Introduction to Algorithms - Homework #5
Jonah Mudse (mudse)

2.16. You are given an infinite array A[·] 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 don't 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.)

Goal: find a good estimate for *n* in O(log n) then run a binary search, also O(log n).

To find a good estimate for *n* in O(log n), we can use a random positive integer *k.* For implementation, you would want *k* to be as small as possible but still bigger than 1, so lets use 2.

With an infinite array, we don't know where the 'right end' is but we do know where the 'left end is', so lets start there.

Starting at array[0], then array[1], then array[2]. Array[4], array[8], … array[2n]. These numbers are just multiples of *k* starting with 0 and 1. In this case, we used 2.

This will give us an estimate of *n* that is no greater than double *n* and no less than *n* and we found it by only checking multiples of 2, which can be done in $O(log_2 2n)$ which is just $O(log\ n)$.

Now we can use a binary search to find integer x if it exists which takes $O(log\ n)$.

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).

Since binary search is a form of divide and conquer, we can modify the algorithm to solve this problem for us. At each 'divide' of the array, instead of comparing the *middle* element to whatever we are searching for, we can compare it to its position within the array and continue until we have found an element that's value is equal to its position or we are out of elements to check.

At each comparison, if the middle element's value > its position, we *throw out* the upper half since we now only care about the lower positions. If the middle element's value < its position, we *throw out* the lower half since we now only care about the higher positions. Basically, since the array is sorted, we are able to determine which half A[i] = i must exist if it does.

2.19. 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.

Introduction to Algorithms - Homework #5
Jonah Mudse (mudse)

(a) Here's 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?

Merging two sorted lists (say lists *a* and *b*) can be done in $O(a+ b)$.
Since we are sorting a list of size *n* with another of the same size (now size *2n*), then with another of size *n* (now *2n* + *n* = *3n*), then again with size *n* (now *3n* + *n* = *4n*), etcetera all the way to (*k-1)n+n,* we get the following series:

$(2+3+4+5+...+k)*n$

Which can be simplified to:

$$(\frac{k(k+1)}{2} - 1) * n$$

Which further simplifies to: $O(n*k^2)$ since with Big O, we only care about the exponent of *k.*

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

For a more efficient solution, simply divide the arrays into two sets, each of *k / 2* arrays.
Then, merge each set into its own single array.
Finally, merge the two remaining sets into one.