COMP 250: Introduction to Computer Science
Assignment 1 - Sample solutions

1. [15 points] **More on list intersection**

   Suppose that the arrays provided as arguments for listIntersection might contain duplicates. Write, in pseudocode, a list intersection algorithm for this case. You will have to return the number of *unique* elements in the intersection. E.g., for arrays:
   1 3 2 3 5
   and
   1 3 4 1 7 3
   your algorithm should return 2, because two distinct values (1 and 3) occur in both arrays.

   Solution: There are several ways of doing this, here we look at a solution which relies on search. We will loop over the first list, look for the current element in the first part. If found, we would have accounted for it already and we skip to the next element. If not accounted for yet, we will look for it in the second list. The code is very similar to the one we saw in class:

   Algorithm: listIntersection(a,n, b,m)
   Input: An array a of n strings and an array b of m strings.
   Output: The number of elements present in both a and b

   int $intersect \leftarrow 0$
   for $i \leftarrow 0$ **to** $n-1$ **do**
   if (**not** search(a, $i-1$, $a[i]$)) **then**
   if search(b,m,$a[i]$) **then**
   $intersect \leftarrow intersect + 1$
   return $intersect$

2. [20 points] **A recursive algorithm problem**

   A king owns a collection of n gold coins, stored in an array a. All coins have exactly the same weight except for exactly one that is actually filled with lead and is thus heavier than the normal gold coins. The king asks you to find the index of the lead coin as quickly as possible. You own a scale that allows you to compare the total weight of any two regions $a[i \ldots j]$ and $a[k \ldots j]$ of the array. Let us represent this scale by a method called compare(a, i, j, k, l), defined as:

$$\text{compare}(a,i,j,k,l) = \begin{cases} -1, & \text{if } a[i \ldots j] \text{ has weight smaller than } a[k \ldots l] \\ +1, & \text{if } a[i \ldots j] \text{ has weight greater than } a[k \ldots l] \\ 0, & \text{otherwise} \end{cases}$$

   Using the compare method, write a recursive algorithm (in pseudocode) that returns the index of the lead coin. Your algorithm should make $O(\log n)$ calls to the compare method (but you do not need to prove this). Your algorithm has to work for any value of $n > 0$.

Solution: The idea is to mimic binary search, and recurse on the heavier side of the array. However, we need to make sure we treat even and odd array sizes correctly.

Algorithm FindHeavyCoin (a,n)
Input: An array a of size n
Output: The index of the heavy coin
return FindHeavyCoinRec (a,0,$n-1$)

Algorithm FindHeavyCoinRec (a, i, j)
Input: An array a and indices i and j in between which we seek the heavy coin
Output: The index of the heavy coin

if $i = j$ **then return** i
if ($j - i + 1$ **mod** $2 == 1$) **then**
    // This is the case with even number of elements (as we search between i and j inclusive)
    if compare($a, i, (j+i)/2, (j+i)/2+1, j$)$< 0$ **then**
        return FindHeavyCoinRec($a, (j+i)/2+1, j$)
    else return FindHeavyCoinRec($a, i, (j+i)/2$)
else // We have an odd number of elements
    if compare($a, i, (j+i)/2-1, (j+i)/2+1, j$)$< 0$ **then**
        return FindHeavyCoinRec($a, (j+i)/2+1, j$)
    else if compare($a, i, (j+i)/2-1, (j+i)/2+1, j$)$> 0$ **then**
        return FindHeavyCoinRec($a, i, (j+i)/2-1$)
    else return $(j+i)/2$

3. [25 points] **An array problem**

   (a) [20 points] Suppose that you are given an array of integers. Write a Java program which prints out the element or elements that occur *least often* in the array, and the number of times they occur. For example, for array:
   1 4 4 3 4 3 5 2 1
   your algorithm should print the message:

   Elements: 5 2
   Number of occurrences: 1

   The implementation should be done in the main method of your class.
   Solution: The solution is in LeastFrequentElement.java

   (b) [5 points] What is the time complexity ($O()$) of your algorithm? Justify your answer.
   Solution: The first two nested loops have $O(n^2)$ in the worst case (which corresponds to having all distinct elements in the original array). Finding the min count and then printing the elements are both $O(n)$. Hence, the sequence results in $O(n^2)$. Note that other solutions are possible too, e.g. one could make a copy of the original array and sort it, then traverse

it once and find the shortest contiguous segment in which all elements have the same value. This would yield $O(n \log n)$ complexity.

4. [40 points] **Polynomials**

   In this problem, you will extend the Poly class we are using in the lectures.

   Solution: See the updated version of Poly.java

   (a) [15 points] Write a method for multiplying two polynomials, with the signature:

   Poly multiply(Poly p) throws Exception

   Your method should throw an Exception if the argument p is null.

   (b) [5 points] Write a method which multiplies a polynomial by a constant. It should work by transforming the constant into a polynomial of degree 0, then calling the multiply method.

   (c) [10 points] Write a method that checks if there are any leading zeros in the coefficients of the polynomial. If so, it should re-size the coefficient array appropriately. Nothing is returned.

   (d) [10 points] Write a method that computes the derivative of a polynomial (and returns it as another polynomial)