Problem 1

To compute $A \cup B$, we begin with an empty set, C. We copy every element of A into C. This takes $\mathcal{O}(m)$ time. Next, we sort A in $\mathcal{O}(m \log m)$ time. This step will enable us to perform binary search on A in $\mathcal{O}(\log m)$ time. For each element of B, perform binary search on A. If the element is not found in A, add it to C. This takes $\mathcal{O}(n \log m)$ time, since we preform n binary searches. This totals $m + m \log m + n \log m$, or simply $\mathcal{O}(n \log m)$.

To compute $A \cap B$, we again start with an empty set, C. We start by sorting A, again in $\mathcal{O}(m \log m)$ time. For every element of B, perform binary search on A. This will total $\mathcal{O}(n \log m)$ time. If the element of B is found in A, add it to C. This will take a total of $m \log m + n \log m$ time, or just $\mathcal{O}(n \log m)$.

Problem 2

Because the intervals are disjoint, we can define a strict ordering by sorting them either their upper or lower bounds. We will choose to sort by the lower bound here. This takes $\mathcal{O}(q \log q)$ time. Next, create an empty list of size q and initialize each cell to zero. This takes $\mathcal{O}(q)$ time. For each element of X, identify the interval it belongs to by the following method: select the center interval. If the element of X is in the center interval, increment the corresponding cell in the list by one. If the element of X is greater than the intervals upper bound, perform the same operation with endpoints at the current interval and last interval, recursively. If the element of X is less than the lower bound of the interval, perform the same operation with endpoints at the current interval and the first interval, recursively. Because this is just a modified binary search, it runs in $\mathcal{O}(\log q)$ time. If we do this for every element of X, we will have a total of $q \log q + n \log q$ or just $\mathcal{O}(n \log q)$.

Problem 3

Because the list can be pre-processed to contain a list of pairs in the form of (index, value). Therefore, finding the maximum element and the index of the maximum element are the same thing. The following algorithm will Therefore simply identify the maximum element of a list, as that is a cleaner solution.

Algorithm 1 Return the maximum element of a list recursively

```
function findMax(list)
    if list.length = 1 then
        return list[0]
    else
        lhs ← findMax(list[0 .. list.length / 2])
        rhs ← findMax(list[list.length / 2 ..])
        return max(lhs, rhs)
    end if
end function
```

This algorithm calls itself twice on an array of half the size in the recursive case. We can establish the following recurrence relation. Note that comparison is constant time.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 1$$

This is a trivial case one of the master theorem (because $c = 1$ and $c < \log_2 2$). By the master theorem, this algorithm belongs in $\Theta\left(n^{\log_2 2}\right)$, or just $\Theta(n)$. This is the same complexity as the brute-force method.