MSQs

1. DSA 075 JAVA 5 100
2. Yes
3. O(nlog(max(n,m)))
4. O(2^n)
5. Runtime Error
6. n-1
7. Yes
8. Insertion Sort
9. O(nlogn) , O(n^2)
10. O(n)
11. Memory usage in linked list is less for same size
    Linked list are faster than arrays
    Random access of elements could be possible in linked list
12. Deleting a node whose location is given
    Reversing a list
13.
```
for(int j=n; j>=1; j/=2){
        for(int i = 0; i<j ; i*=2){
                System.out.println("hello world");
        }
}

int fun(int x,int y){
        if(y==0) return x;
        return fun(y,x%y);
}
```

Subjective

Soln 1: [10 marks: Full points for all conditions written correctly, deduct 25% if stack not reversed at the end]
[Only 50% if any other data structure is used, or complexity exceeds O(n)]

```
public Stack mergeStacks (Stack A , Stack B){
        Stack C = new Stack();
        while(!A.empty() && !B.empty()){
                if(A.top() < B.top() ) {        C.push(A.pop());     }
```

```
            else                        {          C.push(B.pop());      }
        }
        while(!A.empty()){
            C.push(A.pop());}
        while(!B.empty()){
            C.push(B.pop());}
        Stack D = new Stack();
        while(!C.empty()){
            D.push(C.pop());}
        return D;
}
```

Soln 2:

[Marking : 3 + 3 + 7 = 13 Marks]

(a) Stack [1 mark]
Since the plates will be stacked on top of each other hence we can use stack for it. [2 marks]
Or any valid explanation.

(b) Queue [1 mark]
Since without partiality we need to develop a system for first come first serve, queue is suitable as it follows FIFO. [2 marks]5.5
Or any valid explanation.

(c) We stack the plates on top of each other and arrange the customers in the queue. If the plate on top of the stack is veg and the first customer is also veg we pop the plate and dequeue the customer. Similarly, if the plate is non-veg and the customer is non-vegetarian we pop the plate and remove the customer. However, if there is no matching we dequeue the customer and then enqueue them again, ie, we put the present customer at the end of the queue.

Soln 3:

[Marking: 7 Marks for Writing the pseudo code + 3 Marks for writing recurrence]

At each step you have 2 options

1. Include the current chocolate, then you cannot include the next chocolate so you call the function for index+2 and add the sweet value of the current chocolate to the result.

2. Exclude the current chocolate, then you can choose to include/exclude the next chocolate so you call the function for index+1.

The max of these 2 options gives us the maximum sweet value.

```
int maxSweetValue(int sweetValue[], int index){
      if(index>= sweetValue.length) return 0;

      int includeCurrent = maxSweetValue(sweetValue,index+2) + sweetValue[index];
      int excludeCurrent = maxSweetValue(sweeValue,index+1);

     return Math.max(includeCurrent, excludeCurrent);
}
```

Recurrence Relation: $T(n) = T(n-2) + T(n-1) + c$
$T(1) = c$

Soln 4:

[Marking: 6 Marks, 3 + 3, (1 for correct answer + 2 for explanation)]

1. O(n)

To rotate the linked list one time when we are given the pointer to the head of the linked list we'll have to traverse the list till the second last node (O(n)) and set its next pointer to null. Then we make the last node as the new head and make it point to current head((O(1)). Hence the overall complexity is O(n).

2. O(n)(Optimal) or O(nk)

Note: Students do not need to write the code, any valid explanation of the algorithm/ complexity will work.

Algorithm O(n)

Rotate the linked list right k times.

1. Find the length of the linked list and the last node.
2. Divide k by the length

3. Set the last node next pointer to point to head.
4. Now we find the starting node of the rotated linked list which is going to be the (len-k)th node from the beginning.
5. Now set the tail to again point to null.

```
ListNode rotateList( ListNode head, int k){
      if(head==null || head.next ==null) return head;

      int len = 1;
      ListNode ptr = head;
      while(ptr.next !=null){
            len++;
            ptr = ptr.next;
      }

      k%=len;

      ptr.next = head;
      for(int i=0;i< len-k;i++){
            ptr = ptr.next;
      }
     head = ptr.next;
     ptr.next = null;

     return head;
}
```

Soln 5: [(2+3) + 3 + 3 = 11 Marks]

2 → For Writing Merge Sort
3 → Explanation
3 → For Writing Recurrence
3 → Solving Recurrence & Finding time complexity

[Marking Scheme: 50% Marks if the solution is partially correct]

1. Merge Sort
Explanation:
We need to find the numbers of pairs in an array arr such that j>i and arr[i] > arr[j]. This can be done optimally using merge sort. Merge sort follows a divide and conquer

strategy. We can divide our array into 2 subarrays and count the number of inversions in the left array + number of inversions in the right array + number of inversions where one element is in the left subarray and other is in the right subarray. In other words,

Number of inversions
= mergeSort(arr, start, mid) + mergeSort(arr, mid+1, end) + merge(arr,start,mid,end)

Note: Marks should be given for any valid explanation

2. $T(n) = 2\,T(n/2) + O(n)$, $T(1) = O(1)$
3. Solving the recurrence

Using Master's Theorem
$T(n) = aT(n/b) + cn^k$, $T(1) = c$
$T(1) = c$

$a = 2$, $b = 2$, $c=1$, $k = 1$
$b^k = 2$, hence we have $a = b^k$

$T(n) = O(n^k \log n) = O(n \log(n))$

Note: Any method can be used for solving the recurrence.

Soln 6:

Marking Scheme: 10 marks [Full for O(1) soln] as well as for O(n)
~~50% for O(n) soln~~, deduct 2 marks each for any missing base case.

```
ListNode concatenate( ListNode A, ListNode B){
        if(A ==null){
                return B;
        }
        if(B == null){
                return A;
        }
        ListNode temp = A.nxt;
        A.nxt = B.nxt;
        B.nxt = temp;
        return B;
}
```

Soln 7:

[+2 for each correct Line and +1 for each correct blank] [Total 10 marks]

We can use Stack/Queue.

```
boolean find_path( int mat[][], int i, int j, int n){
        Stack<int[]> stack = new Stack<>();
        stack.add( {i, j} );              // Line 1 or { j, i} is also fine
        while(stack.isEmpty() == false){
                int ele[] = stack.pop();
                // Blank 1 and Blank 2
                if(ele[0] == n-1 && ele[1]==n-1){   return true;}
                if(ele[0]+1 < n && mat[ele[0] +1][ele[1]]==1)      //Blank 3
                        { stack.add( {ele[0]+1, ele[1]} ); }          //Line 2
                if(ele[1]+1 < n && mat[ele[0]][ele[1]+1]==1)       //Blank 4
                        { stack.add( {ele[0], ele[1]+1} ); }          // Line 3

        }
        return false;
}
```