CSE 564
10/04/2015
Julius D. Higiro

1. Formulate Problem. Take as input a sequence of N randomly ordered objects and output the sequence of N objects in sorted order using the divide and conquer method.

2. Pseudocode.

```
A is a sequence of N natural numbers.
mergeSort(A)
n is the length of A.
   if(n<=1)       // line 4
         return   // line 5
   midIndex = n/2 // line 6
   leftSubArray is A[0 to midIndex] // line 7
   rightSubArray is A[midIndex to n-1] // line 8
   mergeSort(leftSubArray) // line 9
   mergeSort(rightSubArray) // line 10
   merge(A, leftSubArray, rightSubArray) // line 11

merge(A, leftArray, rightArray)
   nL is the length of leftArray.
   nR is the length of rightArray.
   set i, j, and k equal to 0
   while i < nL and j < nR
         condition: leftArray[i] is <= to rightArray[i]
         if condition is true then A[k] = leftArray[i] and i++
         else A[k] = rightArray[j] and j++
   k++
   while i < nL
         A[k] = leftArray[i],  i++ and k++

   while j < nR
         A[k] = rightArray[j], j++ and k++
```

Loop invariant of the merge function:
- A[0 to k-1] contains the lowest valued objects of the subarrays leftArray[0 to nL-1] and rightArray[0 to nR-1].
- leftArray[i] and rightArray[j] are the lowest valued objects of their respective subarrays leftArray[0 to nL-1] and rightArray[0 to nR-1] that are not present in the array A.

Proof of correctness:
In order to prove the correctnes of the merge sort algorithm, we start by proving the correctness of the merge function. The proof of correctness for the merge function requires that we illustrate that the loop invariant for the merge function holds at the following instances:

(1) invariant holds before the initial iteration of the while loop.
(2) invariant holds for every every iteration of the while loop.
(3) invariant holds after the while loop terminates.

Assuming the loop invariant holds at the specified instances and the merge function holds true, we can now prove the correctness of the above merge sort algorithm.

P(n): the merge sort algorithm takes a sequence of n natural numbers and returns a sorted sequence of n natural numbers.

P(1): the merge sort algorithm returns a sequence that consists of 1 object per lines 4 & 5 of the pseudocode, which is sorted.

Assume that for every natural number k, where 0 <= k <=n, the merge sort algorithm returns a sorted sequence of k natural numbers.

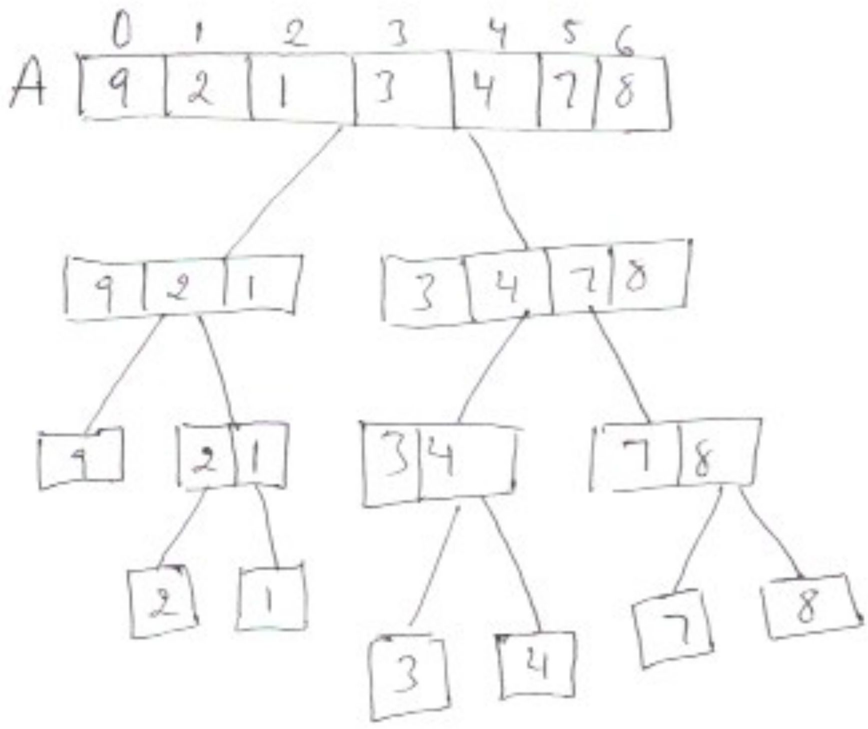
Let k = n + 1

Now, we have to show that the merge sort algorithms returns a sorted sequence of n+1 natural numbers:

Since k > n, the program will execute lines 6 through 11. Therefore, line 9 will produce sub sequences of A[0 to k/2]. Similarly, line 10 will produce sub sequences A[k/2 to k-1]. Since, we have assumed that the merge function holds, then the sub sequences will be recursively sorted and merged into a single sequence of k natural numbers.

3. The Python implementation of the algorithm is presented below and it is included in the zip file as a compilable program.

4. The merge sort algorithm recursively divides the sequences of objects into two sequences of length n/2. As show in the image below, the division of the sequence creates a recursion tree that has log n levels and each level has n operations, so the time complexity of the merge sort algorithm is O(nlogn).



In contrast, the time complexity of the quick sort algorithm is O(n^2) for the worst case. The merge sort algorithm is faster than the worst case time complexity of the quick sort algorithm. However, the time complexity of quick sort algorithm can be transformed to O(nlogn) by selecting the proper pivot point in the sequence. As such, the quick sort algorithm is the preferred sorting algorithm because the sorting of the sequence occurs inline. This is in contrast to the merge sort algorithm, which creates additional arrays in which to combine the merged subarrays and this imposes additional memory requirements.

5. The mergesort algorithm was discovered by John von Neumann.

```
#!/usr/bin/python

def mergeSort(a):

    # n is the length of the array a
    n = len(a)

    # base case
    if(n<=1):
        return

    # Identify the middle index of the array
    # the index is used to divide the array into two parts
    middleIndex = n/2
    # fill the left subarray
    leftSubArray = a[0:middleIndex]

    # fill the right subarray
    rightSubArray = a[middleIndex:n]

    # recursively divide the left part of the subarray
    # into smaller parts
    mergeSort(leftSubArray)

    # recursively divide the right part of the subarray
    # into smaller parts
    mergeSort(rightSubArray)

    # merge and sort the subarrays into the sorted
    # solution for the original array
    merge(a, leftSubArray, rightSubArray)

def merge(a, left, right):

    leftLength = len(left)
    rightLength = len(right)
    i = j = k = 0

    # compare elements of left and right arrays and merge
    # into a single array
    while(i < leftLength and j < rightLength):

        # case where element in left array is less or equal to
        # the element in right array
        if(left[i] <= right[j]):
            a[k] = left[i]
            i+=1

        # case where element in right array is greater than
        # the element in left array
        else:
            a[k] = right[j]
            j+=1

        k=k+1

    # two possible situations may arise during sort: situation 1
    # or 2 but not both.

    # situation 1: case where there exists elements that are
    # leftover in left array insert into the merged array
    while(i < leftLength):
        a[k] = left[i]
```

```
            i+=1
            k+=1

        # situation 2: case where there exists elements that are
        # leftover in right array
        # insert into the merged array
        while(j < rightLength):
            a[k] = right[j]
            j+=1
            k+=1

def main():

    data = [3,7,5,9,2,8,10,6,-9,4,1]
    mergeSort(data)
    print(data)

main()
```