IMPERATIVE PROGRAMMING 2009 MOISE GABRIEL

QUESTION 3

```
// Companion object
object HashBag {
    // (a)
    private class Node (var word : String, var count : int, var next : Node)
}
// Abstraction function: table = { {n(i).word -> n(i).count | n(i) in L (table(i).next) } |
i is in [0.. H)}
// DTI : L (table(i).next) is finite for all i in [0..H)
class HashBag {
    private def hash (word : String) : int = {
        def f (e : int, c : Char) = (e * 41 + c.toInt) % H
        word.foldLeft(1)(f)
    }
    private var H = 100
    private var N = 0
    // (b)
    private val table = new Array [HashBag.Node](H)
                                        for (i <- 0 until H) table(i)=new HashBag
                                                                     Node("?",0,
                                                                     null)
    private def find (w: String, head : HashBag.Node) : HashBag.Node = {
            var n = head.next
            while (n!= null && n.word != w) n= n.next
            n
    }
    def Tally (w: String) = {
         val h = hash(w)
         val n= find (w, table (h))
         if (n!= null) n.count += 1
         else {
          table(h).next= new HashBag.Node (w, 1, table (h).next)
          N += 1
          }
    }
```

```
// (c)
def sortList (n : Int) : Unit = {
    var head = table(n)
    var current = head.next
    // invariant I : the nodes up to current are sorted decreasingly
    while (current != null)
    { var n1 = current.next
      var prev = head
      var pos = head.next
      while (pos.count > n1.count) {pos = pos.next ; prev = prev.next}
      // Putting n1 between prev and pos as prev.count > n1.count ≥ pos.count (except
for the dummy header case, where we consider its count to be ∞)
     prev.next = n1
     n1.next = pos
     // Deleting n1 from its previous position
     current.next = current.next.next
    }
}
// (d)
// The resulting list will be in the first list
def mergeList (i : Int, j : Int) : Unit = {
    var head1 = table(i)
    var head2 = table(j)
    var prev = head1
    var pos = head1.next
    var current = head2.next
    while (current != null)
    {
       // We take every node from the second list and insert it in the first one,
maintaining the decreasing order, starting from where we stopped last time as the lists are
already sorted decreasingly
       while (pos.count > current.count) {prev = prev.next ; pos = pos.next}
       var n1 = current
       // Insert the node in the first list
       prev.next = n1
       n1.next = pos
       // Continue the procedure for the other nodes of the second list
       current = current.next
```

```
    }
  }

  //(e)
  def sortAll = {
      var i=0
      for (i <- 0 until H) sortList(i)
      // Merging the lists in pairs and combining the pairs until we get to one
      var lists = H
      while (lists > 1)
      { i = 0
        var del = 0
        while (i < lists/2) { mergeList(i, lists-i-1) ; i+= 1 ; del += 1}
        lists= lists-del
      }
  }

  //f)
  /*
```

We'll say that $N/H = a$, with $a$ = constant, as we are told that $H/N$ stays constant as $N$ becomes large. The sortList function requires $O(t^2)$ time-complexity, where $t$ is the size of the list to be sorted, so on average it needs $O(N^2/H^2)$, and in the worst case scenario, for a list with $N$ words, $O(N^2)$.

The function mergeList requires $O(t_1+t_2)$ time complexity, where $t_1$ and $t_2$ are the sizes of the two lists to be merged together. If all the lists have $N/H$ elements, the function will run in $O(N/H)$ time.

Sorting each list will take $H \cdot O(N^2/H^2) = O(N^2/H) = O(aN) = O(N)$ time complexity on average and $O(N^2)$ in the worst-case scenario.

Merging all the sorted list will take $\frac{H}{2}\cdot 2a + \frac{H}{4}\cdot 4a + \dots + \frac{H}{2^{\lceil \log_2 H \rceil}}\cdot 2^{\lceil \log_2 H \rceil} a = O(H \log H)$ complexity, and as $N = a \cdot H$, it will take $O(N \log N)$ time complexity, whereas in the worst-case scenario, when all the words are in a list, it still needs $O(N \log N)$.

So, our sortAll function needs, on average $O(N \log N)$ time, and in the worst-case scenario $O(N^2)$.