Name: Pheakdey Luk
ID: 986591

Assignment 5

R-4.2 Give a pseudo-code description of the merge-sort algorithm. You can call the merge algorithm as a subroutine.

Answer:

```
Algorithm mergeSort(S,C)
    Input:sequence S with n elements, Comparator C.
    Ouput: Sequence S sorted according to C.

    if(S.size() > 1)
        S1 <- partition(S,0,n/2-1)
        S2 <- partition(S,n/2,n)
        mergeSort(S1,C)
        mergeSort(S2,C)
        S->merge(S1,S2)



Algorithm partition(S,i,j)
    Input:Sequence S with the position of start and end for making partition
    Output: Sequence S1 with the elements from i to j

    S1<-empty sequence
    while i <= j do
        S1.insertLast(S.elementAtRank(i))
        i <- i + 1

    return S1



Algorithm merge(A, B, C)
    Input:sequences A and B with n/2 elements each,comparator C
    Output:sorted sequence of A U B

    S <- empty sequence
    while !A.isEmpty() ^ !B.isEmpty() do
        if C.isLessThan( B.first().element(),A.first().element() ) then
            S.insertLast(B.remove(B.first()))
        else
            S.insertLast(A.remove(A.first()))
        while not A.isEmpty() do
            S.insertLast(A.remove(A.first()))
        while not B.isEmpty() do
            S.insertLast(B.remove(B.first()))
    return S
```

R-4.5 Suppose we are given two n-element sorted sequences A and B that should not be viewed as sets (that is, A and B may contain duplicate entries). Give an O(n)-time pseudo-code algorithm for computing a sequence representing the set A ∪ B (with no duplicates).

Answer:

```
Algorithm removeDuplicateAndUnion(A, B)
    Input:sequences A and B with n elements each
    Output:sorted sequence of A U B

    S <- empty sequence
    while !A.isEmpty() ^ !B.isEmpty() do
        if B.first().element() < A.first().element() then
            S.insertLast(B.remove(B.first()))
        else B.first().element() > A.first().element() then
            S.insertLast(A.remove(A.first()))
        else
            S.insertLast(A.remove(A.first()))
            B.remove(B.first())

    return S
```

R-4.9 Suppose we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at rank (index) $\lfloor n/2 \rfloor$, that is, an element in the middle of the sequence. What is the running time of this version of quick-sort on a sequence that is already sorted?

Answer:

If the middle element of a sorted sequence 'S' is selected as a pivot then,
-Size of both Lower Partition (L) and Greater partition (G) will be always at least S/4.
-So the height of the quick-sort tree will be $\mathbf{\log_{4/3} n}$
-The running time for each depth is **O(n)**
-Therefore, total running time of quick sort will be $\mathbf{O(n\log_{4/3} n)}$**→O(nlogn).**

C-4.10 Suppose we are given an n-element sequence S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Without making any assumptions about who is running or even how many candidates there are, design an O(n log n)-time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins.

Answer:

```
Algorithm electionWinner(S)
    Input: n-element sequence S where each element represent a different vote
    Ouput: ID of winning candidate
    mergeSort(S,C)
    winnerId <- S.first()
    maxVote <- 0
    previousId <- S.first()
    while !S.isEmpty() do
        currentId <- S.remove(S.first())
        if currentId != previousId then
            if maxVote < noOfVote
                maxVote <- noOfVote
                winnerId <- currentId
        else
            previousId <- currentId
            noOfVote<- noOfVote + 1
```

Let L be a List of objects colored either red, green, or blue. Design an in-place algorithm sortRBG(L) that places all red objects in list L before the blue colored objects, and all the blue objects before the green objects. Thus the resulting List will have all the red objects followed by the blue objects, followed by the green objects. Hint: use the method swapElements to move the elements around in the List. To receive full credit, you must use positions for traversal, e.g., first, last, after, before, swapElements, etc. which is necessary to make it in-place.