Step 7 : Theoretical analysis of time complexity of linked list program (inference from google and chat gpt)

Breaking down the program into different parts and assessing the time complexity of each part.
1. Generating Random Numbers and appending to the Initial Linked List:
  - Generating `no_terms` random integers in the range [1, `end_range`] and inserting them into the linked list takes O(no_terms) time.
2. Sorting the Linked List:
  - The sorting of the linked list is done using merge sort. Merge sort has a time complexity of O(n log n) for the average and worst cases, where 'n' is the number of elements in the list.
3.Deleting an Element at a Specific Position:
  - The `delete_ele` method has a time complexity of O(pos), where 'pos' is the position of the element to be deleted. In the worst case, this operation could take O(n) time if you delete the last element.
4. Inserting an Element in Sorted Order:
  - Both `add_to_asc` and `add_to_dsc` methods involve inserting an element into its correct sorted position in the linked list. The time complexity for these operations is O(n) in the worst case because you might need to traverse the entire list to find the correct position.
5. Printing the Linked List:
  - Printing the linked list using the `__repr__` method has a time complexity of O(n) because it iterates through all the elements in the list.
6. Insert Operations:
  - Input operations like reading `no_terms` and `end_range` have a time complexity of O(1).
7. Counting elements greater than 50:
  - Operations such as counting elements greater than 50 & conditional checks are generally O(n).
8. Other constant operations. several operations that can be considered constant time (O(1)).
* LinkedList `add_first` Method: Adding an element to the head of a linked list is a constant-time operation because it involves updating a few references, but the number of operations does not depend on the size of the list.
* LinkedList `add_to_dsc` Method: Adding an element to a doubly linked list in descending order is also a constant-time operation because it involves traversing the list until the correct position is found and then updating references.
* LinkedList `add_to_asc` Method: Adding an element to a doubly linked list in ascending order is another constant-time operation for the same reason as `add_to_dsc`.
* LinkedList `delete_ele` Method: Deleting an element from a doubly linked list by specifying its position is a constant-time operation because it involves updating references to remove the element.
* Adding 10 to the Linked List: adding the value 10 to the linked list is a constant-time operation. This is because you are adding a single element, and the time it takes does not depend on the size of the list.
* Printing the Linked List: Printing the linked list using the `print` statements is generally considered a constant-time operation because the time it takes to display the elements is not dependent on the number of elements in the list.
* Measuring Execution Time: Measuring the execution time using `time.time()` before and after the code execution is a constant-time operation because it captures the start and end times, which are independent of the input size.

The program can be expressed as function : f(n)
f(n)= n+n log n+ n + n + n + 1 + n+ 1 + 1 +1+1 +1+1+1

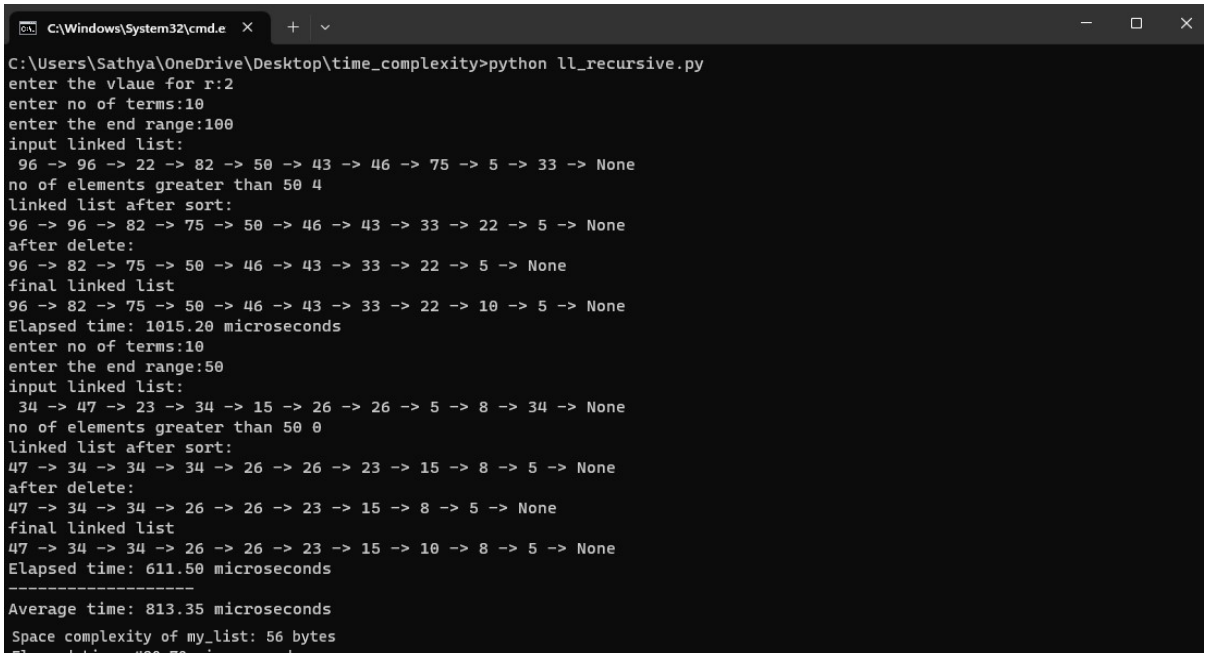
f(n)=5n+n log n + 8
f(n)=5n+n log n (omitting the constant time)
*n log n dominates over constant 4n so asymptotic f(n) can be expressed as
f(n)=0(n log(n))

To put it all together:
- The most significant time complexity-contributing operation is the sorting step O(n log n).
Therefore, the overall time complexity of the code can be approximated as O(n log n).

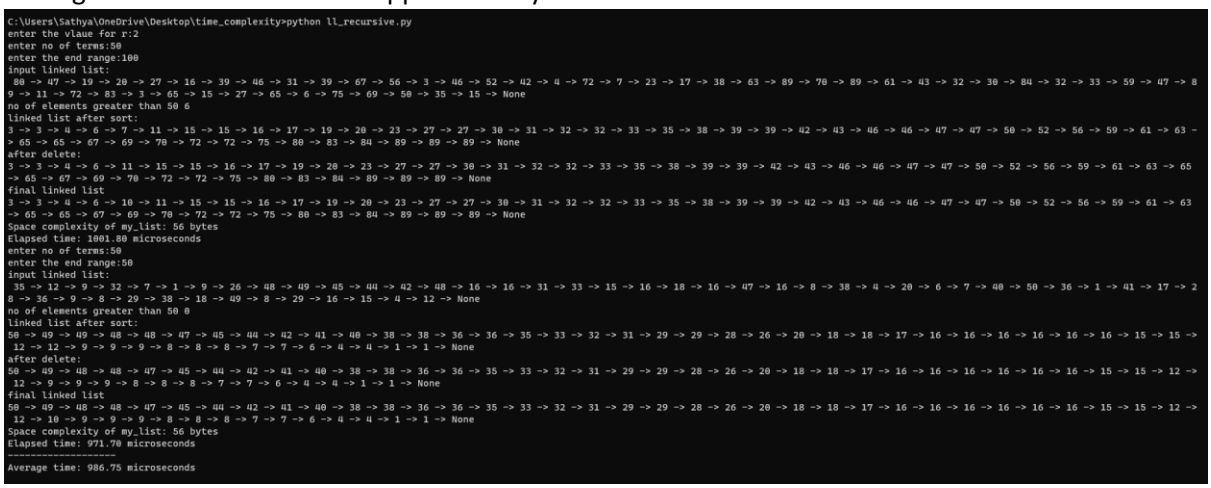
Analyzing the space complexity of linked list program:
The space complexity of the given code can be broken down into several components:
1. Linked List Nodes:
  - The code creates a linked list to store the random integers. Each integer is stored in a `Node` object, which contains data, a reference to the next node, and a reference to the previous node.
  - The space required for the linked list nodes is O(no_terms) because there are 'no_terms' nodes created.
2. Additional Linked Lists:
  - The code creates additional linked lists (`final_linked_list`) to perform sorting and other operations.
  - The space required for these additional linked lists is also O(no_terms) because they can have a similar number of nodes as the original linked list.
3. Function Call Stack:
  - The code uses recursive functions like `merge_sort_linked_list`, which can lead to a function call stack.
  - The maximum depth of the function call stack in merge sort is O(log(no_terms)).
4. Temporary Variables:
  - The code uses temporary variables within functions for various operations.
  - The space used by these temporary variables is generally O(1) and does not depend on the size of the input.
5. Input Variables:
  - The code stores input variables such as `no_terms` and `end_range`.
  - The space required for these input variables is O(1) because they are single values.

To summarize the space complexity:
- The dominant factor in the space complexity is the linked list nodes, which require O(no_terms) space.

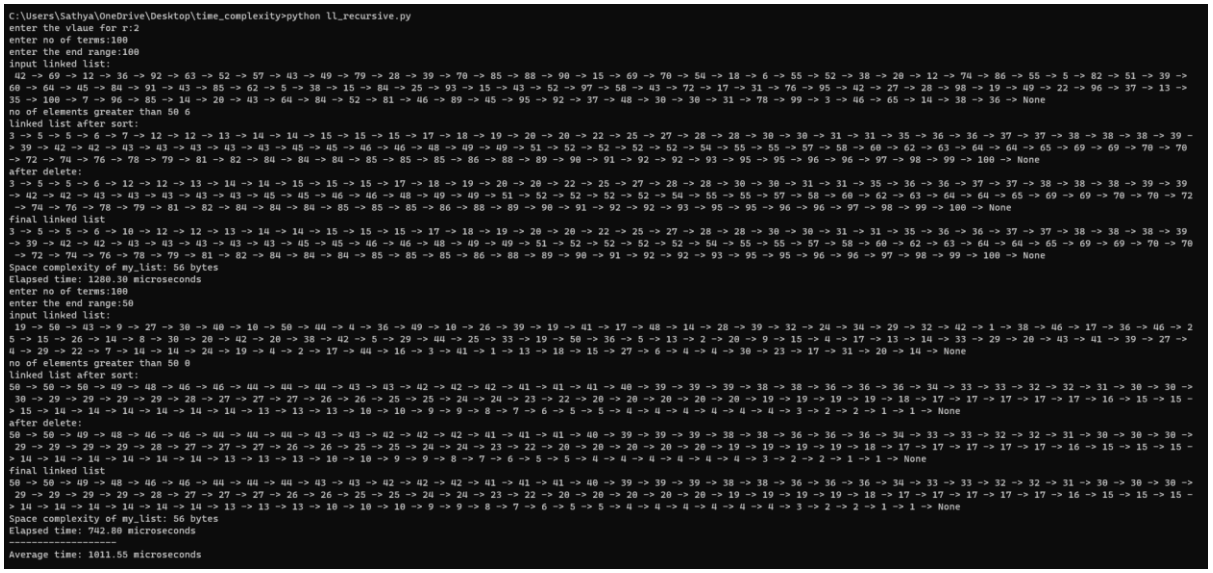
Step 8 - Empirical analysis of time and space complexity.
Average time taken for n=10 is approximately 1001.1 micro sec



Average time taken for n=50 is approximately 986.75 micro sec



Average time taken for n=100 is approximately
1011.55 micro sec



Step 9: Recursive testing for higher order of N for the comparison between theoretical and empirical measurements.

```
C:\Users\Sathya\OneDrive\Desktop\time_complexity>python ll_recursive.py
enter the vlaue for r:4
enter the end range for m:300
value of n=500
Elapsed time: 3158.20 microseconds
Elapsed time: 2832.40 microseconds
Elapsed time: 2982.60 microseconds
Elapsed time: 4140.90 microseconds
--------------------
Average time: 3278.52 microseconds
```

```
C:\Users\Sathya\OneDrive\Desktop\time_complexity>python ll_recursive.py
enter the vlaue for r:4
enter the end range for m:800
value of n=1000
Elapsed time: 4275.90 microseconds
Elapsed time: 4439.30 microseconds
Elapsed time: 3652.40 microseconds
Elapsed time: 3653.40 microseconds
--------------------
Average time: 4005.25 microseconds
```

```
C:\Users\Sathya\OneDrive\Desktop\time_complexity>python ll_recursive.py
enter the vlaue for r:4
enter the end range for m:3000
value of n=5000
Elapsed time: 21124.20 microseconds
Elapsed time: 17594.60 microseconds
Elapsed time: 17513.80 microseconds
Elapsed time: 16239.30 microseconds
--------------------
Average time: 18117.98 microseconds
```

```
C:\Users\Sathya\OneDrive\Desktop\time_complexity>python ll_recursive.py
enter the vlaue for r:4
enter the end range for m:8000
value of n=10000
Elapsed time: 44273.60 microseconds
Elapsed time: 38337.70 microseconds
Elapsed time: 36557.50 microseconds
Elapsed time: 34047.50 microseconds
--------------------
Average time: 38304.08 microseconds
```

```
C:\Users\Sathya\OneDrive\Desktop\time_complexity>python ll_recursive.py
enter the vlaue for r:4
enter the end range for m:8000
value of n=50000
Elapsed time: 219985.20 microseconds
Elapsed time: 228807.90 microseconds
Elapsed time: 219750.90 microseconds
Elapsed time: 251558.70 microseconds
--------------------
Average time: 230025.68 microseconds
```

```
enter the vlaue for r:4
enter the end range for m:8000
value of n=100000
Elapsed time: 89114.16 microseconds
Elapsed time: 83768.00 microseconds
Elapsed time: 74881.22 microseconds
Elapsed time: 83488.49 microseconds
--------------------
Average time: 82812.97 microseconds
```

```
enter the vlaue for r:4
enter the end range for m:8000
value of n=1000000
Elapsed time: 11836066.60 microseconds
Elapsed time: 12395266.35 microseconds
Elapsed time: 12133055.60 microseconds
Elapsed time: 12305493.19 microseconds
--------------------
Average time: 12167470.44 microseconds
```