Algorithms
Homework 2 Solutions Instructor: Anna Karlin

Problems:

1. Problem 2.27 on page 83 of [DPV].

   Solution:

   (a) Suppose the matrix A is the following.

   $$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

   Then the square of A, i.e. AA, can be computed to be:

   $$\begin{pmatrix} aa + bc & b(a+d) \\ c(a+d) & bc + dd \end{pmatrix}.$$

   It is easy to see that we only need to compute five multiplications, which are $aa, bc, b(a+d), c(a+d), dd$.

   (b) There are two problems.

   First, in general, matrix multiplication does not commute. That is to say, if A, B, C, D are matrices, then AC is not equal to CA in general.

   Second, we're trying to write a recursive method to square matrices, so recursive calls need to be squaring matrices, but the five multiplications we need to compute are general matrix multiplications, not squaring.

   Therefore, we cannot get five mulitplications as we did for part (a).

   (c) i. Since $AB+BA = (A+B)(A+B)-AA-BB$, we can calculate AB + BA by squaring three matrices. Also, the addition/substraction of matrices is $O(n^2)$. Therefore, the time complexity is $3S(n) + O(n^2)$.

   ii.
   $$AB = \begin{pmatrix} 0 & XY \\ 0 & 0 \end{pmatrix}.$$

   $$BA = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

   $$AB + BA = \begin{pmatrix} 0 & XY \\ 0 & 0 \end{pmatrix}.$$

   iii. Remember that the size of A and B is $2n \times 2n$. Therefore, by (i), $AB + BA$ can be computed in $3S(2n) + O(n^2)$. Then, by (ii), we only need to read the top right corner $n \times n$ submatrix from $AB + BA$, in order to get XY, which takes $O(n^2)$ time.

In total, the complexity of computing XY is $3S(2n) + O(n^2)$.

The size of the output is $O(n^2)$, so it takes at least that long to just write down the answer. Therefore, $c \geq 2$ in $S(n) = O(n^c)$ for squaring a matrix. Hence $3S(2n) + O(n^2)$ is simply $O(n^c)$. We can conclude that matrix multiplication takes time $O(n^c)$.

2. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values – 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.

   However, 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 it contains. Since queries are expensive, you'd like to compute the median using as few queries as possible.

   Give an algorithm that finds the median value using at most $O(\log n)$ queries.

   Solution:

   We call the two databases D_1 and D_2.

   The observation is that the median must be between the median of D_1 and the median of D_2. A divide and conquer approach is designed based on this observation. Also, since we can only access the database using queries, the recursive calls must track the portion of the database that is still under consideration.

   Algorithm:

   We assume that `Query(D, k)` is the function to return the k-th smallest value in database D. Then we can run the following function `FindMedian(D1, 1, n, D2, 1, n)` to produce the median.

```
//inputs are two databases D1, D2, and the portion of each database under consideration.
// i.e., a pointer to the smallest and largest element in that database under consideratio
function FindMedian (D1, lower1, upper1, D2, lower2, upper2):
    // k1 is the pointer to specify the median in D1[lower1...upper1]
    Let k1 := lower1 + floor((upper1 - lower1)/2);
    // k2 is the pointer to specify the median in D2[lower2...upper2]
    Let k2 := lower2 + floor((upper2 - lower2)/2);
    m1 := Query(D1, k1); // get the median of D1[lower1...upper1]
    m2 := Query(D2, k2); // get the median of D2[lower2...upper2]
    if m1 > m2
       //finish searching and return the answer
       if (lower1 = upper1) and (lower2 = upper2)
            return m2;
       //continue searching in the upper half of D2[lower2...upper2] and
       //the lower half of D1[lower1...upper1]
       else
            //If (upper2 - lower2) is even, then we keep D[k2] to make sure that
            //D1[lower1...k1] and D2[k2...upper2] have the same number of elements.
            if (upper2 - lower2) is even
                return FindMedian(D1, lower1, k1, D2, k2, upper2);
            //If (upper2 - lower2) is even, then we remove D[k2] to make sure that
            //D1[lower1...k1] and D2[k2+1...upper2] have the same number of elements.
            else
                return FindMedian(D1, lower1, k1, D2, k2 + 1, upper2);
    // when m1 < m2
    else
       //finish searching and return the answer
       if (lower1 = upper1) and (lower2 = upper2)
            return m1;
       //continue searching in the upper half of D1[lower1...upper1] and
       //the lower half of D2[lower2...upper2]
       else
            if (upper1 - lower1) is even
               return FindMedian(D1, k1, upper1, D2, lower2, k2);
            else
               return FindMedian(D1, k1 + 1, upper1, D2, lower2, k2);
```

Correctness Proof: We will prove by induction.

Base case: When $n = 1$, each database has exactly one element. By running the algorithm, we return the smaller one of the two elements, which satisfies the definition of median (remember that by the definition of median, the median of a, b is a if $a < b$).

Inductive case: Suppose that the algorithm works correctly for $1, 2, \ldots, n-1$, then we will show that it is also right for n.

For n, the median of D_1 is m_1, and the median of D_2 is m_2. We will show that the algorithm works correctly for the case that $m_1 > m_2$ and that n is even. Similar arguments can be

applied to the other three cases.

When $m_1 > m_2$ and n is even, $n-1$ is odd, the algorithm returns m, the median of $D_1[1, \ldots, \frac{n}{2}]$ and $D_2[\frac{n}{2}+1, n]$. By the induction hypothesis, we only need to show that m is also the median of D_1 and D_2.

First, it is easy to see that $m_1 \geq m \geq m_2$. (If $m > m_1$, then the $n/2$ numbers in $D_1[1, \ldots, \frac{n}{2}]$ will be strictly smaller than m, contradicting the definition of median. Hence, $m \leq m_1$. Similar arguments can be used to get $m \geq m_2$.) Therefore, by the definition of the median, the numbers in $D_1[\frac{n}{2}+1, n]$ are greater than m; and the numbers in $D_2[1, \ldots, \frac{n}{2}]$ are smaller than or equal to m.

Second, by the induction hypothesis and also by the definition of median, in $D_1[1, \ldots, \frac{n}{2}]$ and $D_2[\frac{n}{2}+1, n]$, exactly $\frac{n}{2}$ numbers are smaller than or equal to m, and exactly $\frac{n}{2}$ numbers are greater than m.

In total, there are n numbers in D_1 and D_2 that are smaller than or equal to m, and there are n numbers in D_1 and D_2 that are greater than m, meaning that m is indeed the median for D_1 and D_2.

Runtime Analysis: The algorithm is a divide and conquer appoach. Each time, there is only one subproblem, the size of the problem is reduced by half (each database is kept only half of its elements), and two queries (issue one query to each database) are sent to the databases. Hence, the recurrence relation is $T(n) = T(n/2) + 2$. By master theorem, we have $T(n) = O(\log n)$.

3. Recall the problem of finding the number of inversions: You are given a sequence of n numbers $a_1, \ldots, a_n$, which we are assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

   We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitivie. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

   Solution: The algorithm is very similar to the algorithm of counting inversions. The only change is that here we separate the counting of significant inversions from the merge-sort process.

   Algorithm: Let $A = (a_1, a_2, \ldots, a_n)$.

```
Function CountSigInv(A[1...n])
       if n = 1
             return 0; //base case

       Let L := A[1...floor(n/2)]; // Get the first half of A
       Let R := A[floor(n/2)+1...n]; // Get the second half of A
       //Recurse on L. Return B, the sorted L,
       //and x, the number of significant inversions in $L$
       Let B, x := CountSigInv(L);
       Let C, y := CountSigInv(R);

       //Do the counting of significant split inversions
       Let i := 1;
       Let j := 1;
       Let z := 0; // to count the number of significant split inversions
       while(i <= length(B) and j <= length(C))
              if(B[i] > 2*C[j])
                    z += length(B)-i+1;
                    j += 1;
              else
                    i += 1;

       //the normal merge-sort process
       i := 1;
       j := 1;
       //the sorted A to be output
       Let D[1...n] be an array of length n, and every entry is initialized with 0;
       for k = 1 to n
             if B[i] < C[j]
                   D[k] = B[i];
                   i += 1;
             else
                   D[k] = C[j];
                   j += 1;


       return D, (x + y + z);
```

Runtime Analysis: At each level, both the counting of significant split inversions and the normal merge-sort process take $O(n)$ time, because we take a linear scan in both cases. Also, at each level, we break the problem into two subproblems and the size of each subproblem is $n/2$. Hence, the recurrence relation is $T(n) = 2T(n/2) + O(n)$. So in total, the time complexity is $O(n \log n)$.

Correctness Proof: We will prove a stronger statement that on any input A our algorithm produces correctly the sorted version of A and the number of significant inversions in A. And,

we will prove it by induction.

Base Case: When $n = 1$, the algorithm returns 0, which is obviously correct.

Inductive Case: Suppose that the algorithm works correctly for $1, 2, \ldots, n-1$. We have to show that the algorithm also works for n.

By the induction hypothesis, B, x, C, y are correct. We need to show that the counting of significant split inversions and the normal merge-sort process are both correct. In other words, we are to show that z is the correct number of significant split inversions, and that D is correctly sorted.

First, we prove that z is correct. Due to the fact that B and C are both sorted by the induction hypothesis, for every $C[j]$, where $j = 1, \ldots, length(C)$, we can always find the smallest i such that $B[i] > 2C[j]$. Therefore, $B[1], B[2], \ldots, B[i-1]$ are all smaller than or equal to $2C[j]$, and $B[i], B[i+1], \ldots, B[length(B)]$ are all greater than $2C[j]$. Thus, $length(B) - i + 1$ is the correct significant split inversion count for pairs involving $C[j]$. After counting the number of correct significant split inversions for all $C[j]$, where $j = 1, \ldots, length(C)$, z is the correct number of significant split inversions.

Second, we show that D is the sorted version of A. This is actually the proof of merge-sort. Each time, we append one element from B or C to D. At each iteration k, since B and C are both sorted by the induction hypothesis, $\min(B[i], C[j])$, the one to be appended to D, is also the smallest number among all the numbers that have not been added to D yet. Consequently, every time we append the smallest remaining element (of B and C) to D, which makes D a sorted array.