1.a) The code defines a function addition_ finder that takes the length of the list - "N", the target, sum "S", and the list of integers 'arr' as inputs. It iterates through each pair of distinct positions in the list using nested loops. If the sum of the numbers of the current pair of positions is equal to the target sum, it returns the positions. If no such pair is found, it returns "~~Impossible~~". "IMPOSSIBLE".

b) The code keeps track of the numbers, it has come across so far using a dictionary 'dic'. It cycles through the list of numbers. Computing the complement for each number (s less the current number). The presence of the complement in the dictionary indicates that we have discovered two integers that add up to S. The complement location and the current number are then returned. We return 'Impossible', if the loop end up being intractable. Because we only iterate through the list of integers once, the time complexity of these solution is O(N).

2a) This program defines a function called 'for' that was the merge sort algorithm to combine two sorted lists into a single sorted list. The items are compared when we iterate through both lists at once, adding the smaller element to the combined list. The remaining components for each list are then added to the combined list. The remaining components for each list are then added to the combined list. The merge sort algorithm makes sure that the final sorted list is produced in $O(n\log n)$ time where n is the sum of the entries in both lists.

2.b) It iterates accross both lists currently, comparing elements and adding them in asending order to the combined list. It adds the remaining components of the other list to the merged list after using up all the elements in one of the the lists.

The output file is then written with the final merged list. This algorithm has an O(n) time complexity, where n is the sum of the elements in the two input list.

3. The 'merge sort' function separates the input list recursively until each sublist includes just one element ~~each~~ then merges them back together in sorted order. Then ~~'merge'~~ 'merge' function joins two sorted lists together.

4. This algorithm time complexing, where N is the lists length, is O(NlogN). This is due to the lists repeated division into halves and comparison at each level of the higest value from each half. The recursion has logN levels. and since we perform a fixed amount of work at each level, the overall time complexity is O(NlogN).