1. Design an algorithm that, given two lists of integers, creates a list consisting of those integers that appear in both lists (each integer on the final list should appear only once). Describe your algorithm in terms of a high-level pseudo code focusing on main algorithmic tasks and not on low-level details. Analyze the running time of your algorithm. You will get full credit only if your algorithm achieves an asymptotically better worst-case performance than Θ(n2), where n is the sum of the lengths of the two input lists.
    a. While (j<=n1 and k<= n2){

```
If A[j] < B[k]{ j+=1}
Else if (A[j] > B[k]) {k +=1}

If I == 0 {C[i]: C[i] = A[j]
J+=1,k+=1,i+=1

If A[j] ~= C[i-1]
C[i] = A[i]

I+=1,j+=1,k+=1

Return C;
```

Therefore the algorithm above has a worst case of O(nlogn), because it is a derivation of mergesort.

2. Give a high-level pseudo code for an algorithm that, given a list of n integers from the set {0, 1, . . . , k – 1}, preprocesses its input to extract and store information that makes it possible to answer any query asking how many of the n integers fall in the range [a..b] (with a and b being input parameters to the query) in O(1) time. Explain how your algorithm works.
    a. function sort(A[1…k],a,b):

```
Preprocessing:
for i = 1 to k:
        c[i] = 1
for i = 1 to n:
        c[a[i]] = c[A[i]] + 1
for i = 1 to k:
        c[i] = c[i-1] + c[i]
```

Query:
if a = 0
    return c[b]
else:
    return c[b] – c[a-1]

Preprocessing is 0(n+k) due to the for loops that iterate from 1 to k. The query is 0(1) as it is enough to return c[b] - c[a-1].

3. Describe an algorithm (high-level pseudocode) to sort a list of n integers, each in the range $[0..n^2 - 1]$, in O(n) time. Justify the correctness and the running time of your algorithm. Generalize to an arbitrary *constant integer* k. That is, describe an algorithm to sort a list of n integers, each in the range $[0..n^k - 1]$, in O(n) time.
    a. Function x(a[1…n]
       Input: an array a with integers in range 0 to n^2 – 1
       Output: a sorted a

       For i = 0 to n - 1:
           a[i] = ((a[i] – a[i] mod n) / a[i] mod n)
       radixsort(a)
       for i = 0 to n -1:
           a[i] = a[i][0] * n + a[i][1]
       return a

       Radix sort is O(n), therefore this algorithm will be O(n)

4. Describe (in high-level pseudocode) an algorithm to find the maximum element in a unimodal sequence of integers $x_1, x_2, . . . , x_n$. The running time should be O(log n). Show that your algorithm meets the bound.

a. Mode(a)
   N = A.length
   If n == 1
      i. Return 1
   mid = floor(n/2)
   if A[mid] > A[mid + 1]
      return mode(a[1…mid]

   else
      return mid + mod(a[mid + 1..n]

5. Describe an algorithm to merge k sorted lists containing altogether n elements into one sorted list. Give a pseudo-code. The algorithm must run in time O(n log k). Show that your algorithm meets the bound.

```
lists[k][?]     // input lists
c = 0           // index in result
result[n]       // output
heap[k]         // stores index and applicable list and uses list value for comparison
                // if i is the index and k is the list
                //   it has functions - insert(i, k) and deleteMin() which returns i,k
                // the reason we use the index and the list, rather than just the value
                //   is so that we can get the successor of any value

// populate the initial heap
for i = 1:k                 // runs O(k) times
  heap.insert(0, k)         // O(log k)

// keep doing this - delete the minimum, insert the next value from that listinto the heap
while !heap.empty()         // runs O(n) times
  i,k = heap.deleteMin();   // O(log k)
```

```
result[c++] = lists[k][i]
i++
if (i < lists[k].length)    // insert only if not end-of-list
  heap.insert(i, k)         // O(log k)
```