*“If people are doubting how far you can go, go so far that you can’t hear them anymore.” – Michele Ruiz*

Dear reader, I hope that you are not bored yet with linked list data structure. I welcome you to the new problem: ‘[***Merge Sort A Linked List***](https://www.pepcoding.com/resources/online-java-foundation/linked-lists/mergesort-linkedlist-official/ojquestion)***’***. It is a problem in continuation of the previous problems: ‘[***Mid of Linked List’***](https://www.pepcoding.com/resources/online-java-foundation/linked-lists/mid-linked-list-official/ojquestion) and ‘[***Merge Two Sorted Linked Lists***](https://www.pepcoding.com/resources/online-java-foundation/linked-lists/merge-two-sorted-linked-lists-official/ojquestion)’.

***Problem Statement***

* You are given a partially written LinkedList class.
* You are required to complete the body of mergeSort function. The function is static and is passed the head and tail of an unsorted list. (Input and Output is managed for you.)
* The function is expected to return ***a new sorted list***. The original list must not change.

To understand the theory concept behind merge sort algorithm and the problem statement in detail, I recommend you to watch the [question video](https://www.youtube.com/watch?v=DEvWhHgBjL0) associated with this problem.

***Deducing Algorithm***

We know how to merge-sort an unsorted array. How can we use the concept we learnt there to solve this problem?

As we know that merge sort is a ***Divide & Conquer Algorithm***, we can solve a larger subproblem by dividing the problem into subproblems recursively, and combining the solved smaller problem to solve the larger problem.

**Q)** How to divide the problem into subproblems?

* Similar to what we did in arrays, we will sort the first half of the linked list and the second half of the linked list separately.
* But, in arrays, getting the middle index to split the left & right ranges is easy, as we can randomly access any index in O(1) constant time.
* In the linked list, to get the middle node of the current range, we will have to do a traversal of the current range, as we cannot get direct access to the middle node.
* We know how to get the middle node of a linked list in one traversal using the ***concept of slow and fast pointers***, right?
* I am not going to explain how to find the middle node of the linked list in this article again. We will use the function *mid()* which we wrote earlier directly.

So, now we know how to find smaller subproblems. We will recursively call for the smaller subproblem, i.e. call mergeSort to sort the left part (from head to mid) and also call mergeSort to sort the right part (from mid->next to tail). On calling mergeSort recursively, we will get two new sorted lists.

**Q)** How to combine the results of smaller subproblems to solve the current larger problem, i.e. sort linked list from head to tail?

* We have two linked lists which are in sorted order separately. Now, we need to merge these sorted linked lists into a single linked list, which should be sorted as well.
* We know how to merge two sorted linked lists in-place, right? I am not going to explain how to write the merge operation, which we wrote earlier.

**Q)** What should be the ***base case*** of the recursive merge sort function?

When we have only one node in the range, then this node is already sorted. Hence, we will create a new node with the same value and return this new node.

We are creating a new node, because we are required to make a new sorted list, and do not modify the original one.

***Pseudo Code/ Algorithm***

* If there is only one node in the linked list, (check ***if head == tail*** or not), then ***make a new node*** with data = head->val and ***return this new node***.
* Else,
  + Get the middle node of the linked list range [head, tail] as node *mid*, using mid() function.
  + Recursively call for left and right subproblems as:
    - ***Fsh = mergeSort(head, mid)***
    - ***Ssh = mergeSort(mid.next, tail)***
  + Now, merge two sorted linked list using merge() operation and get the sorted linked list from [head, tail] as
    - ***sl = mergeTwoSortedLists(fsh, ssh)***
    - And return this linked list’s head sl.

Please refer to the [solution video](https://www.youtube.com/watch?v=ClLO6kFF6gs) if you find difficulty in understanding the algorithm completely.

***Implementation (Java)***

Please I request you to give it a try before reading the code! Don’t just blindly copy the code and submit it on any portal for the sake of completion. Doing so will not help you build an insight about Data Structures & Algorithms.

import java.io.\*;

import java.util.\*;

public class Main {

public static class Node {

int data;

Node next;

}

public static class LinkedList {

Node head;

Node tail;

int size;

void addLast(int val) {

Node temp = new Node();

temp.data = val;

temp.next = null;

if (size == 0) {

head = tail = temp;

} else {

tail.next = temp;

tail = temp;

}

size++;

}

public int size() {

return size;

}

public void display() {

for (Node temp = head; temp != null; temp = temp.next) {

System.out.print(temp.data + " ");

}

System.out.println();

}

public void removeFirst() {

if (size == 0) {

System.out.println("List is empty");

} else if (size == 1) {

head = tail = null;

size = 0;

} else {

head = head.next;

size--;

}

}

public int getFirst() {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else {

return head.data;

}

}

public int getLast() {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else {

return tail.data;

}

}

public int getAt(int idx) {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else if (idx < 0 || idx >= size) {

System.out.println("Invalid arguments");

return -1;

} else {

Node temp = head;

for (int i = 0; i < idx; i++) {

temp = temp.next;

}

return temp.data;

}

}

public void addFirst(int val) {

Node temp = new Node();

temp.data = val;

temp.next = head;

head = temp;

if (size == 0) {

tail = temp;

}

size++;

}

public void addAt(int idx, int val) {

if (idx < 0 || idx > size) {

System.out.println("Invalid arguments");

} else if (idx == 0) {

addFirst(val);

} else if (idx == size) {

addLast(val);

} else {

Node node = new Node();

node.data = val;

Node temp = head;

for (int i = 0; i < idx - 1; i++) {

temp = temp.next;

}

node.next = temp.next;

temp.next = node;

size++;

}

}

public void removeLast() {

if (size == 0) {

System.out.println("List is empty");

} else if (size == 1) {

head = tail = null;

size = 0;

} else {

Node temp = head;

for (int i = 0; i < size - 2; i++) {

temp = temp.next;

}

tail = temp;

tail.next = null;

size--;

}

}

public void removeAt(int idx) {

if (idx < 0 || idx >= size) {

System.out.println("Invalid arguments");

} else if (idx == 0) {

removeFirst();

} else if (idx == size - 1) {

removeLast();

} else {

Node temp = head;

for (int i = 0; i < idx - 1; i++) {

temp = temp.next;

}

temp.next = temp.next.next;

size--;

}

}

private Node getNodeAt(int idx) {

Node temp = head;

for (int i = 0; i < idx; i++) {

temp = temp.next;

}

return temp;

}

public void reverseDI() {

int li = 0;

int ri = size - 1;

while (li < ri) {

Node left = getNodeAt(li);

Node right = getNodeAt(ri);

int temp = left.data;

left.data = right.data;

right.data = temp;

li++;

ri--;

}

}

public void reversePI() {

if (size <= 1) {

return;

}

Node prev = null;

Node curr = head;

while (curr != null) {

Node next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

}

Node temp = head;

head = tail;

tail = temp;

}

public int kthFromLast(int k) {

Node slow = head;

Node fast = head;

for (int i = 0; i < k; i++) {

fast = fast.next;

}

while (fast != tail) {

slow = slow.next;

fast = fast.next;

}

return slow.data;

}

public int mid() {

Node f = head;

Node s = head;

while (f.next != null && f.next.next != null) {

f = f.next.next;

s = s.next;

}

return s.data;

}

public static LinkedList mergeTwoSortedLists(LinkedList l1, LinkedList l2) {

LinkedList ml = new LinkedList();

Node one = l1.head;

Node two = l2.head;

while (one != null && two != null) {

if (one.data < two.data) {

ml.addLast(one.data);

one = one.next;

} else {

ml.addLast(two.data);

two = two.next;

}

}

while (one != null) {

ml.addLast(one.data);

one = one.next;

}

while (two != null) {

ml.addLast(two.data);

two = two.next;

}

return ml;

}

public static Node midNode(Node head, Node tail){

Node f = head;

Node s = head;

while(f != tail && f.next != tail){

f = f.next.next;

s = s.next;

}

return s;

}

public static LinkedList mergeSort(Node head, Node tail){

if(head == tail){

LinkedList br = new LinkedList();

br.addLast(head.data);

return br;

}

Node mid = midNode(head, tail);

LinkedList fsh = mergeSort(head, mid);

LinkedList ssh = mergeSort(mid.next, tail);

LinkedList sl = mergeTwoSortedLists(fsh, ssh);

return sl;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int n1 = Integer.parseInt(br.readLine());

LinkedList l1 = new LinkedList();

String[] values1 = br.readLine().split(" ");

for (int i = 0; i < n1; i++) {

int d = Integer.parseInt(values1[i]);

l1.addLast(d);

}

LinkedList sorted = LinkedList.mergeSort(l1.head, l1.tail);

sorted.display();

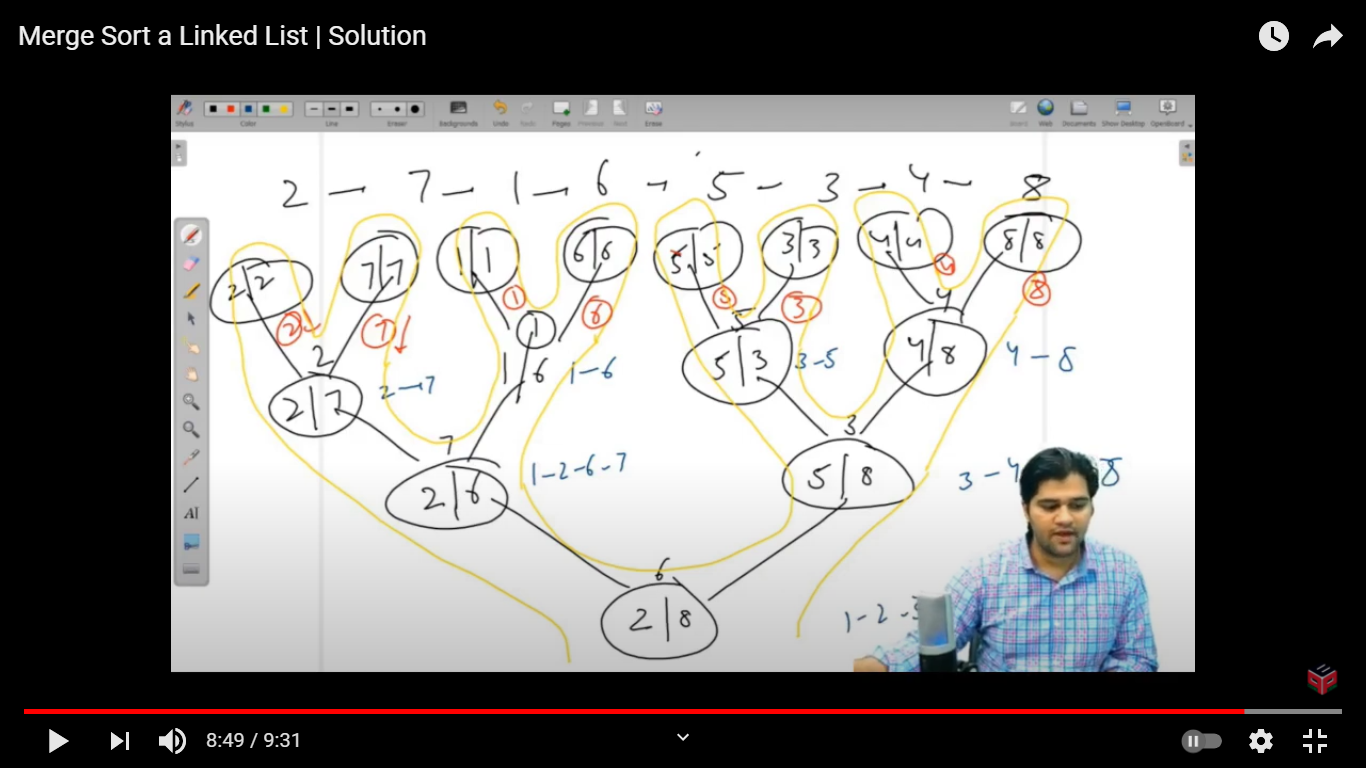
l1.display();

}

}

This code is written and explained by our team in [this video](https://www.youtube.com/watch?v=ClLO6kFF6gs) from *[1:00, 7:53]*. Please refer to it if you are stuck somewhere.

You should perform a **dry run** of the algorithm on some examples, draw the recursion tree, to get a better understanding. It is also explained in the [same video](https://www.youtube.com/watch?v=aMPXhEdpXFA) from *[7:53, 9:15]*.



***Time & Space Complexity Analysis***

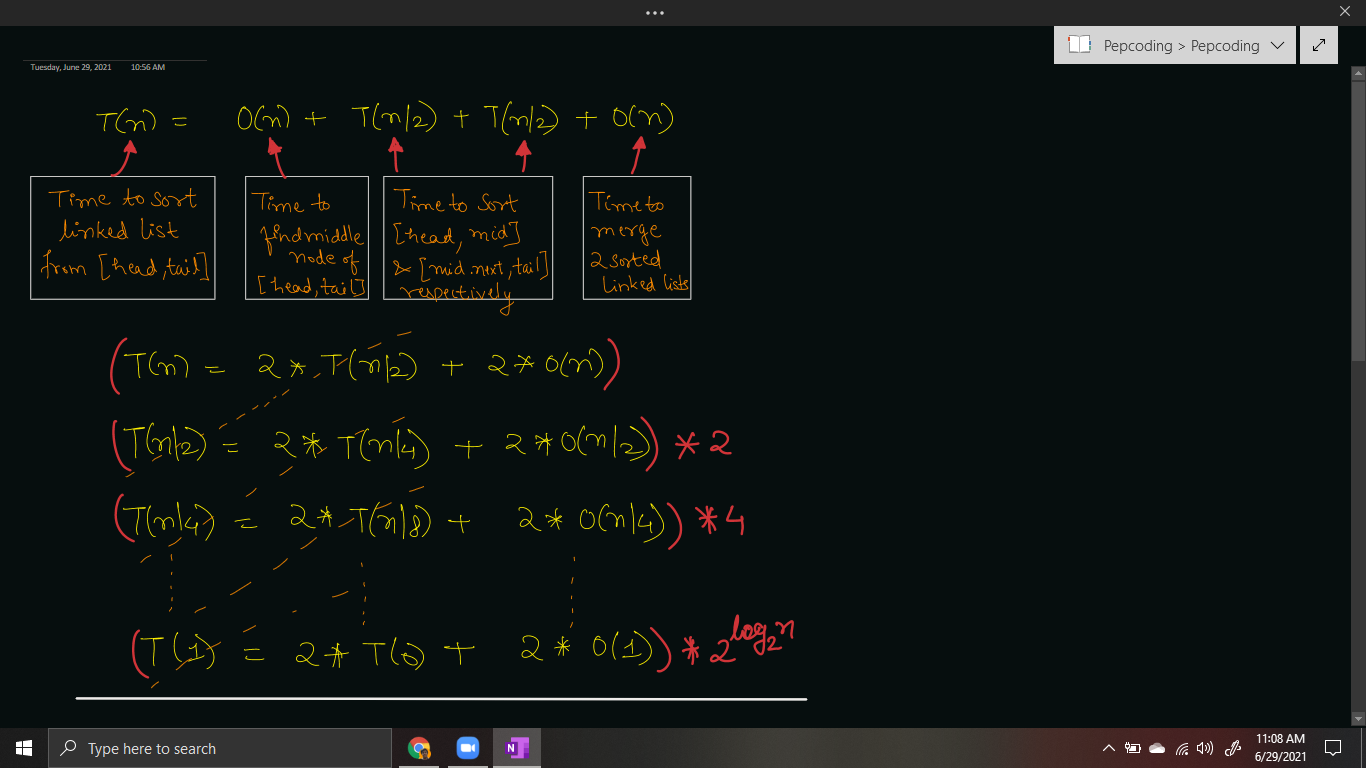
**Time Complexity** -

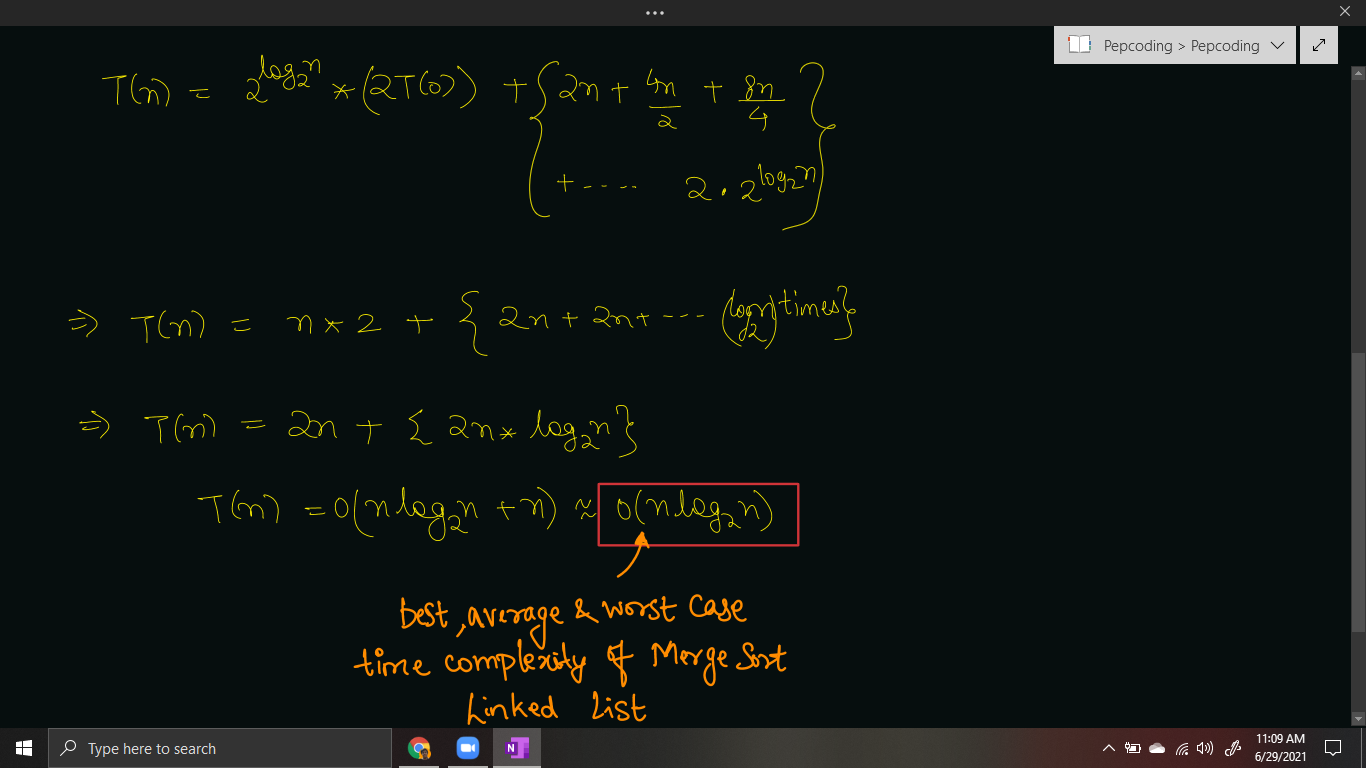
At each recursive step [head, tail], we are calling the mid() function to get the middle node. Getting the middle node takes time as O(nodes in linked list).

After getting the middle node, we are recursively calling mergeSort(head, mid) and mergeSort(mid.next, tail).

Then after getting the two sorted linked lists, we are merging them, which again takes O(nodes in the first linked list + nodes in second linked list) = O(n1 + n2) = O(n).

Let us write the recursive relation in terms of the time complexities:





**Space Complexity** -

As we are creating a new linked list of size equal to the original linked list, we are using an auxiliary space of ***O(n)*** for the new linked list.

Also, we are calling the recursive function for at maximum log2 n time, hence the recursive call stack will take ***O(log2 n)*** space.

***Follow Up***: In this problem, we were required to create a new linked list and not modify the original one. We can also solve this problem by modifying the original linked list only and ***sorting in-place***. We will see how to [***merge-sort the linked list***](https://www.pepcoding.com/resources/data-structures-and-algorithms-in-java-levelup/linked-list/mergesort-linkedlist/ojquestion) by taking only O(log2 n) recursion call stack space in the level-2 section of this course.

I hope you enjoyed solving the problem with me. I will bring an easy problem for you to solve, which is named ‘[Remove Duplicates in Sorted Linked List](https://www.pepcoding.com/resources/online-java-foundation/linked-lists/remove-duplicates-official/ojquestion)’. *Happy Coding*!

Contributor : [Archit Aggarwal](https://www.linkedin.com/in/archit-aggarwal-6a7716189/)