HW6
Daniel Ginsburg

Because we can access the k^{th} smallest value (and therefore any value) in $O(1)$ time, we can view each data base as a sorted array. Let these arrays be labeled D1 and D2. Let the k^{th} smallest value in D1 be defined by D1[k-1] (we start at 0). Let $D1_m$ represent the median of the values of D1 that have not been eliminated. Let the median of a set of numbers be defined as the middle value or, in the case of an even number of values, the smaller of the two middle values

Overview:
This problem can be viewed as a twist on binary search. Binary search has been proven to be $O(log(n))$ run time. It operates by consistently eliminating half the values until either 0 or 1 value remains. Our algorithm aims to do just that. We get the median of each array. If n is odd then it is the $n/2 + 1$ value (ex. D1[n]). If n is even then we take the $n/2$ value. We then compare the median values. Everything lower than the low median (and if n is even the low median itself) in that array will be eliminated and everything higher than the higher median will be eliminated in the other array. ex. if D1 = $[1, 2, 3, 4]$ and D2 = [5,6,7,8] then $D1_m = 2$ and $D2_m = 6$. Because 2 is lower than 6, we disregard [1,2] in D1 and disregard [7,8] in D2, thus we are essentially left with D1 = [3,4] and D2 = [7,8]. Notice how 4, the true median, remains the median of the two "new" arrays. We repeat the process of comparing the medians until two values are left and return the smaller of the two, which is the median of the two (see definition of median above).

Pseudo Code:
findMedian(D1, D2){
start1, start2 = 0;
end1, end2 = length of D1 (or D2);
size = length of D1 (or D2);
findMedian(D1, D2, start1, start2, end1, end2);
}
findMedian(D1, D2, start1, start2, end1, end2){
if(only two values left){
return lower value
}
medOnePos = getMedPosition((D1, start1, end1)
medTwoPos = getMedPosition((D2, start2, end2)
med1 = D1[medOnePos]
med2 = D2[medTwoPos]
if($med1 < med2$){
(if med1 is lower eliminate lower half of D1 and upper half of D2)
start1 = $size/2$
end2 = med2;

}
if($med2 < med1$)
(if med2 is lower eliminate lower half of D2 and upper half of D1)
start2 = $size/2$
end1 = med1;
}
//(Basically cut size in half but slightly different depending on if n was odd or even)
$size = adjustSize(size)$
//run again
findMedian(D1, D2, start1, start2, end1, end2)

We prove that the median will remain the same throughout the algorithm (that the median is an invariant) and that the algorithm will always terminate, thus it will always result in the correct answer.
We know that the median (the n^{th} smallest value) is bigger than $n-1$ values and smaller than n values. Let us assume that n is even. Let us assume that when we compare two medians m1 of D1 and m2 of D2, that m2 is bigger. Because m2 is the bigger median, it is bigger than the lower half of D2 AND the lower half of D1. This means that any value greater than m2 is higher than at least n values ($n/2$ values of D1 and $n/2$ values of D2) and is therefore too high to be the true median. We can thus safely eliminate the upper half of D2. We also know that any value $\leq m1$ is lower than at least n+1 values ($n/2$ values of D1, $n/2$ values of D2, and m2) and can also safely be eliminated. (Note: that we eliminate the lower median itself but not the higher median). Because we eliminate $n/2$ values above and $n/2$ below the median, the median remains the same. ex. if 3 is the median of[1,2,3,4,5] and I eliminate 1 and 4, I am left with [2,3,5] but 3 remains the median. This proves the invariant that the median will remain the same throughout the algorithm
n is Odd:
In this case when n is odd the median will be the actual middle value. We only eliminate the values above or below the medians (which are both $(n-1)/2$. Thus the symmetry and invariant are kept. The math works out because the values above m2 (from previous example), are higher than at least n+1 values ($(n-1)/2 + m1$ from D1 and $(n-1)/2 + m2$) from D2. The values below m1 are lower than n+1 values. Thus anything higher than m2 or lower than m1 can safely be eliminated.
Termination:
Like binary search the algorithm continues to eliminate roughly half the values with each iteration (odd/even will cause slight differences). To verify this we divide into cases:
Case 1: Remaining values in each array are at least three.
The median by definition cannot be the first or last value and therefore some values will be eliminated. Specifically half the values if n is even or $(n-1)/2$ if n is odd. Essentially half the values are eliminated.

Case 2: Remaining values in each array are two
The median by definition is the first value of each array. One will be bigger than the other (no repeat values). Let us assume m1 is the lower median and m2 is the higher. In this case m1 will be eliminated and the value above m2 (in it's array) will be eliminated, leaving one value in each array.
Case 3: Each array has one value left:
The algorithm simply return the lower of the two and terminates.
Because each array must have 1,2, or at least three values it will eventually terminate.
We have proven that this algorithm always maintains the median as the lower of the two middle values, and that it terminates and returns the lower of the two values (which is the median of the two). We also know that this algorithm works like binary search by eliminating rougly half of the remaining values thus it is O(log(n)) run time. Thus it is correct.