Instructions:
For every subpart the following points are mandatory:
1. Recurrence Relation and complexity derivation using Master's theorem
2 What each variable stands for
3 Pseudo Code
4 Name of the algorithm used, and how the answers of subproblems are used to find the answer of the main problem.

Q1 Count of range sum.

Mickeymouse loves numbers between two numbers -> m1, and m2. Minnie decides to gift Mickey a sequence of numbers for his birthday. Mickey wants to find the number of pairs of indices [l, r] for which the sum of elements in the range [l, r] lies between m1 and m2.

Recurrence relation not needed in first two parts. But do explain the reason for the resulting time complexity in words.

1. Come up with an algorithm with a worst case complexity of $O(N^2)$.
2. Now improvise this algorithm, assuming all numbers are positive integers.
3. What if the numbers could be negative as well? Can you think of an O(nlogn) solution in this case?

Answer:

1.
   Step1: Make a prefix sum array, where prefix[index] stores sum of elements of range (0,index) (including index). Initiate a variable ans = 0;
   Step2: Using two for loops, for each indice l , find next indice r, such that prefix[r] - prefix[l] belongs to [m1, m2]. Everytime this is true, ans += 1;
   Step3: return ans;

   As there are two nested for loops, the complexity is $O(n^2)$
   Explanation: 2 marks →prefix sum array, for loops (explaining complexity) (1 mark per keyword)

```
PSEUDO CODE:
fun(){
Int[] prefix  = new int[array.length];
Prefix[0] = array[0];
        For (int i = 1; i < array.length; i++){
            Prefix[i] = prefix[i - 1] + array[i];
        }
Int ans = 0;
For (int i = 0; i < prefix.length; i++){
        For (int j = i + 1; j < prefix.length; j++){
            If (prefix[j] - prefix[i] >= m1 && prefix[j] - prefix[i] <= m2){
                Ans += 1;
            }
        }
}
For (int i = 0; i < prefix.length; i++){
        If (prefix[i] >= m1 && prefix[i] <= m2){ans += 1;}
}  /// We are taking the case where all elements upto index i are included.
print(ans);
}
```

Pseudo code: 3 marks → binary marking
As there are two nested for loops, therefore complexity is O(n2)

2.
As all numbers are positive integers, this implies that the for any two indices i and j, if i >= j then prefix_sum[i] >= prefix_sum[j].
So,, we know that prefix_sum is a sorted array. Utilising this knowledge, we can use binary search.
For every index l, search for two indices r1, and r2 such that prefix[r1] - index[l] < m1 and prefix[r2] - prefix[l] <= r1. Then the answer is r2 - r1.
So for every index l, this requires 2 binary searches on rest of the array. So the time complexity is O(2*nlogn) or O(nlogn).
Explanation → 2 marks (why binary search and prefix array→ 1 mark, explanation of complexity -> O(nlogn))

Pseudo Code:

```
Int binsearch1(int start, int end, int[] array, int u){
        If (end < start) return -1;
        If (start == end){
        If (array[start] <= u){return start;}
        Return -1;
}
Int mid = (start + end)/2;
If (array[mid] <= start]){
Int u1 = binsearch1(mid + 1, end, array, u);
If (u1 == -1)return mid;
Return u1;
}
Else {
Return binsearch1(s, mid - 1, array);
}
}



main(){

//calculate prefix sum array


Int ans = 0;
//for loop
For (int i = 0; i < prefix.length; i++){
Int r1  = binarysearch1(0, prefix.length  - 1, prefix, m1 - prefix[i] - 1); // find the last index with
prefix_sum[index] <= m1
Int r2  = binarysearch1(0, prefix.length  - 1, prefix, m2  - prefix[i]);
Ans += r2 - r1;
}
print(ans);

}
```

PSEUDO CODE: 5 marks → binsearch1 (3 marks → binary). 2 marks → for loop and answer increment.

3.

We can use mergesort. So essentially we are finding answer for each half. And sorting each half. Now we only need to find answer for ranges lying intersecting with the left and right range. To do that we will find answer for <= upper, then < lower and subtract both. Now we will merge both arrays into one sorted array

EXPLANATION: 3 marks → 1 mark for explanation of subproblem and 1 mark for explanation of merging two answers and sorting them.

Recurrence relation:

$T(n) = 2*T(n/2) + n$ → 2 marks

Complexity: nlogn ---> With complete derivation → 2 marks

```
class Solution {
    public int countRangeSum(int[] nums, int lower, int upper) {
        int n = nums.length;
        long[] prefix = new long[n];
        prefix[0] = nums[0];
        for (int i = 1; i < n; i++){prefix[i] = nums[i] + prefix[i - 1];}
        return mergesort(prefix, 0, n - 1, lower, upper);
    }
    public int mergesort(long[] prefix, int s, int e, int low, int up){
        if (s == e){
            if (low <= prefix[s] && prefix[s] <= up)return 1;
            return 0;
        }
        int mid = (s + e)/2;
        int a = mergesort(prefix, s, mid, low, up);
        int b = mergesort(prefix, mid  + 1, e, low, up);
        int ans= a + b;
        int j1 = mid + 1;
        int j2 = mid + 1;
        for (int i = s; i <= mid; i++){
            while (j1 <= e && prefix[j1] - prefix[i] <low){j1 += 1; }
            while (j2 <= e && prefix[j2] - prefix[i] <= up){j2 += 1; }
            ans += (j2 - j1);
        }
        merge(prefix, s, mid, e);
        return ans;

    }
    public void merge(long[] arr, int s, int mid, int e){
        long[] ref = new long[e - s + 1];
        int i= s; int j = mid + 1;  int k = 0;
        while (i <= mid && j <= e){
            if (arr[i] < arr[j]){
                ref[k] = arr[i]; i += 1;
            }
            else {
                ref[k] = arr[j]; j ++;
            }
            k ++;
        }
        while (i <= mid){
            ref[k] = arr[i];
            k ++ ; i++;
        }
        while (j <= e){
            ref[k] = arr[j];
            k ++ ; j ++;
        }
        for (int aj = 0; aj < e - s + 1; aj++){
            arr[aj + s] = ref[aj];
        }

    }
}
```

PSEUDO CODE → 5 marks →
mentioning subproblems -> 1 mark
Using answer of both subproblems to get answer for the main problem -> 2 marks
Merging the two problems to get a sorted array -> 2 marks -> binary

Recurrence relation: $T(n) = 2*T(n/2) + O(n)$ → 1 mark

Complexity: nlog(n) ----------------> Derivation using Substitution method or any other method covered in class. → 1 mark

Q2

Statement (Original):

Rahul lives in City A and would like to travel to City B for work. There is a shuttle service for people in these cities which has a fixed schedule. The schedule for City A is a list of boarding times(at City A) and departure times(from City A) for each bus.
Note: No one is allowed to board a shuttle after the boarding time.
He arrives at time t and sometimes has to wait at the station. You are given a list of arrival times for n days.

a) Devise an algorithm to find him the shuttle with the least waiting time. (waiting time = boardingj - t , where j is the next shuttle. And boardingj >= t) for each ti
b) If he also has access to the travel time of each shuttle. Can you help him find the shuttle which will help him reach his destination at the earliest ?

Return an array of shuttle indexes that Rahul took for n days.
Note: If there is no such shuttle return -1.

Marking: Total 20 points.

a.
  i) +5 points correct algorithm
  ii) +5 points if the algorithm presented is the optimal solution and complexity is derived correctly.

b.
  i) +5 points correct algorithm
  ii) +5 points if the algorithm presented is the optimal solution and the complexity is derived correctly.

a)
Algorithm :

1) Sort input boarding and departure times by boarding times :
2) For each t in input, use binary search to get the shuttle with boarding time >= t.

PseudoCode:

```
procedure merge(int schedule[][], int left, int mid, int right)
```

```
    int len1 = mid - left + 1;
    int len2 = right - mid;

    int leftArr[0....len1];
    int rightArr[0...len2];

    for (int i = 0; i < len1; i++)
      leftArr[i] = schedule[left + i];
    for (int j = 0; j < len2; j++)
      rightArr[j] = schedule[mid + 1 + j];

    int i, j, k;
    i = 0;
    j = 0;
    k = left;

    while (i < len1 && j < len2)
      if (leftArr[i][0] <= rightArr[j][0])
        schedule[k] = leftArr[i];
        i++;
      else
        schedule[k] = rightArr[j];
        j++;
      k++;

    while (i < len1)
      schedule[k] = leftArr[i];
      i++;
      k++;

    while (j < len2)
      schedule[k] = rightArr[j];
      j++;
      k++;
  }
procedure sort(int schedule[][], int start, int right)
    if (start < right)
      int mid = start + (right-start) / 2;
      sort(schedule, start, mid);
      sort(schedule, mid + 1, right);

      sort(schedule, start, mid, right);
```

```
procedure bin_search(schedule[][]: input array, int t: target value)
    int mid, ans = -1;
    while(l<=r)
        mid = l + (r-l)/2;
        int boarding = schedule[mid][0];

        If (boarding>=t)
            r = mid-1;
            ans = schedule[mid][2]; //original index
        Else
            l = mid +1;
        Endif

    return ans;

procedure Main (schedule[][]: schedule 2d vector, input_t[]: arrival times)
    int n = schedule.size();
    int T = input_t.size();
    for (int i = 0;i < n;i++) schedule.push_back(i); // store original index
    sort(schedule[0...n-1]);  // based on first element
    vector<int> output(n,0);
    for(int i = 0;i < T;i++)
        output[i] = bin_search(schedule,input_t[i]);

    return output;
```

Complexity:

1) O(n logn + t logn)

b)

Algorithm:

1) sort according to boarding time
2) you can pre calculate the index of the shuttle with the minimum reach-time out of all shuttles with boarding time >= boarding time of current shell (this should be done for all shuttles)
   Basically this will pre determine the choice of shuttle that Rahul should take if he reaches just before boarding time of some arbitrary shuttle.
3) for each t use binary search to get shuttle with boardingj >= t
4) now find the min-reach-time shuttle from the arr[j .... n-1] using preCalc[] or a min range tree.

Pseudo Code :

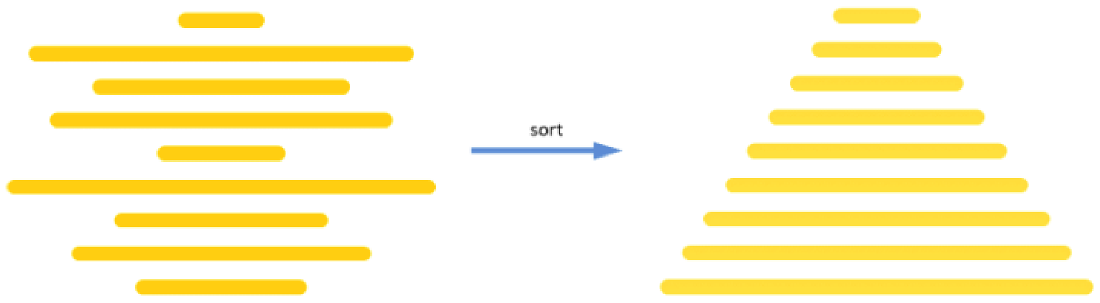
```
procedure bin_search(schedule[][]: input array, int t: target value, preCalc[]: shuttle index with
minimum reachTime  )
    int mid, ans = -1;
    while(l<=r)
        mid = l + (r-l)/2;
        int boarding = schedule[mid][0];

        If (boarding>=t)
            r = mid-1;
            ans = preCalc[mid]; //original index
        Else
            l = mid +1;
        Endif

    return ans;

procedure Main (schedule[][]: schedule 2d vector, input_t[]: arrival times, travelTime[]:
travelling time from A to B for all n shuttles )
    int n = schedule.size();
    int T = input_t.size();
    Array preCalc [0....n-1];
    for (int i = 0;i < n;i++) schedule.push_back(i); // store original index
    sort(schedule[0...n-1]);  // based on first element (boarding time)
    for(int i = n-1; i >=0; i--)
        int reachTime = travelTime[i]+schedule[i][1];   // time shuttle reaches B
        int mn = INT_MAX
        int mn_idx = -1
        if (reachTime<=mn)
            mn_idx = schedule[i][2]; // original index
            mn = reachTime;
       endif
        preCalc[i] = mn_idx; // pre calculate and find the index of the shuttle with the minimum
reach-time out of all shuttles with boarding time >= current boarding time
    Array output[0...n-1]
    for(int i = 0;i < T;i++)
        output[i] = bin_search(schedule,input_t[i], preCalc);

    return output;
```

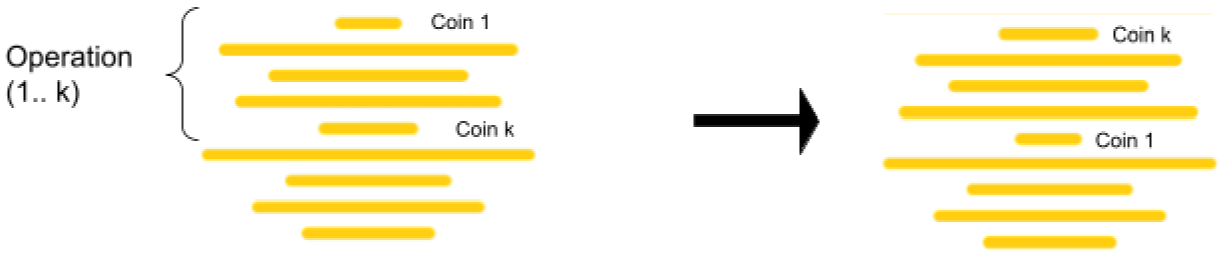
Complexity:

1) O(n logn + t logn)

Q 3: Sorting the Coin pile

Consider a pile of coins. You have given ‘n’ coins of different sizes and you want to sort the pile so that smaller coins are on the top of larger ones. The only “operation” you are allowed to perform is- take top ‘k’ coins together, and place them back to the pile upside down (See fig for reference).





Example Showing the inversion operation performed on the pile.

a) Describe an algorithm to sort an arbitrary pile of n coins using O(n) operations. Exactly how many operations does your algorithm perform in the worst case.
b) For every positive integer ‘n’, describe the pile structure that requires Omega(n) operations to sort. In other words, generate an example of a pile that will require at least cn operations for some c > 0. (n > 3)
c) Now suppose in the sorted array we want all coins to have heads face up. Describe an algorithm to sort an arbitrary pile of ‘n’ coins, such that all coins have tails face down, using O(n) operations. Exactly how many operations does your algorithm perform in the worst case. (No pseudo code required for this subpart, just explain the changes in algorithm and give the worst case analysis)

Note: Assume that the coin information is given to you in the form of an ‘n’ element array A. A[i] is a number between 1 and n and A[i] = j means that the j’th smallest pancake is in position i from the top; in other words A[1] is the size of the top most coin (relative to the others) and A[n] is the size of the bottommost coin.
operation(1, n) should be assumed to be O(1) and can be used directly in pseudo code.

Marking: Total 30 points.

a) (5 points, Algorithm + 5 points Psuedo Code + 5 points worst case analysis and recurrence)
b) (5 points, Binary Marking)
c) (5 points, algorithm + 5 points, worst case analysis and recurrence)

Solution:

a)

Algorithm:

If the size of the pile is 1, we're done.
Else, bring the largest coin to the top, then bring it to the bottom.
Repeat this algorithm on the top n − 1 smallest coins.

Algorithm → 5 marks →
mentioning subproblems -> 1 mark
Describing how to solve given problem using subproblem -> 2 marks
Using defined operation to solve the problem-> 1 mark -> binary
Answer to the original problem: 1 marks

Pseudo Code

PSEUDO CODE → 5 marks →
mentioning the end case -> 1 mark
Using answer of subproblem to get answer for the main problem -> 2 marks
Using the defined operation properly to solve the problem -> 2 marks -> binary

Answer to original problem: sort(arr, n);

```
Void sort( int coins[], int n)
        If ( n==1):
                Return          //Base case
        End if
        Int index;
        for i in 1 to n:
                If coins[i] == n:
                        Index = i
                End if
        End for
        If ( index != n):
                Operation ( coins, index)       // Bring largest coin to the top
                operation( coins, n)            // Bring largest coin to the bottom
        End if
        sort( coins, n-1)
        return
End sort
```

Recurrence Relation:

T(n) = Number of operations for array of size ‘n’

$T(n) <= T(n-1) + 2c$ 1 mark

$T(1) = 0$; 1 mark

On solving using substitution we get

$T(n) = O(n)$ 1 mark

Number of operations in the worst case: 2 marks

It takes at most 2 operations to get the largest coin to the bottom, at most 2 more for the next one, and so on. Hence at most 2n. Actually, for the last coin, it takes no operation. So $2n - 2$ operations, at most.

Slight variation to the worst case: The Algorithm can be optimised to

If the size of the pile is 1, we’re done.

If the size of the pile is 2, sort the two coins using at most 1 operation.

Else, bring coin n to the top, then bring it to the bottom.

Repeat this algorithm on the top $n - 1$ coins.

In this case we will require one 1 operation for n=2, or for n coins it requires 2n -3 operations at most.

b)

Observation:
Suppose that the pile contains a pair of adjacent coins that will not be adjacent in the sorted pile. Then any sequence of flips that sorts S must involve at least one flip that performs one operation between the two members of this pair (and breaks them apart). Furthermore, the same principle holds for the ”pair” formed by the bottom coin of S, and the surface.

So consider any stack where each consecutive pair must be separated, e.g. (5 2 4 1 3), or (3 1 5 2 4) or (2 4 1 5 3). The above observation shows that we’ll need a minimum of 5 flips in order to sort each such stack.

In fact, for any even $n > 2$, consider the lower bound stack $(2\ 4\ 6\ 8 \cdots n\ 1\ 3\ 5 \cdots (n - 1)$) Each consecutive pair needs to be separated, so this shows that Flips_n ≥ n for n = 4, 6, 8, ...

For odd n >3, consider the pile (2 4 8 … (n-1) 1 n 3 … (n-2)) again this requires ≥ n operations to sort.
Binary Marking: 2.5+ 2.5 points if satisfactory and correct justification of an arrangement of coins is present for all values of n (both odd and even).

c)

Algorithm:

- If the size of the pile is 1,
  Check if the head is up: if not then flip the topmost coin.
- Else, bring coin n to the top,
  - check if the tail side is up, if not then perform operation on the topmost coin
  - Then bring it to the bottom.
- Repeat this algorithm on the top n − 1 coins.

Algorithm → 5 marks →
mentioning subproblems -> 1 mark
Describing how to solve given problem using subproblem -> 2 marks
Using defined operation to solve the problem-> 1 mark -> binary
Answer to the original problem: 1 marks

Recurrence Relation:

T(n) = Number of operations for array of size 'n'
$T(n) <= T(n-1) + 3$ 1 mark
$T(1) <= 1$; 1 mark

On solving using Master's theorem we get
$T(n) = O(n)$ 1 mark

Number of operations in the worst case: 2 marks

It takes at most 3 operations to get the largest coin to the bottom such that heads are facing up, at most 3 more for the next one, and so on. Hence at most 3n. Actually, for the last coin, it takes at most one operation. So 3n − 2 operations, at most.

Q4: Nth Magical Number:

Akshita loves chocolates, numbers and algorithms. Her friends Ina and Mina gave her a and b chocolates respectively. Given her love for numbers, she wants to find the nth smallest number that is divisible by either a or b. Complexity $\rightarrow O(\log(N*\min(A,B)))$.
Assume a and b to be co-prime positive integers.

Marking Scheme: Total 15 points.
Algorithm: 5 points.
Recurrence relation and answer to original problem: 3+2 points
Pseudo Code: 5 points

Solution:

Algorithm:

- We will binary search on the value of the answer. We will search from 1 to min(a, b)*n, and find the minimum number k such that there are atleast n numbers divisible by either a or b or both.
- On every recursive call we reduce our search space to about half till there is only one element left.
- We return that element.

Algorithm: 5 marks
Using Binary search, dividing the problem efficiently -> 2 marks
Writing condition to eliminate n/2 out of n. -> 2 marks
Getting answers to the problem using subproblems.-> 1 mark

Recurrence Relation:
T(n) = T(n/2) + C 2 marks
T(n) = O(logn) 1 mark
Here n is the size of the range in which we have to look for our answer.
Answer to the original question: find(a, b, n) 2 marks

Pseudo Code:

```
Boolean check(a, b  k,  n)
        If ( (k/a) + (k/b) - (k/(a*b)) >= n)
                Return true
        end if
        Return false
```

```
End check
```

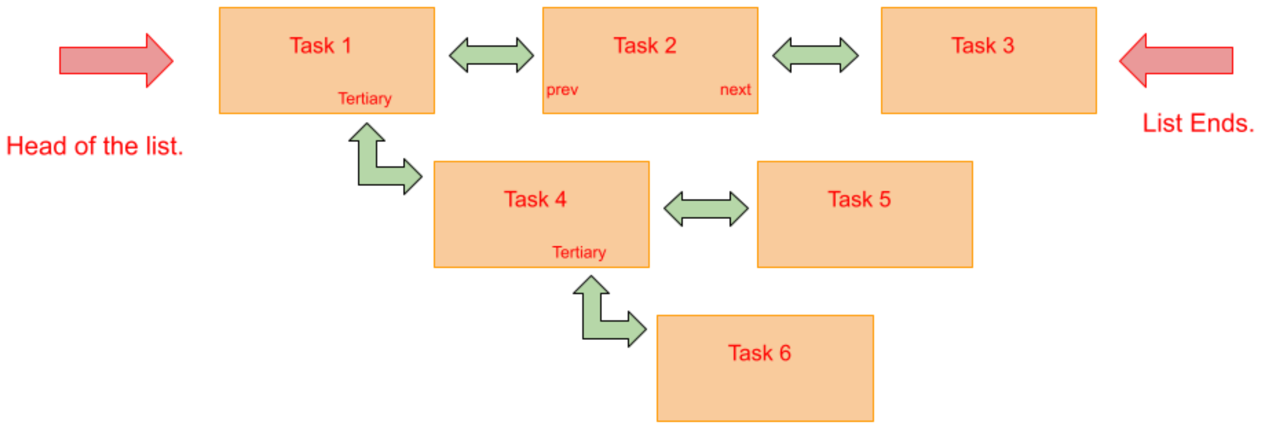
```
Int find (a, b, n )
        Start = 1
        End = n* (min(a,b))
        While (start < end)
                Mid = (start+end)/2
                if(check(a, b, mid, n) == false)
                        Start = mid+1
                Else
                        End = mid
                End if
        End while
        Return start
End find
```

PSEUDO CODE → 5 marks →
mentioning the end case -> 1 mark
Using answer of subproblem to get answer for the main problem -> 2 marks
Writing the checking condition correctly (check function) -> 2 marks -> binary

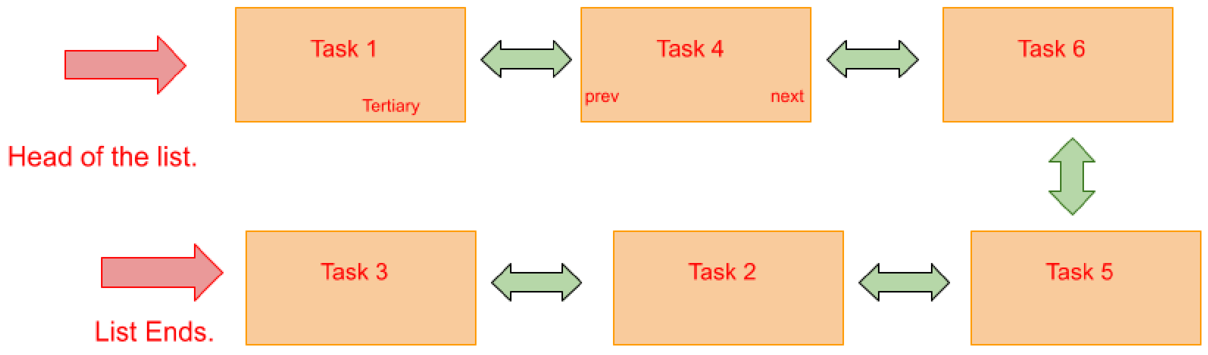
BONUS Question (10 points)

In a hypothetical server, we have a list of tasks arranged in the form of a doubly linked list. In this list in addition to the next and previous pointer each node also has a tertiary pointer, which represents some cleanup tasks required before moving on to the next tasks of the list. The current scheduler however does not recognize this list structure. Your task is to implement an algorithm that takes input the head of the doubly linked list with nodes having a tertiary pointer and gives the head of a rearranged doubly linked list with all tasks arranged linearly (with nodes having just the next and previous pointers only, no tertiary pointer). The modifications should be in-place, that means you cannot change the contents (other than next and previous pointer) or the address of any of the nodes. Also Analyse the running time of the algorithm for the given 'n' being the number of tasks.

Example: Let the given input list be:



The output list that your algorithm should give:


Task 1
Tertiary
Task 4
prev
next
Task 6
Head of the list.
Task 3
Task 2
Task 5
List Ends.

```
class Solution {
    public Node flatten(Node head) {


if (head == null) return null;
        getflatten(head);


        head.prev = null;

        return head;
    }
    public Node getflatten(Node t){
        Node k = t;
        while (k.next != null || k.child != null){
            if (!(k.child == null)){
                Node end = getflatten(k.child);
                Node ref = k.next;
                k.next = k.child;
                k.next.prev = k;


                end.next = ref;
         if (ref != null)       ref.prev = end;
                k.child = null;
                k = end;


            }
        else k = k.next;
        }


        return k;
    }
}
```

Pseudo Code: 5 marks (binary)

Logic: Recursion is to be used. So the base case is: when there is only one node with child pointer null then return the node.
If the child pointer is not null, then we flatten the doubly linked list attached the the child pointer (this is our subproblem), and then insert the flattened list between the current node, and the next node. → (3 marks -> 1 mark for base case, 1 for subproblem, 1 for mentioning how to use the subproblem)
Complexity: O(n) where n is the number of nodes   as we only need to traverse every node once(2 marks -> 1 mark for time complexity and 1 mark for explanation)