1. ***Explain Divide and Conquer algorithm taking reference to Merge Sort.***

***Here's how divide and conquer rule is used:***

***Divide- The elements are sorted into two equal halves. To avoid copying portions of the array, typically the portion of the array being sorted is marked using a start index p and an end index r. The portion of the array being considered is a[p],...,a[r]. Sofortheinitialproblem p = 0and r = n−1. Let q =b(p+r)/2c. Thentheproblemofsorting a[p],...,a[r] is divided into two sub problems: sort a[p],...,a[q] and sort a[q +1],...,a[r].***

***Conquer-If either sub array to sort has size 1 then just return. Otherwise, recursively sort the sub array. This step receives lot of smaller sub problem to be solved. Generally at this level, problems are considered ‘solved’ on their own.***

***Combine-Merge the two sorted sub arrays. An important component of this algorithm is that merging two sorted sub arrays is asymptotically more efﬁcient than sorting the original array. In particular, there is a simple linear time algorithm to merge two sorted arrays. We just cover the intuition here. A supplemental array is used to hold the sorted array, which can then be moved back into a[p],...,a[r]. An index of the current location (initially p) for the left half is maintained, an index of the current location (initially q + 1) of the right half is maintained, and an index of the current location into the supplemental array is maintained. Then the elements at the current locations of the left and right halves are compared. The smaller of these elements (or either if equal) are moved into the next open location in the supplemental array. Then the index of the array from which the element was moved is incremented and the index into the supplemental array is implemented. Since constant time is used at each step, and there is one step for each of the elements in the portion of a being sorted, this merge algorithm has linear worst-case asymptotic time complexity.***

***For any given problem there can be more than one divide-and-conquer algorithm. Later in this course, we will study the quicksort algorithm which is an alternate divide-and-conquer sorting algorithm that spends linear time dividing the array into sub arrays so that no computation is needed to combine.***

***A Divide and Conquer Algorithm to sort an array:***

***voidmergesort(S: array of keytype)***

***len = S'length***

***iflen> 1 then***

***-- Divide: Copy the arrays***

***mid: constant int := len / 2***

***rest: intlen - mid***

***U: array(1..mid) of keytype := S(1..mid)***

***V: array(1..rest) of keytype := S(mid+1..n)***

***-- Conquire: Recursively sort***

***mergesort(U)***

***mergesort(V)***

***-- Combine: merge sorted arrays***

***merge(U, V, S)***

***endmergesort***

***void merge(L, R, S: array(<>) of keytype)***

***i, j, k: int := 1***

***lenL: constant int := L'length***

***lenR: constant int := R'length***

***while i <= lenL and j <= lenR***

***if L(i) < R(j) then***

***S(k) := L(i); i := i + 1;***

***else -- R(i) ≤ L(j) then***

***S(k) := R(j); j := j + 1;***

***k := k + 1***

***if i >lenL then S(k..lenL+lenR) := R(j..lenR)***

***else S(k..lenL+lenR) := L(i..lenL)***

1. ***Write some paragraphs about External Sorting with some examples.***

***External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.***

***One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into runs such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.***

1. ***Explain Partition strategies of Merge Sort and Quick Sort.***

***Merging sort is the process of combining two or more sorted files into a third sorted file. An example of a routine that accepts two sorted arrays a and b of n1 and n2 elements respectively and merges them into a third array C containing n3 elements. Most implementation merge sort produce a stable sort which means that the implementation preserves the input order of equal elements in the sorted output.***

***Code for merge:***

***defmergeSort(alist):***

***print("Splitting ",alist)***

***iflen(alist)>1:***

***mid = len(alist)//2***

***lefthalf = alist[:mid]***

***righthalf = alist[mid:]***

***mergeSort(lefthalf)***

***mergeSort(righthalf)***

***i=0***

***j=0***

***k=0***

***while i <len(lefthalf) and j <len(righthalf):***

***iflefthalf[i] <righthalf[j]:***

***alist[k]=lefthalf[i]***

***i=i+1***

***else:***

***alist[k]=righthalf[j]***

***j=j+1***

***k=k+1***

***while i <len(lefthalf):***

***alist[k]=lefthalf[i]***

***i=i+1***

***k=k+1***

***while j <len(righthalf):***

***alist[k]=righthalf[j]***

***j=j+1***

***k=k+1***

***print("Merging ",alist)***

***alist = [54,26,93,17,77,31,44,55,20]***

***mergeSort(alist)***

***print(alist)***

***Quick sort:***

***Quick sort is an efficient sorting algorithm serving as a systematic method for placing the elements of an array in order. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heap sort. Quick sort is a comparison sort, meaning that it can sort items of any type for which a less than relation is defined. In efficient implementations it is not a stable sort, meaning that the relative order of equal sort item is not preserved. Quick sort can operate in-place on an array, requiring small amount of memory to prefer the sorting. The algorithm takes O(nlogn) comparisons to sort n items. In worst case, it makes O(n2) comparisons.***

***Code for quick sort:***

***defquickSort(alist):***

***quickSortHelper(alist,0,len(alist)-1)***

***defquickSortHelper(alist,first,last):***

***if first<last:***

***splitpoint = partition(alist,first,last)***

***quickSortHelper(alist,first,splitpoint-1)***

***quickSortHelper(alist,splitpoint+1,last)***

***def partition(alist,first,last):***

***pivotvalue = alist[first]***

***leftmark = first+1***

***rightmark = last***

***done = False***

***while not done:***

***whileleftmark<= rightmark and alist[leftmark] <= pivotvalue:***

***leftmark = leftmark + 1***

***whilealist[rightmark] >= pivotvalue and rightmark>= leftmark:***

***rightmark = rightmark -1***

***ifrightmark<leftmark:***

***done = True***

***else:***

***temp = alist[leftmark]***

***alist[leftmark] = alist[rightmark]***

***alist[rightmark] = temp***

***temp = alist[first]***

***alist[first] = alist[rightmark]***

***alist[rightmark] = temp***

***returnrightmark***

***alist = [54,26,93,17,77,31,44,55,20]***

***quickSort(alist)***

***print(alist)***

1. ***Why is merge sort preferred over quick sort for sorting linked lists?***

***Merge sort is very efficient for immutable data structures like linked lists.Quick sort is typically faster than merge sort when the data is stored in memory. However, when the data set is huge and is stored on external devices such as a hard drive, merge sort is the clear winner in terms of speed. It minimizes the expensive reads of the external drive and when operating on linked lists, merge sort only requires a small constant amount of auxiliary storage***

1. ***Write down your analysis on Complexities of each algorithm we've discussed in the class.***
2. ***Bubble sort:***

***• For an array of size n, in the worst case: 1st passage through the inner loop: n-1 comparisons and n-1 swaps***

***• (n-1)st passage through the inner loop: one comparison and one swap.***

***• All together: c ((n-1) + (n-2) + ... + 1), where c is the time required to do one comparison, one swap, check the inner loop condition and increment j.***

***• We also spend constant time k declaring i,j,temp and initialising i. Outer loop is executed n-1 times, suppose the cost of checking the loop condition and decrementing i.***

1. ***Insertion Sort:***

***• In the worst case, has to make n(n-1)/2 comparisons and shifts to the right***

***• also O(n2) worst case complexity***

***• best case: array already sorted, no shifts.***

1. ***Selection Sort:***

***• Same number of iterations***

***• Same number of comparisons in the worst case***

***• fewer swaps (one for each outer loop = n-1)***

***• also O(n2)***

1. ***Merge Sort:***

* ***The time to merge sort n numbers is equal to the time to do two recursive merge sorts of size n/2 plus the time to merge, which is linear.***
* ***We can express the number of operations involved using the following recurrence relations***

***T(1)=1***

***T(n) =2T(n/2)+n***

* ***Going further down using the same logic***

***T(n/2)=2T(n/2)+n/2***

* ***Continuing in this manner, we can write***

***T(n)=nT(1)+nlogn***

***=n+nlogn***

***T(n)=0(nlogn)***

1. ***Quick Sort:***

* ***Best case***
  + 1. ***The best case analysis assumes that the pivot is always in the middle***
    2. ***To simplify the math, we assume that the two subsists are each exactly half the size of the original T(N)=T(N/2)+T(N/2)….+1 leads to T(N)=O(nlogn)***
* ***Average case***

***T(N)=O(nlogn)***

* ***Worst case***

1. ***When we pick minimum or maximum as pivot then we have to go through each and every element so***
2. ***T(N) = O(n2)***