SEMINAR 1

MODEL ANSWERS

Week 5 – Seminar 1

Exercise 1:

The brute force approach tries all possible pairs of numbers from the given list.

```
Function kSum(numbers:List, target:int):List
    pairs = empty List
    for i := 0 to numbers.size()-2 do
        for j := i+1 to numbers.size()-1 do
            if numbers[i] + numbers[j] = target then
                pairs.append((numbers[i], numbers[j]))
            endif
        endfor
    endfor
    return pairs
```

This approach works on unsorted lists too. An interesting task is to count how many times we do the comparison numbers[i] + numbers[j] = target in this algorithm. It is in the order of n^2 where n is the number of elements in the given list.

A cleverer approach would make only n such comparisons. For this we need two pointers that will move asynchronously. One moving from the start of the list and one moving from the end of the list. We take advantage of the list being sorted. If the sum of the two elements of the list referred to by the pointers is smaller than the target, that means we need to use larger numbers and therefore we move the pointer closest to the start of the list. If the sum is larger than the target, it means we need smaller numbers and we decrease the pointer closest to the end of the list.

```
Function kSum(numbers:List, target:int):List
    pairs = empty List
    start = 0
    end = numbers.size()-1
    while start < end do
        if numbers[start] + numbers[end] = target then
            pairs.append((numbers[start], numbers[end]))
            start += 1
            end -= 1
        else if numbers[start] + numbers[end] < target then
            start += 1
        else
            end -= 1
        endif
    endwhile
    return pairs
```

This algorithm works only on sorted list. Run the algorithm manually to understand how it works. How would you modify it so it works with sorted lists containing duplicates?

Exercise 2:

There is a naïve approach to this problem. First, we add all the elements of `listA` into the returned list, and then we add the elements of `listB` to the returned list, ensuring the list remains sorted.

```
Function merge(listA: List, listB: List): List
    merged = empty List
    foreach element in listA do
        merged.append(element)
    endforeach

    foreach element in listB do
        insertIndex = 0
        // this is a comment
        // Find the position to inser the element
        while insertIndex < merged.size() do
            if element < merged[insertIndex] then
                break
            else
                insertIndex += 1
            endif
        endwhile

        // if not at the end of the list use insert
        // otherwise append at the end of the list
        if insertIndex < merged.size() then
            merged.insertAt(element, insertIndex)
        else
            merged.append(element)
        endif
    endforeach

    return merged
```

A better approach take advantage of the two list being sorted. We use to pointers, one pointing to an element of `listA` and one pointing to an element of `listB`, both initially pointing to the start of each list. We then compare the two elements referred by the pointers, if the element of `listA` is smaller we append it to the merged list and increase the pointer of `listA`, otherwise we append the element of `listB` and increase its pointer.

```
Function merge(listA: List, listB: List): List
    merged = empty List
    indexA = 0
    indexB = 0
    while indexA < listA.size() and indexB < listB.size() do
        if listA[indexA] < listB[indexB] then
            merged.append(listA[indexA])
            indexA += 1
        else
            merged.append(listB[indexB])
            indexB += 1
        endif
    endwhile

    // At this stage we have been through one of the two
    // lists and we need to append the remaining elements
    // of the other list. Note that one of the while loops
    // below will not be executed.

    while indexA < listA.size()do
        merged.append(listA[indexA])
        indexA += 1
    endwhile

    while indexB < listB.size() do
        merged.append(listB[indexB])
        indexB += 1
    endwhile

    return merged
```

Exercise 3: *reinventing the wheel!*

We need to define what a word or a token is, it is a series of characters between to delimiters. We need an accumulator to store the word under construction, as long as the character is not a delimiter, add it to the word. When we encounter a delimiter, we are at the end of the word, so add it to the list and reinitialise the accumulator to an empty string.

```
Function splitText(text:String, delimiters:String):List
    words = empty List
    wordUnderConstruction = empty String
    foreach char in text do
        if not delimiters.contains(char) then
            // continue to build the word
            wordUnderConstruction.append(char)
        else //probably end of a word
            if wordUnderConstruction != empty string then
                // we are at the end of a word
                words.append(wordUnderConstruction)
                // reinitialise to empty string to start
                // new word
                wordUnderConstruction = empty string
            endif
        endif
    endforeach

    // be careful of not omitting the last word
    if wordUnderConstruction != empty String then
        words.append(wordUnderConstruction)
        wordUnderConstruction = empty string
    endif

    return words
```