# (Sol) HW02 Divide and Conquer

**Problem 1. (12 points)** based on Exercise 5.1 You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains $n$ numerical values in a sorted array —so there are $2n$ values total—and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the $n^{th}$ smallest value. For example, if the two databases are $[-1, 0, 5, 6, 10]$ and $[1, 3, 7, 9, 14]$, then the return value should be $5$.

The only way you can access these values is through *queries* to the databases. In a single query, you can specify a value $k$ to one of the two databases, and the chosen database will return the $k^{th}$ smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

a. **(4 points)** Explain how your algorithm finds the median value. (You actual implementation should be submitted on GitHub.)

   **Answer**: Finding a median value in each database is easy. Sending a query for $n/2$ to each database will return a median value from each database. Compare the median values, and query for a median from the smaller half of the database with a greater median, and query for a median from the larger half of the database with a smaller median. Repeat until you find $n^{th}$ value.

   Assume that query(k, db) returns the $k^{th}$ value from the database. One way to implement this query is to have a sorted array for each database, and refer to  element in the array. The following java code assumes such implementation.

```
public static int findMedian(int[] a, int[] b) {
        return findMedianHelper(a, b, (a.length+b.length)/2-1, 0, a.length-1, 0, b.length-1);

    }

    public static int findMedianHelper(int[] a, int[] b, int k, int aFrom, int aTo, int bFrom, int bTo) {
        if (aFrom > aTo) {
            return b[bFrom + k];
        } else if (bFrom > bTo) {
            return a[aFrom + k];
        } else if (k == 0) {
            return a[aFrom] < b[bFrom] ? a[aFrom] : b[bFrom];
        }

        int aMid = k/2 + aFrom;
        int bMid = (k-1)/2 + bFrom;

        if (a[aMid] < b[bMid]) {
            return findMedianHelper(a, b, k-(aMid-aFrom+1), aMid+1, aTo, bFrom, bMid);
        } else {
            return findMedianHelper(a, b, k-(bMid-bFrom+1), aFrom, aMid, bMid+1, bTo);
        }
    }
```

b. **(4 points)** Show the time complexity of your algorithm. Ideally your algorithm takes at most $O(\log n)$ queries.

**Answer**: Every time `findMedianHelper` is called, the size of the input array reduced by $k/2$. The first `findMedianHelper` call is made with $n-1$, so the size of the input starts from $n$ and goes down to $n/2$, $n/4$, $\cdots$, until one of the arrays is empty.

$T(n) = c_0$ when $n = 0$ (base case)

$= T(n/2) + c$ otherwise (recursive case)
So the overall time complexity is $O(\log n)$.

c. **(4 points)** Show the space complexity of your algorithm. (Big $O$ is enough.)

**Answer**: Every time `findMedianHelper` is called, 2 new variables `aMid` and `bMid` are created. So this version of the algorithm takes $O(\log n)$ space complexity. However, it is possible to replace `aMid` and `bMid` with the formula so the same algorithm may be implemented with $O(1)$ space complexity.