Chanh Dao Le – 986178

Assignment 5

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

```
Algorithm mergeSort(S, C)
        Input: Sequence S, comparator C
        Output: Sequence S sorted according to C
        If S.size() > 1 then
                (S1, S2) <- partition(S, n/2)
                mergeSort(S1, C)
                mergeSort(S2, C)
                S <- merge(S1, S2, C)
        return S
Algorithm merge(A, B, C)
        Input: Sequences A and B with n/2 elements each, comparator C
        Output: Sorted sequence of A v B
        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 ¬!A.isEmpty() do
                S.insertLast(A.remove(A.first()))
        while ¬B.isEmpty() do
                S.insertLast(B.remove(B.first()))
```

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).

```
Algorithm merge(A, B, C)
        Input: Sequences A and B with n elements each, comparator C
        Output: Sorted sequence of A v B
        lastElement <- NULL
        while ¬A.isEmpty() ^ ¬B.isEmpty() do
                if C.isLessThan(B.first().element(), A.first().element()) then
                        p <- B.remove(B.first())
                        element <- p.element()
                        if(lastElement = NULL v ¬C.isEqual(element, lastElement)) then
                                S.insertLast(p)
                                lastElement = element
                else
                        p <- A.remove(A.first())
                        element <- p.element()
                        if(lastElement = NULL v ¬C.isEqual(element, lastElement)) then
                                S.insertLast(p)
                                lastElement = element
        while ¬!A.isEmpty() do
```

```
            p <- A.remove(A.first())
            element <- p.element()
            if(¬C.isEqual(element, lastElement)) then
                  S.insertLast(p)
                  lastElement = element
      while ¬B.isEmpty() do
            p <- B.remove(B.first())
            element <- p.element()
            if(¬C.isEqual(element, lastElement)) then
                  S.insertLast(p)
                  lastElement = element
```

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?

The running time is O(n log n)

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.

```
Algorithm getWinner(S)
      Input: Voting sequence S
      Output: Winning candidate Id
      C <- Comparator to compare candidate Id
      sortedS <- mergeSort(S, C)              // O(n log n)
      lastCandiateId <- NULL
      winningCandidateId <- NULL
      maxVote <- 0
      count <- 0
      iterator <- sortedS.elements()
      while iterator.hasNext() do             // O(n)
            candidateId <- iterator.nextObject()
            if lastCandiateId ¬= NULL && ¬C.isEqual(lastCandiateId, candidateId) then
                  if maxVote < count then
                        maxVote <- count
                        winningCandidateId <- lastCandiateId
                  count <- 0
            lastCandiateId = candidateId
            count < count + 1
      if maxVote < count then
            maxVote <- count
            winningCandidateId <- lastCandiateId
      return winningCandidateId
```

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.

```
Algorithm sortRBG(L)
        Input: List L
        Output: Sorted list L
        C <- Comparator to compare red, green and blue objects such that red < blue < green
        inPlaceQuickSort(L, L.first(), L.last(), C)
Algorithm inPlaceQuickSort(L, l, r, C)
        Input: List L, ranks l and r, and comparator C
        Output: Sorted sequence S
        if r ¬= NULL  ∧ L.after(r) ¬= l then
                p <- inPlacePartition(L, l, r, C)
                inPlaceQuickSort(L, l, L.before(p))
                inPlaceQuickSort(L, L.after(p), r)
Algorithm inPlacePartition(L, lo, hi, C)
        Input: List L, positions lo and hi, and comparator C
        Output: The pivot is now stored at its sorted rank
        pivot <- hi
        j <- lo
        k <- L.before(hi)
        if hi ¬= NULL  ∧ L.after(hi) ¬= lo then
                while k ¬= NULL  ∧ L.after(k) ¬= j
                        if C.isLessThan(j.element(), pivot.element()) then
                                j <- L.after(j)
                        else if C.isGreaterThan(k.element(), pivot.element()) then
                                k <- L.before(k)
                        else
                                L.swapElements(j, k)
        L.swapElements(pivot, j)
        return j
```