(1) DPV Problem 2.16.

You are given an infinite array $A[\cdot]$ in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are not given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(log(n))$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you dont know this length, and that the implementation of the array data type in your programming language returns the error message ∞ whenever elements $A[i]$ with $i > n$ are accessed.)

The algorithm to find x in $O(log(n))$ would consist of two distinct steps, each of which would be $O(log(n))$.
The first step would be to find the bounds or size of n. To do this, create a loop which checks the $A[1]$ position for infinity, following which it will check $A[2], A[4], A[8], ...$ using powers of 2 until it finds a position with infinity. The index of that position will then be stored as a variable (I will call it m).
Following this, the second step is to conduct a binary search of the array up till $A[m]$ to find the position of x if it exists.

(2) DPV Problem 2.17.

Given a sorted array of distinct integers $A[1, ..., n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(log(n))$.

The algorithm for this involves implementing a binary search which is a form of divide-and-conquer algorithm. Most of the algorithm is like a typical binary search except the value to be found changes from a set value to a dynamic value. The pseudocode involved would resemble:

```
binarySearch(int[] A, int left, int right) {
    if(right >= 1) {
        int mid = left + (right − 1)/2;
        if(A[mid] == mid) {
            return true;
        }
        else if(A[mid] > mid) {
            return binarySearch(A, 1, mid − 1);
        } else {
            return binarySearch(A, mid + 1, r);
        }
    }
    return false;
}
```

This is the same as a usual binary search with the difference being the use of mid as a comparison rather than another integer that is taken as an input. This is because mid in this case represents i from the problem and because the array is already sorted, the indexes before when $A[i] = i$ will always have $A[i] > i$ while those after will always have $A[i] < i$. this makes it possible to conduct a straightforward binary search.

(3) DPV Problem 2.19. You may assume the elements in the arrays are integers. For part (b), say what the runtime complexity of your algorithm is. It should be better than just combining them into one array and re-sorting them from scratch.

A k-way merge operation. Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

(a) Heres one strategy: Using the merge procedure from Section 2.3, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?

To merge two arrays of sizes n_1 and n_2, the time complexity would be $O(n_1 + n_2)$. Each additional merge will involve adding a new value to the previous total, this will result in the form:

$(n + n) + (2n + n) + ... + ((k - 1)n + n)$
$= 2n + 3n + ... + kn$
$= k^2 n$
Therefore, the total time complexity for this algorithm will be $O(k^2 n)$.

(b) Give a more efficient solution to this problem, using divide-and-conquer.

A possible method to reduce the time complexity involves still using merge, but changing the way the k arrays combine to form one. Rather than iterating and merging each array to a single array, it is possible to first merge the first array with the second, the third with the fourth and so on matching the arrays into pairs. The time complexity of this single operation will be $O(kn)$ while there will now be $k/2$ remaining arrays. Those $k/2$ arrays can the be merged to form $k/4$ arrays and so on till a single array is left remaining. To go from k arrays to a single array would require the same operation to be carried out $log(k)$ times with a time complexity of $O(kn)$ each time. Therefore , the total new time complexity would be $O(kn \cdot log(k))$.

(4) DPV Problem 2.22. You can provide an algorithm with running time either $O(log(m) + log(n))$ or $O(log(m + n))$.

You are given two sorted lists of size m and n. Give an $O(log(m) + log(n))$ time algorithm for computing the kth smallest element in the union of the two lists.

Such an algorithm involves using a binary search through both arrays in order to find the value of the kth element. Therefore, by combining two binary search algorithms for arrays of size m and size n, it is possible to find the kth element. THis could be done in the form of a function roughly shown in the pseudo code below:

```
compare(int* A1start, int* A2start, int* A1end, int* A2end, int k) {
    if(A1start == A1end) {
        return A2[k];
    }
    if(A2start == A2end) {
        return A1[k];
    }
    int mid1 = A1[(A1end − A1start)/2];
    int mid2 = A2[(A2end − A2start)/2];
    if (mid1 + mid2 < k) {
        if(A1[mid1] > A2[mid2]) {
            return compare[A1start, A2start + mid2 + 1, A1end, A2end, k − mid2 − 1];
        } else {
            return compare[A1start + mid1 + 1, A2start, A1end, A2end, k − mid2 − 1];
        }
    } else {
        if(A1[mid1] > A2[mid2]) {
            return compare[A1start, A2start, A1start + mid1, A2end, k];
        } else {
            return compare[A1start, A2start, A1end, A2start + mid2, k];
        }
    }
}
```

The function can be considered $O(log(m) + log(n))$ because it at most combines either $O(log(m))$, $O(log(n))$, or a combination of the two.
The function works like a normal binary search except with two binary searches combined into it. Up till the creation of $mid1$ and $mid2$ is the usual binary search though it then changes slightly. The first if condition works mainly to reduce the space to the left of the kth and reduce k appropriately. The else reduces to the right. Thus, the two will continuously narrow down the search by a factor of two until the initial if statements are validated and returned.