print(arr[i][j] + " ");
			out.println();
		}
	}
}
```

Row Wise:

```
class DSA {
	public void rowWise(int[] arr) {
		out.println("Row Wise Array: ");
		for(int i = 0; i < arr.length; i++) {
			for(int j = 0; j < arr[i].length; j++)
				out.print(arr[j][i] + " ");
			out.println();
		}
	}
}
```

Wave Form:

```
class DSA {
	public void waveform(int[] arr) {
		out.println("Wave Form Array: ");
		for(int i = 0; i < arr.length; i++) {
			if(i % 2 == 0)
				for(int j = 0; j < arr[i].length; j++)
					out.print(arr[i][j] + " ");
			else
				for(int j = arr[i].length - 1; j >= 0; j--)
					out.print(arr[i][j] + " ");
			out.println();
		}
	}
}
```

Results for reference:

```
Column Wise Array:
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Row Wise Array:
0 0 0 0
1 1 1 1
2 2 2 2
3 3 3 3

Wave Form Array:
0 1 2 3
3 2 1 0
0 1 2 3
3 2 1 0
```

XXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXX

TOPIC 3: LINKED LIST.

So the LinkedList Data Structure basically maintains a head (or tail) pointer to the data locations which contains the value. So now the next question is how it might be more useful than the array?
It has two specific scenarios for which it is very useful.
These scenarios are 1. While inserting at the start or end 2. While deleting at the start or the end.

The insertion in arrays is also usually O(1) but if there is a need to resize the array then the time complexity becomes O(N) as we need to copy all the elements from one location to another.

LinkedList doesn't need to deal with shifting all the elements in case of deletion or insertion like Array.

Apart from these it has some limitations as well, like the searching algorithm Binary Search cannot be applied for it efficiently.

LinkedList Time Complexities:

1. Insertion at start/end - O(1).
2. Deletion at start/end - O(1).
3. Insertion at specific location - O(N).
4. Deletion at specific location - O(N).
5. Updation - O(N).
6. Traversal - O(N).
7. Searching - O(N).

The following is the LinkedList class definition:

```
class Node {

	int value;
	Node next;

	Node() {}

	Node(int value) {
		this.value = value;
	}

	Node(int value, Node next) {
		this.value = value;
		this.next = next;
	}
}
```

When we want to insert in a LinkedList we have to keep the following things in mind:
If the list is empty then we have to insert at head.
If the list is not empty we have to insert until we don't find the end of the list (by looking for temp.next == null).
If we have to insert at a specific location then just iterate until we reach a node before that location and insert it after that node.

The following are the implementation of the insert codes:

Insert at Start:

```
public static void insertAtStart(int value) {
	Node newNode = new Node(value);
	newNode.next = head;
	head = newNode;
}
```

At Specific location:

```
public static void insertAt(int location, int value) {
      if(isEmpty()) {
            out.println("\nLinkedList is null. Cannot insert.");
      }
      else {
            if(location <= 0) {
                  out.println("\nLocation cannot be negative or 0
bruh. It starts from 1 in LinkedList.");
            }
            else if(location == 1) {
                  insertAtStart(value);
                  out.println("\n" + value + " inserted at the
start.");
            }
            else {
                  Node temp = head;
                  for(int i = 1; i < location - 1; i++) {
                        if(temp.next != null) {
                              temp = temp.next;
                        }
                  }
                  if(temp == null || temp.next == null) {
                        out.println("\nLinkedList does not have " +
location);
                  }
                  else {
                        Node tempNext = temp.next;
                        Node newNode = new Node(value);
                        temp.next = newNode;
                        newNode.next = tempNext;
                        out.println("\nInserted at location " +
location + ".");
                  }
            }
      }
}
```

Insert at End:

```
public static void insertAtLast(int value) {
      if(isEmpty()) {
            insertAtStart(value);
            out.println("\n" + value + " inserted at the start.");
      }
      else {
            Node temp = head;
            while(temp.next != null)
                  temp = temp.next;
            Node newNode = new Node(value);
            temp.next = newNode;
            newNode.next = null;
            out.println("\n" + value + " inserted at the last.");
      }
}
```

The following is the code for updating a LinkedList:

Update the Value:

```
public static void updateValue(int oldValue, int newValue) {
      if(isEmpty()) {
            out.println("\nLinkedList is very empty sir.");
      }
      else {
            Node temp = head;
            while(temp != null) {
                  if(temp.value == oldValue) {
                        temp.value = newValue;
                        out.println("\n" + newValue + " inserted.");
                        return;
                  }
                  temp = temp.next;
            }
            out.println("\nCould not find " + oldValue);
      }
}
```

Update the Locations value:

```
public static void updateNode(int location, int newValue) {
	if(isEmpty()) {
		out.println("\nLinkedList is very empty sir. So cannot
update it.");
	}
	else {
		Node temp = head;
		for(int i = 1; i < location; i++) {
			if(temp.next != null) {
				temp = temp.next;
			}
		}
		if(temp != null) {
			int deletedValue = temp.value;
			temp.value = newValue;
			out.println("\n" + newValue + " inside the Node with
value " + deletedValue + ".");
		}
		else
			out.println("\n" + location + " does not exist
inside the LinkedList.");
	}
}
```

To check if the LinkedList is empty or not:

```
public static boolean isEmpty() {
	return head == null;
}
```

To check if the LinkedList is full or not:

```
public static void insertAtStart(int value) {
	Node newNode = new Node(value);
	newNode.next = head;
	head = newNode;
}
```

Now the following for the deletion of the LinkedList:

Delete First:

```
public static void deleteFirst() {
	if(isEmpty()) {
		out.println("\nLinkedList is very empty sir. Cannot delete
from it.");
	}
	else {
		int deleted = head.value;
		head = head.next;
		out.println("\nDeleted " + deleted + " from the start.");
	}
}
```

Delete Last:

```
public static void deleteLast() {
	if(isEmpty()) {
		out.println("\nLinkedList is very empty sir. Cannot delete
from it.");
	}
	else {
		Node temp = head;
		while(temp.next != null && temp.next.next != null) {
			temp = temp.next;
		}
		int deleted = temp.next.value;
		temp.next = null;
		out.println("\nDeleted " + deleted + " from the end.");
	}
}
```

Delete At Index Number:

```
public static void deleteAt(int location) {
	if(isEmpty()) {
		out.println("\nLinkedList is very empty sir. Cannot delete
from it.");
	}
	else {
		Node temp = head;
		for(int i = 1; i < location - 1; i++) {
			if(temp != null && temp.next != null) {
				temp = temp.next;
			}
		}
		if(temp != null && temp.next != null) {
			int deleted = temp.next.value;
			temp.next = temp.next.next;
			out.println("\nDeleted " + deleted + " from " +
location);
		}
		else
			out.println("\n" + location + " not reachable.");
	}
}
```

LinkedList Traversal:

```
public static void printLinkedList() {
	if(isEmpty())
		out.println("LinkedList is empty sir.");
	else {
		Node temp = head;
		out.print("\nLINKEDLIST:\nHEAD->");
		while(temp != null) {
			out.print(temp.value + "->");
			temp = temp.next;
		}
		out.print("END.\n");
	}
}
```

Reverse a LinkedList:

Initialize three pointers prev as NULL, curr as head, and next as NULL. Iterate through the linked list. In a loop, do the following:

Before changing the next of curr, store the next node

```
next = curr -> next
```

Now update the next pointer of curr to the prev

```
curr -> next = pr
```

Update prev as curr and curr as next

```
prev = curr
curr = next
```

Implementation:

```
Node reverse(Node node) {
      Node prev = null;
      Node current = node;
      Node next = null;
      while (current != null) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
      }
      node = prev;
      return node;
}
```

Finding the Mid of the LinkedList:

Finding the mid is not that difficult if we don't have a restriction on iteration of the linkedlist, we can easily iterate through it once and then find out the total length and then use that to find the mid.

But if we are restrained with the number of iterations, for examples if we can only iterate through the linkedlist only and only once then we have to follow the following solution:

Initialize a slow pointer `slow_ptr = head` and a fast pointer `fast_ptr = head`.
Iterate till fast_ptr reaches the last node(`fast_ptr->next != null`) or becomes NULL(`fast_ptr != null`).
Move fast_ptr by two nodes, `fast_ptr = fast_ptr->next->next`.
Move slow_ptr by one node, `slow_ptr = slow_ptr->next`.
As soon as the fast_ptr reaches the last node or becomes NULL, return the value at slow_ptr.

Implementation:

```
public static int getMiddle(Node head) {
      Node slow_ptr = head;
      Node fast_ptr = head;

      while (fast_ptr != null && fast_ptr.next != null) {
            fast_ptr = fast_ptr.next.next;
            slow_ptr = slow_ptr.next;
      }
      return slow_ptr.data;
}
```

This approach is called the Floyd's Cycle Finding Algorithm (To use the fast and slow pointer basically).

Finding cycles in the LinkedList:

We will again use the Floyd's Cycle Finding Algorithm for this:

```
public static void detectLoop() {
      Node slow_p = head, fast_p = head;
      int flag = 0;
      while (slow_p != null && fast_p != null && fast_p.next != null)
{
            slow_p = slow_p.next;
            fast_p = fast_p.next.next;
            if (slow_p == fast_p) {
                  flag = 1;
                  break;
            }
      }
      if (flag == 1)
            System.out.println("Loop Found");
      else
            System.out.println("No Loop");
}
```

The driver code:

```
class MyLinkedList {

	static Node head = null;

	public static void main(String...args) {
		printLinkedList();
		insertAtStart(40);
		printLinkedList();
		insertAtStart(30);
		printLinkedList();
		insertAtStart(20);
		printLinkedList();
		insertAtStart(10);
		printLinkedList();
		insertAt(4, 50);
		printLinkedList();
		insertAtLast(60);
		printLinkedList();
		insertAtLast(70);
		printLinkedList();
		updateNode(7, 65);
		printLinkedList();
		updateValue(65, 70);
		printLinkedList();
		deleteFirst();
		printLinkedList();
		deleteAt(3);
		printLinkedList();
		deleteLast();
		printLinkedList();
	}
}
```

The output of the above code for reference:

```
LinkedList is empty sir.

LINKEDLIST:
HEAD->40->END.

LINKEDLIST:
HEAD->30->40->END.

LINKEDLIST:
HEAD->20->30->40->END.

LINKEDLIST:
HEAD->10->20->30->40->END.

50 inserted at location 4.

LINKEDLIST:
HEAD->10->20->30->50->40->END.

60 inserted at the last location.

LINKEDLIST:
HEAD->10->20->30->50->40->60->END.

70 inserted at the last location.

LINKEDLIST:
HEAD->10->20->30->50->40->60->70->END.

65 inserted at Location 7 inside the Node with value 70.

LINKEDLIST:
HEAD->10->20->30->50->40->60->65->END.

70 inserted instead of 65

LINKEDLIST:
```

```
HEAD->10->20->30->50->40->60->70->END.

Deleted 10 from the start.

LINKEDLIST:
HEAD->20->30->50->40->60->70->END.

Deleted 50 from location 3

LINKEDLIST:
HEAD->20->30->40->60->70->END.

Deleted 70 from the end.

LINKEDLIST:
HEAD->20->30->40->60->END.
```

XXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXX

TOPIC 4: STACKS.

The following are the Stack operations along with their time complexities:

Push: Add an element to the top of a stack. [O(1)]

Pop: Remove an element from the top of a stack. [O(1)]

IsEmpty: Check if the stack is empty. [O(1)]

IsFull: Check if the stack is full. [O(1)]

Peek: Get the value of the top element without removing it. [O(1)]

Print: Print the content of the stack. [O(1)]

Algorithm for Push Operation:

Before pushing the element to the stack, we check if the stack is full .
If the stack is full (top == capacity-1) , then Stack Overflows and we cannot insert the element to the stack.
Otherwise, we increment the value of top by 1 (top = top + 1) and the new value is inserted at top position .
The elements can be pushed into the stack till we reach the capacity of the stack.

Implementation:

```
public void push(String element) {
      if(isFull()) {
            System.out.println("Stack is full.");
            return;
      }
      arr[++top] = element;
      System.out.println("Inserted Element: " + element);
}
```

Algorithm for Pop Operation:

Before popping the element from the stack, we check if the stack is empty . If the stack is empty (top == -1), then Stack Underflows and we cannot remove any element from the stack.
Otherwise, we store the value at top, decrement the value of top by 1 (top = top – 1) and return the stored top value.

Implementation:

```
public String pop() {
      if(isEmpty()) {
            System.out.println("Stack is empty so cannot pop.");
            return "-1";
      }
      return arr[top--];
}
```

Algorithm for Top/Peek Operation:

Before returning the top element from the stack, we check if the stack is empty.
If the stack is empty (top == -1), we simply print “Stack is empty”.
Otherwise, we return the element stored at index = top .

Implementation:

```
public String peek() {
      if(isEmpty()) {
            System.out.println("Stack is empty so cannot peek.");
            return "-1";
      }
      return arr[top];
}
```

Algorithm for isEmpty Operation :

Check for the value of top in stack.
If (top == -1) , then the stack is empty so return true .
Otherwise, the stack is not empty so return false .

Implementation:

```
public boolean isFull() {
      return (top + 1) == SIZE;
}
```

Algorithm for isFull Operation:

Check for the value of top in stack.
If (top == capacity-1), then the stack is full so return true .
Otherwise, the stack is not full so return false.

Implementation:

```
public boolean isEmpty() {
      return (top == -1);
}
```

Print Stack Implementation:

```
public void printStack() {
      System.out.print("\nTOP -> ");
      for(int i = top; i>= 0; i--)
            System.out.print(arr[i] + "");
}
```

The driver code:

```
class MyStack {

	private int SIZE = 0;
	private int top = -1;
	private String[] arr;

	MyStack() {
		SIZE = 10; //DEFAULT SIZE SET TO 10.
		arr = new String[SIZE];
	}

	MyStack(int size) {
		SIZE = size;
		arr = new String[SIZE];
	}

	MyStack(String data) {
		SIZE = data.length();
		arr = new String[SIZE];
		if(data.length() != 0)
			for(int i = 0; i < data.length(); i++)
				push(data.charAt(i) + "");
	}

	public int getSize() {
		return SIZE;
	}
}
```

Output for reference:

```
Trying to pop an empty stack:
Stack is empty so cannot pop.
Popped element: -1
Inserted Element: 10
Inserted Element: 9
Inserted Element: 8
Inserted Element: 7
Inserted Element: 6
Inserted Element: 5
Inserted Element: 4
Inserted Element: 3
Inserted Element: 2
Inserted Element: 1
Trying to push an element beyond the size of the stack:
Stack is full.
Stack after all the elements got pushed:
TOP
1
2
3
4
5
6
7
8
9
10
Popping 3 elements from the stack:
Stack after all the elements got popped:
TOP
4
5
6
7
8
9
10
```

Another famous question on Stacks is the implementation of the Undo and Redo Operations using Stack Data Structure:

All Functions:

```
public void push(String element) {
      if(isFull()) {
            return;
      }
      arr[++top] = element;
}

public String pop() {
      if(isEmpty()) {
            return "-1";
      }
      return arr[top--];
}

public boolean isFull() {
      return (top + 1) == SIZE;
}

public boolean isEmpty() {
      return (top == -1);
}

public StringBuffer getTheData() {
      StringBuffer data = new StringBuffer("");
      for(int i = 0; i < top + 1; i++)
            data.append(arr[i]);
      return data;
}
```

The Main Class Code:

```
class MyStackClass {

	private int SIZE = 0;
	private int top = -1;
	private String[] arr;

	MyStackClass() {
		SIZE = 10; //DEFAULT SIZE SET TO 10.
		arr = new String[SIZE];
	}

	MyStackClass(int size) {
		SIZE = size;
		arr = new String[SIZE];
	}

	MyStackClass(String data) {
		SIZE = data.length();
		arr = new String[SIZE];
		if(data.length() != 0)
			for(int i = 0; i < data.length(); i++)
				push(data.charAt(i) + "");
	}
}
```

This MyStackClass is being used as a Data Structure and following is the driver UndoRedoClass code:

```
class UndoRedoStack {
	public static void main(String...args) {
		Scanner scanner = new Scanner(System.in);
		System.out.println("Enter text: ");
		String data = scanner.nextLine();
		System.out.println("Data entered is: " + data);
		MyStackClass undo = new MyStackClass(data);
		MyStackClass redo = new MyStackClass(data.length());
		performUndoRedoActions(undo, redo);
	}
}
```

The following is the performUndoRedoActions() method:

```
public static void performUndoRedoActions(MyStackClass undo, MyStackClass
redo) {
      while(true) {
            System.out.println("Please Enter U for Undo Operation and R for
Redo Operation and E for Exit: ");
            String input = scanner.nextLine();

            if(input != null) {
                  if(input.equals("U")) {
                        String element = undo.pop();
                        if(element != "-1") redo.push(element);
                        else System.out.println("There is nothing to
Undo.");
                        System.out.println("Data after the operation: " +
undo.getTheData());
                  }
                  else if(input.equals("R")) {
                        String element = redo.pop();
                        if(element != "-1") undo.push(element);
                        else System.out.println("There is nothing to
Redo.");
                        System.out.println("Data after the operation: " +
undo.getTheData());
                  }
                  else if(input.equals("E")) {
                        System.out.println("Data before exiting: " +
undo.getTheData());
                        break;
                  }
                  else {
                        System.out.println("Please enter a valid input from
the options.");
                        break;
                  }
            }
      }
}
```

Output for reference:

```
Enter text:
Gaurav
Data entered is: Gaurav
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
U
Data after the operation: Gaura
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
U
Data after the operation: Gaur
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
U
Data after the operation: Gau
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
U
Data after the operation: Ga
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
U
Data after the operation: G
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
U
Data after the operation:
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
U
There is nothing to Undo.
Data after the operation:
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
R
Data after the operation: G
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
R
Data after the operation: Ga
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
R
Data after the operation: Gau
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
R
Data after the operation: Gaur
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
R
Data after the operation: Gaura
```

```
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
R
Data after the operation: Gaurav
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
R
There is nothing to Redo.
Data after the operation: Gaurav
Please Enter U for Undo Operation and R for Redo Operation and E for Exit:
E
Data before exiting: Gaurav
```

XXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXX

TOPIC 5: QUEUES.

The following are the operations performed on Queue Data Structure:

Enqueue: Add an element to the end of the queue. [O(1)]

Dequeue: Remove an element from the front of the queue. [O(N)]

IsEmpty: Check if the queue is empty. [O(1)]

IsFull: Check if the queue is full. [O(1)]

Front: Get the value of the front of the queue without removing it. [O(1)]

Now, some of the implementations of queue operations are as follows:

Enqueue: Addition of an element to the queue. Adding an element will be performed after checking whether the queue is full or not. If rear == size then it is said to be an Overflow condition as the array is full.else which indicates that the array is not full then store the element at arr[rear] and increment rear by 1 .

Dequeue: Removal of an element from the queue. An element can only be deleted when there is at least an element to delete. If front == rear, then it is said to be an Underflow condition as the array is empty . else, the element at arr[front] can be deleted but all the remaining elements have to shift to the left by one position in order for the dequeue operation to delete the second element from the left on another dequeue operation.

Front: Get the front element from the queue i.e. arr[front] if the queue is not empty.

Display: Print all elements of the queue. If the queue is non-empty, traverse and print all the elements from the index front to rear.

Implementation:

Enqueue:

```
public void enqueue(int element) {
      if(isFull()) {
            System.out.println("Queue is full.");
            return;
      }
      if(front == -1) front = 0;
      arr[++rear] = element;
}
```

Dequeue:

```
public int dequeue() {
      if(isEmpty()) {
            System.out.println("Queue is empty.");
            return -1;
      }
      return arr[front++];
}
```

Print Queue:

```
public void printQueue() {
      System.out.print("\nFRONT -> ");
      for(int i = front; i < rear + 1; i++)
            System.out.print(arr[i] + " -> ");
      System.out.println("REAR.");
}
```

Get Front:

```
public int peek() {
      if(isEmpty()) {
            System.out.println("Queue is empty.");
            return -1;
      }
      return arr[rear];
}
```

Is Full:

```
public boolean isFull() {
	return rear == SIZE - 1;
}
```

Is Empty:

```
public boolean isEmpty() {
	return front == SIZE;
}
```

Main Class:

```
class MyQueue {

	private int SIZE;
	private int front = -1;
	private int rear = -1;
	private int[] arr;

	MyQueue() {
		SIZE = 10;
		arr = new int[SIZE];
	}

	MyQueue(int size) {
		SIZE = size;
		arr = new int[SIZE];
	}
}
```

Driver Class:

```
class MyQueueImplementation {
	public static void main(String...args) {
		MyQueue myQueue = new MyQueue();
		performEnqueue(myQueue);
		performDequeue(myQueue);
	}
}
```

Perform Enqueue Method:

```
public static void performEnqueue(MyQueue myQueue) {
      myQueue.enqueue(1);
      myQueue.printQueue();
      myQueue.enqueue(2);
      myQueue.printQueue();
      myQueue.enqueue(3);
      myQueue.printQueue();
      myQueue.enqueue(4);
      myQueue.printQueue();
      myQueue.enqueue(5);
      myQueue.printQueue();
      myQueue.enqueue(6);
      myQueue.printQueue();
      myQueue.enqueue(7);
      myQueue.printQueue();
      myQueue.enqueue(8);
      myQueue.printQueue();
      myQueue.enqueue(9);
      myQueue.printQueue();
      myQueue.enqueue(10);
      myQueue.printQueue();
      myQueue.enqueue(1);
      myQueue.printQueue();
}
```

Perform Dequeue Method:

```
public static void performDequeue(MyQueue myQueue) {
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
      System.out.println("Element Removed: " + myQueue.dequeue());
      myQueue.printQueue();
}
```

Output for reference:

```
FRONT -> 1 -> REAR.

FRONT -> 1 -> 2 -> REAR.

FRONT -> 1 -> 2 -> 3 -> REAR.

FRONT -> 1 -> 2 -> 3 -> 4 -> REAR.

FRONT -> 1 -> 2 -> 3 -> 4 -> 5 -> REAR.

FRONT -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> REAR.

FRONT -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> REAR.

FRONT -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> REAR.

FRONT -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> REAR.

FRONT -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> REAR.
Queue is full.

FRONT -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> REAR.
Element Removed: 1

FRONT -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> REAR.
Element Removed: 2

FRONT -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> REAR.
Element Removed: 3

FRONT -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> REAR.
Element Removed: 4

FRONT -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> REAR.
Element Removed: 5
```

```
FRONT -> 6 -> 7 -> 8 -> 9 -> 10 -> REAR.
Element Removed: 6

FRONT -> 7 -> 8 -> 9 -> 10 -> REAR.
Element Removed: 7

FRONT -> 8 -> 9 -> 10 -> REAR.
Element Removed: 8

FRONT -> 9 -> 10 -> REAR.
Element Removed: 9

FRONT -> 10 -> REAR.
Element Removed: 10

FRONT -> REAR.
Queue is empty.
Element Removed: -1

FRONT -> REAR.
```

XXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXX

TOPIC 6: RECURSION.

Recursion is basically a function calling itself.

Now let's say that we need to find the sum of n numbers.

Sum(n) = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + ... + n
Sum(n) = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + ... + (n-1) + n
Sum(n) = Sum(n-1) + n

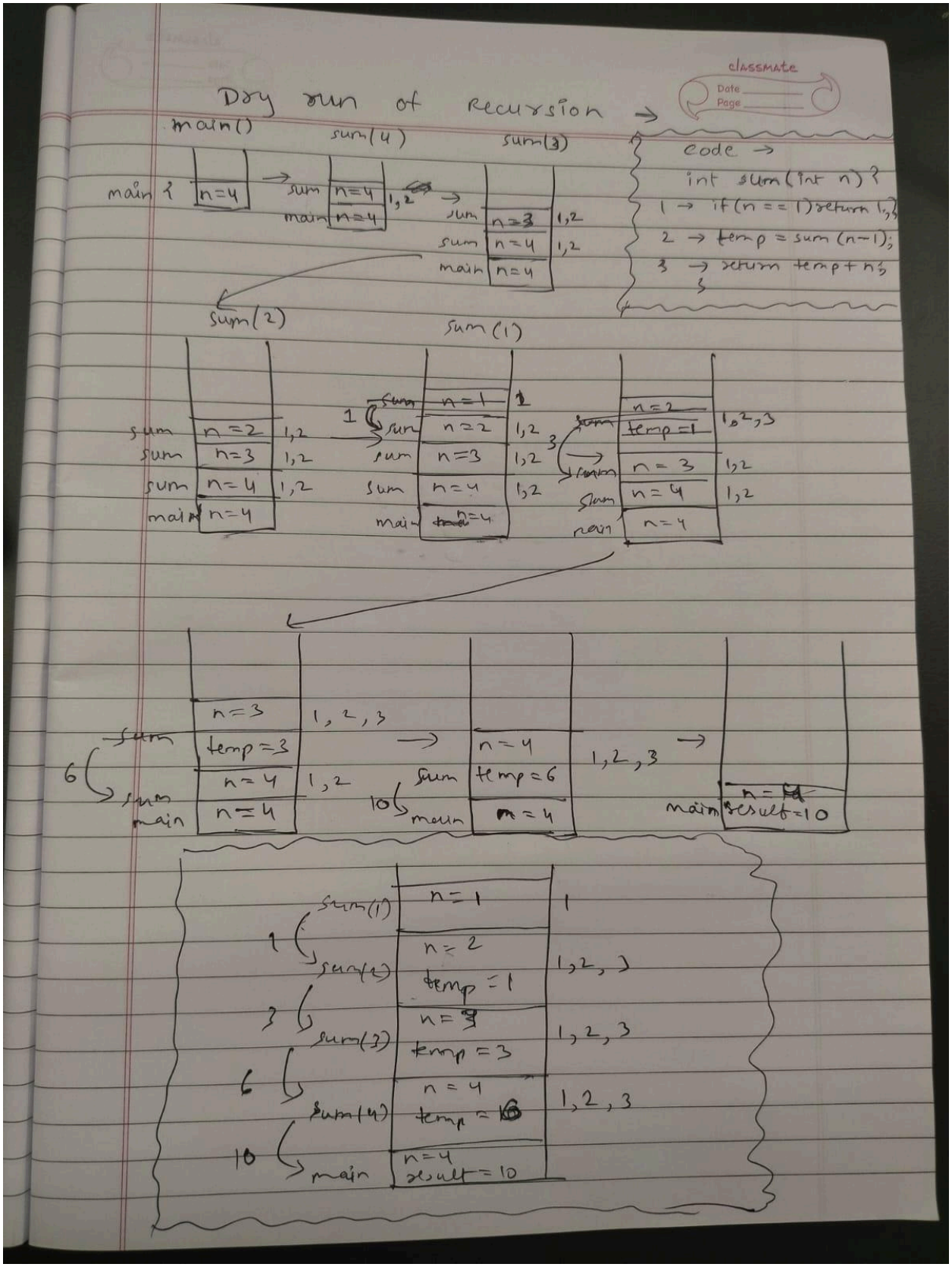
Sum(n-1) = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + ... + (n-1)
Sum(n-1) = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + ... + (n-2) + (n-1)
Sum(n-1) = Sum(n+2) + (n-1)

Sum(n-2) = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + ... + (n-2)
Sum(n-2) = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + ... + (n-3) + (n-2)
Sum(n-2) = Sum(n-3) + (n-2)
...
...
...
...
...
Sum(1) = Sum(0) [Which is basically 0] + 1
Sum(1) = 0 + 1

THREE MAGICAL STEPS FOR RECURSION:
1. Faith - Have faith that the function will work.
2. Main Logic - Solving your problem with a sub-problem.
3. Base condition - Solution to smallest sub-problem.

Q. Given n, find sum of numbers from (1 to n):

```
public int sum(int n) {
      if(n == 1) return 1;
      int temp = sum(n-1);
      return temp + n;
}
```


Dry run of Recursion →
main()
sum(4)
sum(3)
sum(2)
sum(1)
code →
int sum(int n){
1 → if(n==1) return 1;
2 → temp = sum(n-1);
3 → return temp+n;
}
n=4
n=3
n=2
n=1
temp=1
temp=3
temp=6
result=10
1,2
1,2,3
sum
main

Q. Find n factorial.

```
public int factorial(int n) {
      if(n == 2) return n;
      int temp = factorial(n-1);
      return temp * n;
}
```

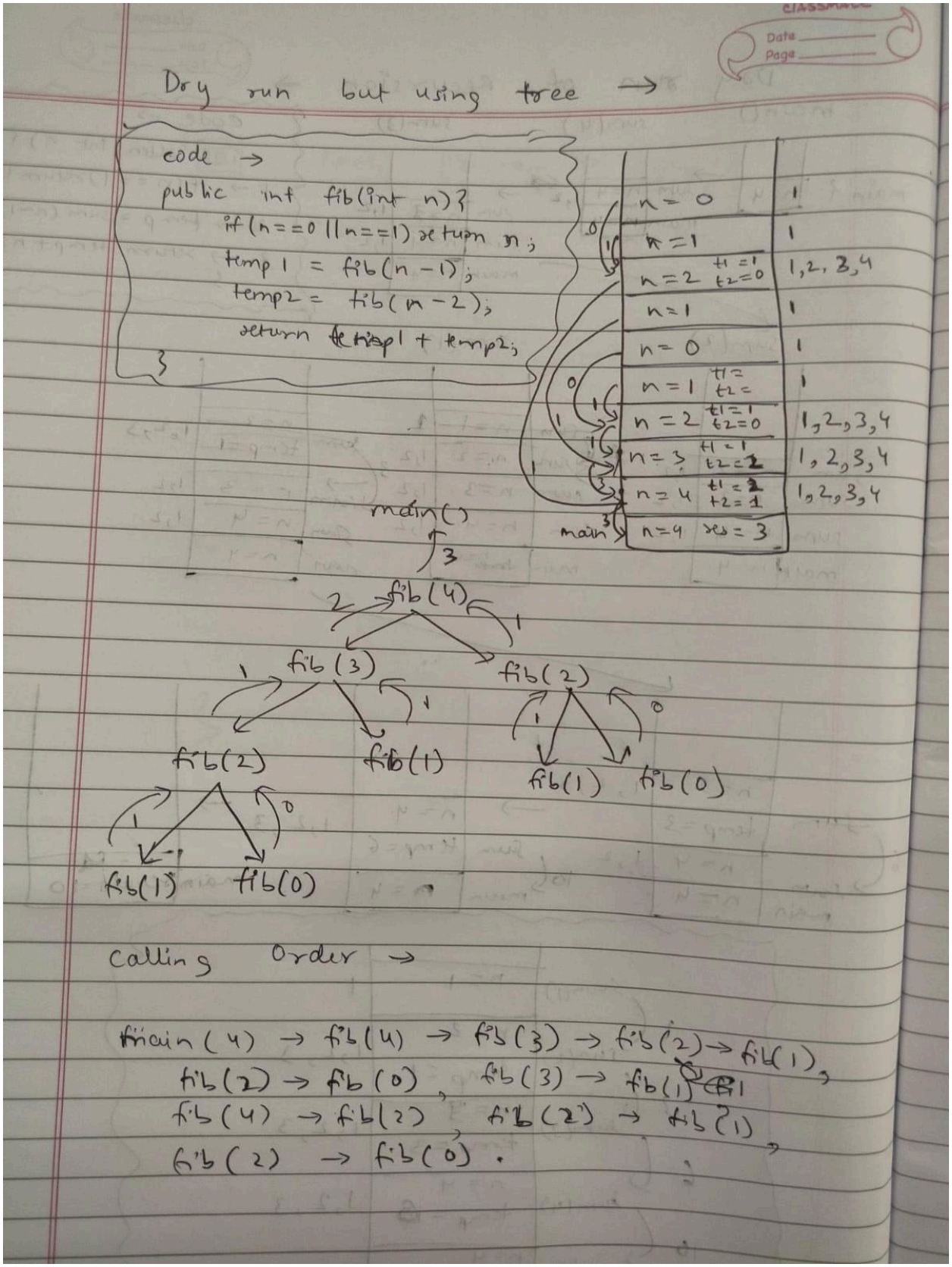
Q. Print 1 to n.

```
public void printAll(int n) {
      if(n == 1) {
            System.out.println(n);
            return;
      }
      printAll(n-1);
      System.out.println(n);
}
```

Q. Fibonacci Series. **[fib(n) = fib(n-1) + fib(n−2)]**

```
public int fibonacci(int n) {
      if(n == 0 || n == 1) return n;
      int temp1 = fibonacci(n-1);
      int temp2 = fibonacci(n-2);
      return temp1 + temp2;
}
```

The following is the dry run of the fibonacci problem using stack and a tree to make it more easier to understand while solving recursion problems:


Dry run but using tree →
code →
public int fib(int n){
if(n==0||n==1) return n;
temp1 = fib(n-1);
temp2 = fib(n-2);
return temp1 + temp2;
}
n=0
n=1
n=2 t1=1 t2=0
n=1
n=0
n=1
n=2 t1=1 t2=0
n=3 t1=1 t2=2
n=4 t1=2 t2=1
main
n=4 res=3
1,2,3,4
1,2,3,4
1,2,3,4
main()
fib(4)
fib(3)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
Calling Order →
main(4) → fib(4) → fib(3) → fib(2) → fib(1),
fib(2) → fib(0), fib(3) → fib(1)
fib(4) → fib(2), fib(2) → fib(1),
fib(2) → fib(0).

Q. Find the power of a ^ n.

```
public long power(int a, int n) {
      if(n == 1) return a;
      long temp = power(a, n-1);
      return temp * a;
}
```

Time Complexity Calculation for Recursion:

Function calls:

```
power(2, 4) -> power(2, 3) -> power(2, 2) -> power(2, 1)
```

Returns:

```
power(2, 1) -> 2 -> power(2, 2)
power(2, 2) -> 4 -> power(2, 3)
power(2, 3) -> 8 -> power(2, 4)
power(2, 4) -> 16 -> main()
```

```
Total function calls: 4 -> n.
Time complexity of each function: O(1).
Total time complexity: O(1) * n = O(n).
```

Q. Find the power of a ^ n with **improved Time Complexity [O(log n)]**:

```
public long power(int a, int n) {
      if(n == 1) return a;
      long temp = power(a, n/2);
      if(n % 2 == 0) return temp*temp;
      return temp*temp*a;
}
```

Space Complexity: O(n). At one given time at worst only n number of functions will be in the stack. Each function has space complexity of O(1) so finally it is O(1) * n = O(n).

Q. Find if Palindrome or not.

```java
public boolean isPalindrome(char[] ch, int s, int e) {
      //Base Condition.
      if(s == e) return true;

      //Keep checking from start till end.
      if(ch[s] == ch[e]) return isPalindrome(ch, s+1, e-1);

      //If a character doesn't match with another, return false.
      return false;
}
```

XXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXX

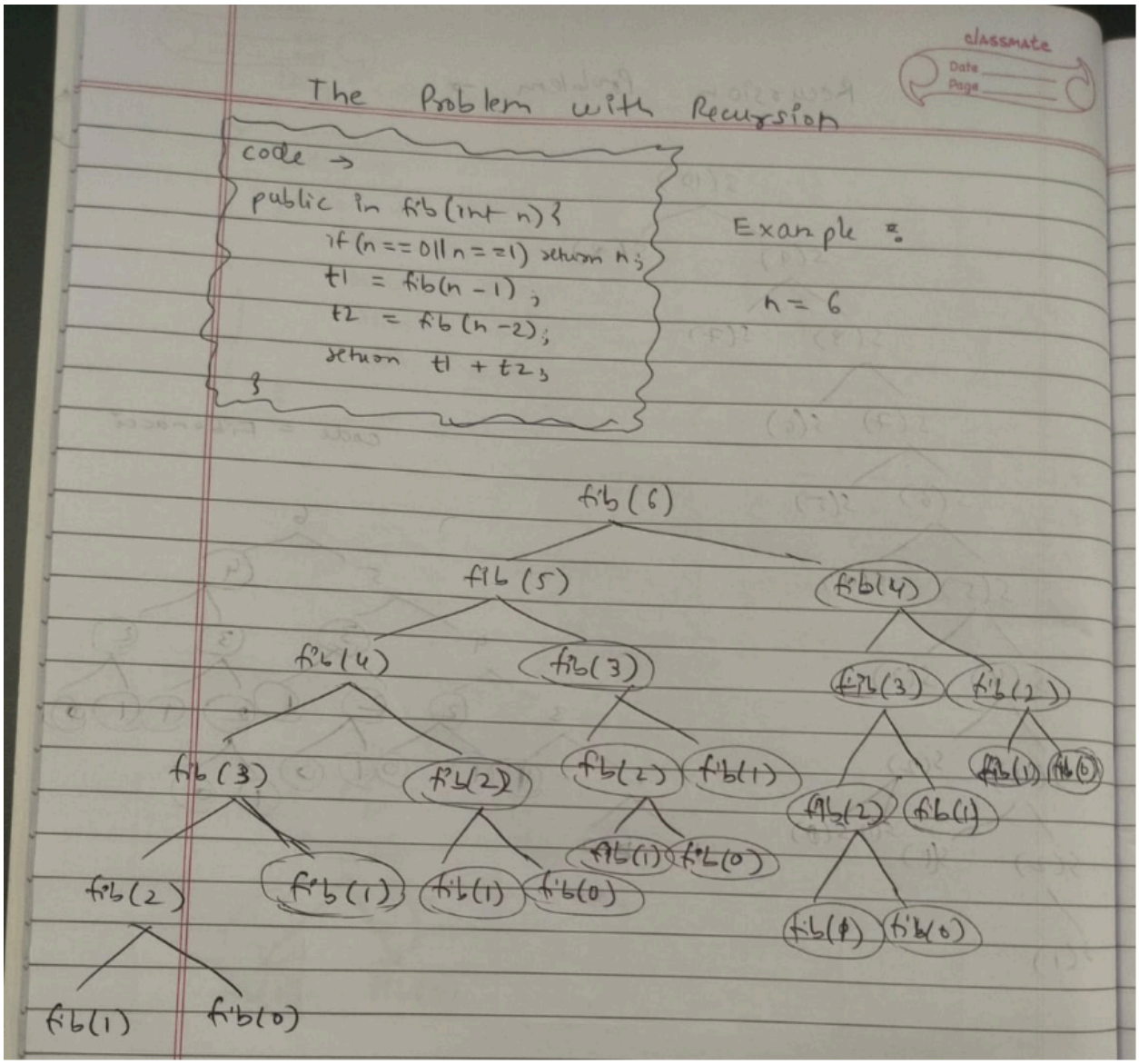
TOPIC 7: DYNAMIC PROGRAMMING.

DP is easy if not looking at hard hard questions where we need to make an array of 5 to 6 dimensions.

It is pretty much a simple add-on on recursion.

We use DP to solve the problem that we witnessed during recursion where we were calling the same functions multiple times.

The following circles the functions that are being called more than once:



To solve this problem we can use Dynamic Programming in the following way:

```
import java.util.Arrays;
import static java.lang.System.*;

class DSA {

	public static int fib(int n) {
		int[] dp = new int[n+1];
		Arrays.fill(dp, -1);
		return fibHelper(n, dp);
	}

	public static int fibHelper(int n, int[] dp) {
		if(n == 0 || n == 1) return n;
		if(dp[n] != -1) return dp[n];

		int temp1 = fibHelper(n-1, dp);
		int temp2 = fibHelper(n-2, dp);

		dp[n] = temp1 + temp2;
		return temp1 + temp2;
	}
}
```

Time Complexity got reduced from O(2^n) to O(n).
Space Complexity is O(n).

Fancy terms for referring DP:

Optimal Substructure: Dividing the problem into subproblems.
Overlapping subproblems: Same parts of problem repeating.

Important Question:

Q. n-stairs problem.
Given n, how many ways we can go from 0th - nth stair.
Note: You can take steps of length 1 or 2.

```
import java.util.Arrays;
import static java.lang.System.*;

class DSA {

	public static void main(String...args) {
		out.println("Number of ways to reach are " + stairs(10));
	}

	public static int stairs(int n) {
		int[] dp = new int[n+1];
		Arrays.fill(dp, -1);
		return stairsHelper(n, dp);
	}

	public static int stairsHelper(int n, int[] dp) {
		if(n == 0 || n == 1) return 1;
		if(n == 2) return n;
		if(dp[n] != -1) return dp[n];

		int temp1 = stairsHelper(n-1, dp);
		int temp2 = stairsHelper(n-2, dp);

		dp[n] = temp1 + temp2;
		return dp[n];
	}
}
```

There are several other DP problems you can solve. In this sheet only this problem is solved for now. The list of famous and most asked DP questions is attached below.

List of DP/Recursion Questions: (Solve at your own risk)

1. Nth Catalan Number.

2. Minimum Steps.

3. Minimum steps to delete a string after repeated deletion of palindrome substrings.

4. Find minimum number of coins to make a given value.

5. Maximum Product Cutting.

6. Count number of ways to cover a distance.

7. Minimum number of deletions and insertions to transform one string into another.

8. Minimum sum subsequence such that at least one of every four consecutive elements is picked.

XXXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXXX

TOPIC 8: TREES.

Naming Conventions in Tree:

Root Node: Top most node.
Edges: Lines connecting nodes.
Levels: Starting from Root Node (Level 0) it increases with each set of children.
There is a Parent Child relation between the nodes.
Leaf Node: Nodes without any children.
Siblings: Nodes with same parents.

Property of Tree:

There can only be 1 Root Node.
For every node there can be only 1 parent.
Cycles are not allowed in the Tree.

Height of a Node = Length of longest path from node to any of its descendent leaf nodes.

Depth of a Node = Length of path from given node to the Root Node.

In this section we will be focusing on the **Binary Tree.**

In Binary Tree every node can have **at max 2 childrens.**

```
class Node {
      int value;
      Node left;
      Node right;

      Node(int value) {
            this.value = value;
      }
}
```

How to construct a tree: (This won't be needed for solving questions on tree but it is good to know)

```
public void tree() {

      //Creates a node with value = 10.
      //left and right are set to null.
      Node node1 = new Node(10);

      //Creates a node with value = 20.
      //left and right are set to null.
      Node node2 = new Node(20);

      //Creates a node with value = 30.
      //left and right are set to null.
      Node node3 = new Node(30);

      //Sets the left and right nodes of node1.
      node.left = node2;
      node.right = node3;

      //Accesses the value of the left node of node1.
      System.out.println(node1.left.value);
}
```

Traversing Tree (Important):

There are mainly 4 ways of traversal in a tree:

1. Level Order Traversal (Breadth First Search).
2. Recursion Traversal (Depth First Search)
2.1 Preorder (NODE -> LEFT -> RIGHT).
2.2 Inorder (LEFT -> NODE -> RIGHT).
2.3 Postorder (LEFT -> RIGHT -> NODE).

Let's start with Recursion Traversal:

Basic Implementation (Common for all 3 types):

```
public void traversal(Node root) {
      if(root == null) return

      traversal(root.left);
      traversal(root.right);
}
```

For Preorder Traversal we will print the node value at the start:

```
public void preorder(Node root) {
      if(root == null) return

      System.out.println(root.value);
      traversal(root.left);
      traversal(root.right);
}
```

For Inorder Traversal we will print after left and before right:

```
public void inorder(Node root) {
      if(root == null) return

      traversal(root.left);
      System.out.println(root.value);
      traversal(root.right);
}
```

For Postorder Traversal we will print after left and right:

```
public void postorder(Node root) {
      if(root == null) return

      traversal(root.left);
      traversal(root.right);
      System.out.println(root.value);
}
```

Q. Calculate Size of a Tree (Total number of Nodes):

```
public int size(Node root) {
	if(root == null) return 0;

	int l = size(root.left);
	int r = size(root.right);
	return l+r+1;
}
```

Q. Calculate Sum of a Tree:

```
public int sum(Node root) {
	if(root == null) return 0;

	int lSum = sum(root.left);
	int rSum = sum(root.right);
	return lSum+rSum+root.value;
}
```

Level Order Traversal:

```
public void levelorder(Node root) {
	Queue<node> queue = new ArrayList<>();
	queue.add(root);

	while(queue.size() > 0) {
		Node temp = queue.remove();
		System.out.println(temp.value);
		if(temp.left != null) queue.add(temp.left);
		if(temp.right != null) queue.add(temp.right);
	}
}
```

A specialized case of level order where you have to print the values based on their levels.
So root node value will be on the first line.
Root Nodes childrens value on the next.
Root Nodes grand childrens value on the next and so on.

```
public void levelorder(Node root) {
      Queue<node> queue = new ArrayList<>();
      queue.add(root);
      while(queue.size() > 0) {
            n = queue.size();
            for(int i = 1; i < n; i++) {
                  Node temp = queue.remove();
                  System.out.println(temp.value);
                  if(temp.left != null) queue.add(temp.left);
                  if(temp.right != null) queue.add(temp.right);
            }
            System.out.println();
      }
}
```

Please use the following links for additional material on Trees:

50+ tree questions and solutions (easy, medium, hard) - IGotAnOffer

Top 25 Tree Data Structure Interview Questions and Answers - InterviewPrep

The following is the live code and the output of all the methods and more combined:

```
import java.util.*;
import static java.lang.System.*;

class DSA {
	public static void main(String[] args) {
		DSA object = new DSA();
		Node root = object.tree();
		object.traverse(root);
		int size = object.size(root);
		int sum = object.sum(root);
		out.println("Sum is " + sum + " and size is " + size);
		out.println("Level Order: ");
		object.levelorder(root);
		out.println("Modified Level Order: ");
		object.modifiedLevelorder(root);
		out.println("Whole Tree revered in Level Order: ");
		object.modifiedReversedLevelorder(root);
		out.println("Each Level reversed in Level Order: ");
		object.reverseLevelorder(root);
	}

	public Node tree() {
		Node root = new Node(10);
		Node node1 = new Node(20);
		Node node2 = new Node(30);
		Node node3 = new Node(40);
		Node node4 = new Node(50);
		Node node5 = new Node(60);
		Node node6 = new Node(70);
		Node node7 = new Node(80);
		Node node8 = new Node(90);
		Node node9 = new Node(100);
		Node node10 = new Node(110);
		Node node11 = new Node(120);
		Node node12 = new Node(130);
```

```
        Node node13 = new Node(140);
        Node node14 = new Node(150);

        root.left = node1;
        root.right = node2;

        node1.left = node3;
        node1.right = node4;

        node2.left = node5;
        node2.right = node6;

        node3.left = node7;
        node3.right = node8;

        node4.left = node9;
        node4.right = node10;

        node5.left = node11;
        node5.right = node12;

        node6.left = node13;
        node6.right = node14;

        return root;
    }

    public void traverse(Node root) {
        out.println("\nPreorder: ");
        traversePreorder(root);
        out.println("\nInoroder: ");
        traverseInorder(root);
        out.println("\nPostorder: ");
        traversePostorder(root);
    }
```

```
    public void traversePreorder(Node root) {
        if(root == null) return;

        out.println(root.value);
        traversePreorder(root.left);
        traversePreorder(root.right);
    }

    public void traverseInorder(Node root) {
        if(root == null) return;

        traverseInorder(root.left);
        out.println(root.value);
        traverseInorder(root.right);
    }

    public void traversePostorder(Node root) {
        if(root == null) return;

        traversePostorder(root.left);
        traversePostorder(root.right);
        out.println(root.value);
    }

    public int size(Node root) {
        if(root == null) return 0;

        int l = size(root.left);
        int r = size(root.right);
        return l+r+1;
    }

    public int sum(Node root) {
        if(root == null) return 0;

        int lSum = sum(root.left);
        int rSum = sum(root.right);
        return lSum+rSum+root.value;
```

```
    }

    public void levelorder(Node root) {
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);
        while(queue.size() > 0) {
            Node temp = queue.remove();
            out.println(temp.value);
            if(temp.left != null) queue.add(temp.left);
            if(temp.right != null) queue.add(temp.right);
        }
    }

    public void modifiedLevelorder(Node root) {
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);
        while(queue.size() > 0) {
            int n = queue.size();
            for(int i = 0; i < n; i++) {
                Node temp = queue.remove();
                out.println(temp.value);
                if(temp.left != null) queue.add(temp.left);
                if(temp.right != null) queue.add(temp.right);
            }
            out.println();
        }
    }

    public void modifiedReversedLevelorder(Node root) {
        ArrayList<Integer> values = new ArrayList<>();
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(root);
        while(queue.size() > 0) {
            int n = queue.size();
            for(int i = 0; i < n; i++) {
                Node temp = queue.remove();
                values.add(temp.value);
                if(temp.left != null) queue.add(temp.left);
                if(temp.right != null) queue.add(temp.right);
```

```
                }
            }
            Collections.reverse(values);
            out.println(values);
        }

        public void reverseLevelorder(Node root) {
            List<List<Node>> levels = new ArrayList<>();
            Queue<Node> queue = new LinkedList<Node>();
            queue.add(root);
            while(queue.size() > 0) {
                int n = queue.size();
                List<Node> eachLevel = new ArrayList<>();
                for(int i = 0; i < n; i++) {
                    Node temp = queue.remove();
                    eachLevel.add(temp);
                    if(temp.left != null) queue.add(temp.left);
                    if(temp.right != null) queue.add(temp.right);
                }
                levels.add(eachLevel);
            }
            Collections.reverse(levels);
            for(List<Node> each : levels) {
                for(Node eachNode : each) {
                    out.println(eachNode.value);
                }
                out.println();
            }
        }
}

class Node {
        int value;
        Node left;
        Node right;

        Node(int value) {
            this.value = value;
        }
```

```
}
```

Output for reference:

```
Preorder:
10
20
40
80
90
50
100
110
30
60
120
130
70
140
150

Inoroder:
80
40
90
20
100
50
110
10
120
60
130
30
140
70
150

Postorder:
80
```

```
90
40
100
110
50
20
120
130
60
140
150
70
30
10

Sum is 1200 and size is 15

Level Order:
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150

Modified Level Order:
10

20
30
```

```
40
50
60
70

80
90
100
110
120
130
140
150

Whole Tree revered in Level Order:
[150, 140, 130, 120, 110, 100, 90, 80, 70, 60, 50, 40, 30, 20, 10]

Each Level reversed in Level Order:
80
90
100
110
120
130
140
150

40
50
60
70

20
30

10
```

XXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXX

BY GAURAV AMARNANI.

TOPIC 9: GRAPHS.

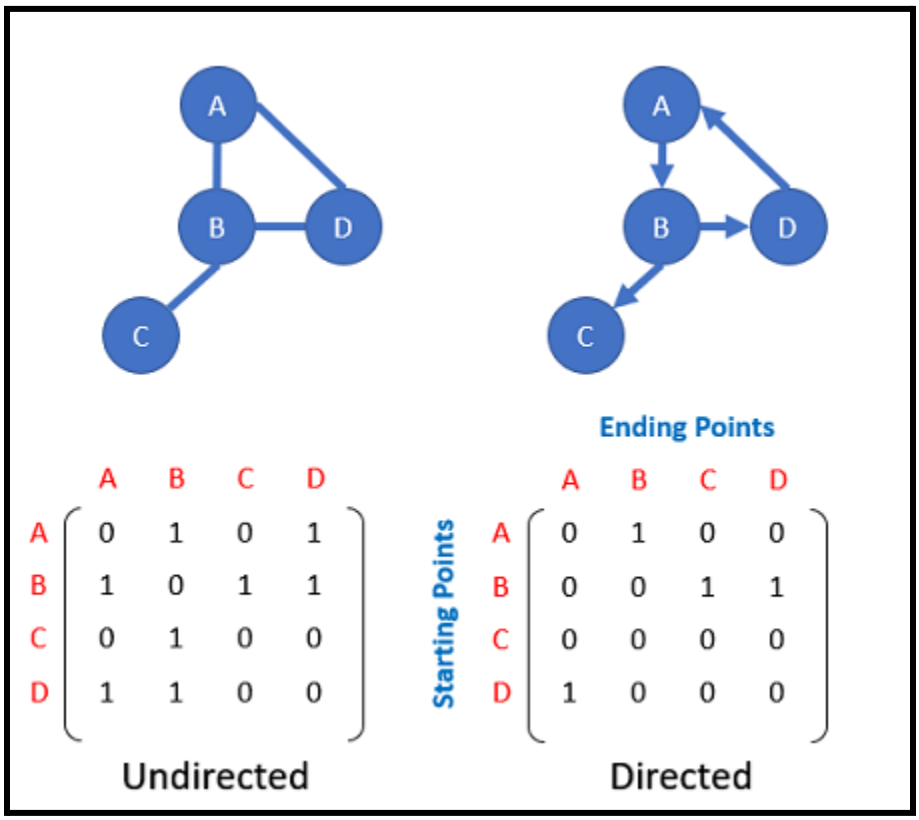
Graphs can be defined as a connection of nodes and edges.

Major differences between Trees and Graphs:

1. Nodes in the graph can have more than one parent.
2. Graph has no hierarchy or root node.
3. Graph can have cycles.
4. Any directional movement is allowed in the graph.

Graphs are usually represented using 2 ways:

1. Adjacency Matrix: (1 is connected and 0 is disconnected)



2. Adjacency List.

We will be using the adjacency list in our examples as majorly all the leetcode questions provide an adjacency list.

We will use a list of lists for this adjacency list representation in Java.

```
List<List<Integer>> graph = new ArrayList<>();
```

We will need the number of nodes to create this list as the number of inner lists will be the same as the number of nodes.
So basically the inner lists will contain all the nodes that are neighbors to the particular list number.

For our own convenience (and generally in easy questions) we will consider that the number of nodes will be from 0 to N.

So the list will look something like this if there are 5 nodes:

[
[1, 2] //For Node 0.
[0, 3] //For Node 1.
[0] //For Node 2.
[1, 4] //For Node 3.
[3] //For Node 4.
]

So from above you can understand that Node 0 has Node 1 and Node 2 as neighbors, Node 1 has Node 0 and Node 3 as neighbors, Node 2 had only Node 0 as neighbor, Node 3 has Node 1 and Node 4 as neighbors and finally Node 4 has only Node 3 as neighbor.

The following is the code that will help us create a Graph by taking input from the user directly with help or array and List of Lists of Nodes.

```
import java.util.*;

class DSA {
	public static void main(String...args) {
		Scanner scanner = new Scanner(System.in);
		int n = scanner.nextInt(); //Nodes
		int m = scanner.nextInt(); //Edges

		int[][] edges = new int[m][2];

		for(int i = 0; i < m; i++) {
			edges[i][0] = scanner.nextInt();
			edges[i][1] = scanner.nextInt();
		}

		new DSA().construction(n, m, edges);
	}

	public void construction(int n, int m, int[][] edges) {
		List<List<Integer>> graph = new ArrayList<>();
		for(int i = 0; i < n; i++) {
			graph.add(new ArrayList<>());
		}

		for(int i = 0; i < m; i++) {
			int u = edges[i][0];
			int v = edges[i][1];

			graph.get(u).add(v);
			graph.get(v).add(u);
		}
	}
}
```

Here we have done graph.get(u).add(v) and graph.get(v).add(u) because if a node u has neighbor node v so node v also has node u as neighbor.

Q. Find the maximum number of edges if n nodes are present in the graph.
[(n * (n-1))/2]

Q. BFS Traversal in Graph.

(In graph traversal we have to decide the start point)

```
public void bfs(int n, int m, int[][] edges) {
      List<List<Integer>> graph = construction(n, m, edges);
      Queue<Integer> queue = new LinkedList<>();
      boolean[] visited = new boolean[n];
      q.add(0);
      visited[0] = true;

      while(queue.size() > 0) {
      int rem = queue.remove();
            System.out.println(rem);
            List<Integer> neighbors = graph.get(rem);
            for(int v : neighbors) {
                  if(visited[v] == false) {
                        q.add(v);
                        visited[v] = true;
                  }
            }
      }
}
```

Here we are basically selecting a root node ourselves and then using the same concept as the tree to iterate but the only difference is that while iterating we are keeping the track of all the visited nodes with help of the boolean array visited and keeping it false at the start and changing it to true as soon as the node is getting visited.
So even if the node is getting added in the queue again and again it won't be printed as visited is already set to true.

Q. Given undirected graph, source node and destination node. Check if the destination node can be visited from the source node or not.

```
public void visit(int n, int m, int[][] edges, int src, int dest) {
      List<List<Integer>> graph = construction(n, m, edges);
      Queue<Integer> queue = new LinkedList<>();
      boolean[] visited = new boolean[n];
      q.add(src);
      visited[src] = true;
      while(queue.size() > 0) {
            int rem = queue.remove();

            List<Integer> neighbors = graph.get(rem);
            for(int v : neighbors) {
                  if(visited[v] == false) {
                        q.add(v);
                        visited[v] = true;
                  }
            }
      }
      return visited[dest];
}
```

Refer the following for more details:

[50+ graph interview questions and cheat sheet - IGotAnOffer]

XXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXX