CS 38 Introduction to Algorithms **Spring 2014**

Solution Set 3

Posted: April 24

If you have not yet turned in the Problem Set, you should not consult these solutions.

1. We INSERT n elements; at this point the root list contains these n singleton nodes (and their key values do not matter to the argument below). Now, we perform an EXTRACT-MIN. This removes the node with the minimum key value and begins the consolidation process from left to right. Let T_0 be the rooted tree with a singleton node; let T_i be the rooted tree formed by linking T_{i-1} to another T_{i-1}'s root. Note that T_i has 2^i nodes and rank i.

   We claim that after processing 2^i singleton nodes from left the right, these nodes are in a single heap-ordered tree with the shape T_i, provided the only other trees in the root-list are of shape T_j for $j > i$. The proof is by induction on i. Clearly it holds for $i = 0$. By induction, it holds for $i - 1$ so after processing the first half of the nodes, they all belong to a single tree with shape T_{i-1}. The next half of the nodes are all singletons, and so applying the induction hypothesis to them, we find that we don't form a tree with rank at least $i - 1$ until the last one is processed. Immediately after processing that node, we have created the second rank $i - 1$ tree (of shape T_{i-1}), so we link it to the first, and this produces the T_i-shaped tree we claimed. We conclude that after adding n nodes, there is a single tree in the root-list of shape T_k and then a final EXTRACT-MIN at this point has cost at least the number of children of the root of T_k, which is the rank, which is $k = \log_2 n$.

2. Our recursive algorithm returns the count together with a sorted list.

   We split the list into two sublists of $n/2$ elements each, and count the inversions recursively in each half (the base case, of two elements, is easy to handle in constant time). When the two recursive calls return, we have two sorted lists, one for each hald.

   Now we must add the inversions that straddle the boundary. These are the a_i, a_j pairs with $i \in \{1, 2, \ldots, n/2\}$ and $j \in \{n/2+1, \ldots, n\}$ and $a_i > a_j$. We process the two lists as we would for mergesort. If the two elements we are currently comparing are a_i and a_j and $a_i \leq a_j$, then we simple advance the pointer on the smaller one. If the two elements we are comparing are a_i and a_j and we have $a_i > a_j$, then we have as many inversions as there are elements after a_i in that sorted list (since all of those are greater than a_j and they all lie in the first list, while a_j lies in the second list). We continue in this fashion until we have scanned through both lists (exactly in the order we would have for the merge phase of mergesort). As we go we copy the elements in order so that we can indeed return a merged sorted list, in addition to the total number of inversions crossing the boundary.

   The running time recurrence is $T(2) = O(1)$ and $T(n) = 2T(n/2) + O(n)$ so the overall running time is $O(n \log n)$.

3. Let us call the matrix whose rows and columns are indexed by n bit strings and whose a, b entry is $(-1)^{\sum_i a_i b_i}$, M_n.

If we order the rows in lexicographic order (so the first half are labeled with i that begins with 0, and the second half are labeled with i that begins with 1), and the same for the columns, we find that the lower right sub-matrix of size $N/2 \times N/2$ is simply $-1M_{n-1}$ while the other three submatrices are M_{n-1}. Thus we can compute the matrix-vector product $M_n x$ by computing $u = M_{n-1}x_1$ and $v = M_{n-1}x_2$, where x_1 and x_2 are the first and second halves of the vector x. Then the result vector has as its first half $u + v$ and as its second half $u - v$.

The running time recurrence is $T(1) = O(1)$ and $T(N) = 2T(N/2) + O(N)$ so the overall running time is $O(N \log N)$ as required.

4. We show how to multiply a vector b by a Toeplitz matrix in $O(n \log n)$ time and then compute the full matrix-matrix product by doing this n times.

   Consider the $n \times n$ Toeplitz matrix M described by $a_0, a_1, \ldots, a_{n-1}, a_n, \ldots, a_{n-2}$, and define the polynomial $A(X) = \sum_i a_i X^i$. Define the polynomial $B(X) = \sum_i b_i X^i$.

   Observe that the first entry of the product Mb is

   $$a_{n-1}b_{n-1} + a_n b_{n-2} + \ldots + a_{2n-2}b_0$$

   and this is exactly the coefficient on X^{2n-2} in the product polynomial $A(X)B(X)$. Similarly, the second entry of the product Mb is

   $$a_{n-2}b_{n_1} + a_{n-1}b_{n-2} + \ldots + a_{2n-3}b_0$$

   and this is the coefficient on X^{2n-3} in the product polynomial. And so on. Thus we can read off the product Mb from the coefficients of the product polynomials $A(X)B(X)$. This gives the promised $O(n \log n)$ algorithm for computing the product of a Toeplitz matrix with a vector.

5. (a) Let $A_{i,j}(X)$ be the polynomial $\prod_{k=i}^{j}(X - a_k)$. We compute $f(X) \bmod A_{0,n/2-1}(X)$ and $f(X) \bmod A_{n/2,n-1}(X)$ which yield two polynomials $f_1(X)$ and $f_2(X)$, both of degree $n/2 - 1$. Note that by the second property in the hint $f_1(X) \bmod (X - a_i) = f(X) \bmod (X - a_i)$, for $i \in \{0, \ldots, n/2 - 1\}$. And by the first property in the hint this value is $f(a_i)$. A similar statement holds with respect to f_i and $i \in \{n/2, \ldots, n - 1\}$. We recursively solve these two problems of half the size, and report the results, which are the evaluations of $f(X)$ and $a_0, a_1, \ldots, a_{n-1}$ as required. The base case is when $n = 1$ and we just return $f(a_0)$ in constant time.

   We just need to have the interval polynomials $A_{ij}(X)$ available for the various recursive calls. We can do this bottom-up by computing $A_{i,i}(X)$ for all i, and then $A_{i,i+1}$ for even i, and then products of pairs of these polynomials, and then products of pairs of the next ones, and so on. The overall running time can be bounded as follows. At each level i (with $i = 1$ being the bottom layer), there are $n/2^i$ products of polynomials being computed, each of degree 2^{i-1}. Thus at each layer the time spent is $O(n/2^i) \cdot O(2^{i-1}(i-1)) = O(in)$, and i ranges from 1 to $\log n$. The overall running time for this part is thus $O(n \log^2 n)$. Then the recursive procedure above satisfies the recurrence: $T(1) = 0(1)$ and $T(n) = 2T(n/2) + O(n \log n)$ (the $O(n \log n)$ accounts for the two remainder computations before the recursive calls.

   Overall, we have running time $O(n \log^2 n)$ as the solution to this recurrence (we saw that in class), and also this is the running time of the overall solution.

(b) As suggested we compute the polynomial $f(X) = \prod_{i=0}^{L=\sqrt{M}-1}(X-i)$ by repeated polynomial multiplication: first we compute $L/2$ products of pairs of linear factors, then $L/4$ products of pairs of these degree 2 polynomials, then $L/8$ products of these degree 4 polynomials, etc... The overall running time is $L/2^{i+1} \cdot O(2^i i)$ at level i, and i ranges from 0 to $O(\log L)$. The overall running time is thus at most $O(L \log^2 L) = O(\sqrt{M} \log^2 M)$. Now, we use the previous part to evaluate $f(X)$ at the values $\sqrt{M}, 2\sqrt{M}, \ldots, \sqrt{M} \cdot \sqrt{M}$. Note that the evaluation at $i\sqrt{M}$ equals

$$\prod_{j=(i-1)\sqrt{M}+1}^{i\sqrt{M}} j \pmod{N}.$$

Now we take the product of all of these evaluations to get $\prod_{j=1}^{M} j \pmod{N} = M! \pmod{N}$. The running time is $O(\sqrt{M} \log^2 M)$ when we apply the previous part, and an additional $(\sqrt{M})$ multiplications for an overall $O(\sqrt{M} \log^2 M)$ running time as required.

(c) All we need to do is take the GCD of the integer $M!$ mod N that we computed in the previous part, and N itself. If N has a factor other than 1 that is at most M, then this will be a common factor of $M!$ and N other than 1, so the GCD will return an answer other than 1 (using the fact that $\text{GCD}(x, y) = \text{GCD}(x \bmod y, y)$. Otherwise, if N's only factor that are at most M is 1, then the GCD will return 1, using the same fact.