1 Problem 1 (20 points)

Devise an algorithm which, given two length-n arrays of numbers A and B, returns a list C of all elements x that appear in both A and B. That is, C should be a list of all elements in the set

$$X = \{x : A[i] = x, B[j] = x \text{ for } 1 \leq i \leq n, 1 \leq j \leq n\}.$$

Your algorithm should run in time $O(n \log n)$ for full credit. Prove the correctness of your algorithm and prove its runtime.

Solution: It suffices to sort lists A and B using mergesort, and then use the following algorithm to find all elements that appear in both A and B.

```
i ← 1, j ← 1
C ← ∅
while i ≤ n and j ≤ n do
  if A[i] = B[j] then
    C ← C ∪ {A[i]}
    x ← A[i]
    while A[i] = x do
      i ← i + 1
    end while
    while B[j] = x do
      j ← j + 1
    end while
  else if A[i] > B[j] then
    j ← j + 1
  else
    i ← i + 1.
  end if
end while
return C
```

Correctness: after calling mergesort on A and B, we know that both A and B are sorted in ascending order by the correctness of mergesort. Thus we need only prove the correctness of the algorithm just given. First, we will show that if there exists an element x that appears in both array A and array B that it will be placed into set C. Then we will show the converse. This will imply that an element x appears in C if and only if x appears in both list A and list B.

Consider an element x appearing in both A and B, say $A[a] = x$ and $B[b] = x$. Assume further that a and b are the lowest such indices in A and B containing element x.

Firstly, by construction of the algorithm, there exists an iteration of the main while loop during which, at some point, $i = a$. Similarly, there exists an iteration of the main while loop during which, at some point, $j = b$. We must show that these two iterations are indeed the same iteration. Suppose without loss of generality that $j = b$ at the beginning of the main while loop. We must have $i \leq a$, since if $i > a$, we would have already had an iteration of the main while loop during which $i = a$ and $j < b$. Since both A and B are sorted in ascending order, we would have incremented index j until $j = b$. Since $i \leq a$ and since A is sorted, we must have $A[i] < A[a] = B[j]$, and thus the algorithm will increment i until $i = a$, at which point we will have $A[i] = B[j] = x$, and we will add x to C.

Conversely, note that if x is included in C, then by construction of the algorithm, this could only have happened if there existed i, j for which $A[i] = x = B[j]$ (this is the only condition under which we add elements to C). Thus, $x \in C$ if and only if x appears in lists A and B.

Runtime: we require $O(n \log n)$ time to sort arrays A and B. The algorithm after sorting traverses both sorted arrays "in parallel", terminating when we reach the end of one or the other. Thus, we increment i and j at most $2n$ times in total over the course of the second-stage algorithm, and we conclude that this procedure runs in time linear in n. Thus our overall algorithm requires $O(n \log n + n) = O(n \log n)$ time.

2 Problem 2 (20 points)

A binary array is one that contains only 0s and 1s. Devise an $O(n)$ time algorithm for sorting a binary array of length n. Prove the correctness of your algorithm and its runtime.

Solution: it is simple enough to count the number of 0s and the numbers of 1s in the given array. Suppse there are k 0s in the input array and $n - k$ 1s. We then simply construct a new array in which the first k elements are 0 and the rest are 1 (or, for an in-place sort, simply rewrite the entries of the input array). Note that this is just count sort. The correctness of this algorithm follows immediately from construction. The runtime is linear since we require only a linear pass through the data to count how many 0s and 1s there are.