Massachusetts Institute of Technology | Handout 9
6.046J/18.410J: Introduction to Algorithms | February 27, 2003
Professors Piotr Indyk and Bruce Tidor

Problem Set 2 Solutions

Problem 2-1. Matching Nuts and Bolts

(a) The problem statement does not describe exactly how the nuts and bolts are specified, so we first need come up with a reasonable representation for the input. Let's assume the nuts and bolts are provided to us in two arrays $N[1 \ldots n]$ and $B[1 \ldots n]$, where we are allowed to compare elements across, but not within, these two arrays.

We use a divide-and-conquer algorithm very similar to randomized quicksort. The algorithm first performs a partition operation as follows: pick a random nut $N[i]$. Using this nut, rearrange the array of bolts into three groups of elements: first the bolts smaller than $N[i]$, then the bolt that matches $N[i]$, and finally the bolts larger than $N[i]$. Next, using the bolt that matches $N[i]$ we perform a similar partition of the array of nuts. This pair of partitioning operations can easily implemented in $\Theta(n)$ time, and it leaves the nuts and bolts nicely partitioned so that the "pivot" nut and bolt are aligned with each-other and all other nuts and bolts are on the correct side of these pivots — smaller nuts and bolts precede the pivots, and larger nuts and bolts follow the pivots. Our algorithm then finishes by recursively applying itself to the subarrays to the left and right of the pivot position to match these remaining nuts and bolts. We can assume by induction on n that these recursive calls will properly match the remaining bolts.

To analyse the running time of our algorithm, we can use the same analysis as that of randomized quicksort. We are performing a partition operation in $\Theta(n)$ time that splits our problem into two subproblems whose sizes are randomly distributed exactly as would be the subproblems resulting from a partition in randomized quicksort. Therefore, applying the analysis from quicksort, the expected running time of our algorithm is $\Theta(n \log n)$.

Interesting side note: Although devising an efficient randomized algorithm for this problem is not too difficult, it appears to be very difficult to come up with a deterministic algorithm with running time better than the trivial bound of $O(n^2)$. This remained an open research question until the mid-to-late 90's, when a *very* complicated deterministic algorithm with $\Theta(n \log n)$ running time was finally discovered. This problem provides a striking example of how randomization can help simplify the task of algorithm design.

(b) Let's use a proof based on decision trees, as we did for comparison-based sorting. Note that we can model any algorithm for matching nuts and bolts as a decision tree. The tree will be a ternary tree, since every comparison has three possible outcomes: less than, equal, or greater than. The height of such a tree corresponds to the worst-case number of comparisons made by the algorithm it represents, which in turn is a lower bound on the running time of that algorithm. We therefore want a lower bound of $\Omega(n \log n)$ on the height, H, of any decision tree that solves part (a). To begin with, note that the number of leaves L in any ternary tree must satisfy

$$L \leq 3^H.$$

Next, consider the following class of inputs. Let the input array of nuts N be fixed and consist of n nuts in increasing sorted order, and consider one potential input for every permutation of the bolts. In order to match the nuts and bolts, our algorithm must in this case essentially sort the array of bolts. In our decision tree, if two different inputs of this type were mapped to the same leaf node, our algorithm would attempt to apply to both of these the same permutation of bolts with respect to nuts, and it follows that the algorithm could not compute a matching correctly for both of these inputs. Therefore, we must map every one of these $n!$ different inputs to a distinct leaf node, so

$$\begin{aligned} L &\geq n! \\ 3^H &\geq n! \\ H &\geq \log_3 n! \\ H &= \Omega(n \log n) \quad \text{[Using Stirling's approximation]} \end{aligned}$$

Problem 2-2. Counting Inversions

(a) The five inversions are $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$, $(4, 5)$.

(b) If an n-element set is sorted in reverse order, then each of the $\binom{n}{2}$ pairs of elements will be an inversion.

(c) Insertion sort runs in $\Theta(n+I)$ time, where I denotes the number of inversion initially present in the array being sorted. To see this, consider the pseudocode for insertion sort:

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       i ← j − 1
       while i > 0 and A[i] > A[i + 1]
           do A[i + 1] ← A[i]
       A[i + 1] ← key
```

Everything except the while loop requires $\Theta(n)$ time. We now observe that every iteration of the while loop can be thought of as swapping an adjacent pair of out-of-order elements $A[i]$ and $A[i+1]$. Such a swap decreases the number of inversions in A by exactly one since $(i, i+1)$ will no longer be an inversion and the other inversions are not affected. Since there is no other means of increasing or decreasing the number of inversions of A, we see that the total number of iterations of the while loop over the entire course of the algorithm must be equal to I.

(d) To count the number of inversions in an array A, we modify merge sort so that it counts inversions as it sorts. Let L denote the lower half of the array $A[1 \ldots \lfloor n/2 \rfloor]$ and let R denote the upper half of the array $A[\lfloor n/2 \rfloor + 1 \ldots n]$. Every inversion in A will be one of three types: we'll call an inversion a *left inversion* if it involves two elements from L, a *right inversion* if it involves two elements from R, and a a *crossover inversion* if it involves one element from L and one element from R.

Our algorithm begins by recursively applying itself to L and then to R. This results in L and R being sorted, and we can also assume by induction that these recursive calls will properly count the number of left and right inversions. The only remaining task is to count the crossover inversions. These inversions are not affected when we sort L and R, so we will count them during the process of merging the sorted contents of L and R together. This is done as follows. Every time we add an element from L to the merged array, we count the number of elements of R with which it will form an inversion. More precisely, suppose that we're comparing the ith element of L to the jth element of R, and that $L[i] \leq R[j]$, so $L[i]$ is the next element to be added to the merged list. In this case, $L[i]$ will be in an inversion with each of the elements in $R[1 \ldots j-1]$, so we add $j-1$ to our running total inversion count. This will in fact count all crossover inversions, since every such inversion will be counted when its left element is added to the merged array.

The extra counting doesn't affect the asymptotic running time of merge sort. Since we are doing only a small constant amount of extra work per element each iteration of the merge step, merging still takes $\Theta(n)$ time. The running time of our algorithm therefore satisfies the recurrence $T(n) = 2T(n/2) + \Theta(n)$, so $T(n) = \Theta(n \log n)$.

For further thought: We haven't proved a lower bound on the running time of a comparison-based algorithm for counting inversions. Apparently, whether or not inversion counting can be performed more efficiently, or if there exists a matching $\Omega(n \log n)$ lower bound, remains an open research question.

Problem 2-3. Matrix Vector Products

(a) A single matrix-vector product, $\mathbf{Ax}$, takes $O(n^2)$ time and since we repeat the operation m times, the overall running time is $O(mn^2)$.

(b) Matrix multiplication being associative, the result can be computed by first evaluating $\mathbf{A}^m$ and then performing the matrix-vector product, $\mathbf{A}^m\mathbf{x}$. In the lecture on divide-and-conquer, we saw that exponentiation can be done in $O(\log m)$ multiplications. Using Strassen's $O(n^{2.81})$ algorithm for matrix multiplication, we get an overall running time of $O(n^{2.81} \log m + n^2) = O(n^{2.81} \log m)$.

This method will be faster than the previous when $m/\log m = \Omega(n^{0.81})$. Getting a closed form solution for an equation of the form $m/\log m = \Omega(f(n))$ is somewhat non-trivial but by guessing a solution of the form $m = \Omega(f(n) \log f(n))$, it's not too hard to prove it's correctness (left as an exercise). In this case, the solution translates to $m = \Omega(n^{0.81} \log n)$.

Problem 2-4. Almost Sorted!

(a) Recall from the solution to problem 2(c) that the running time of insertion sort can be written as $\Theta(n+I)$, where I denotes the number of inversions in the input array. Therefore, if we can establish for our special class of inputs that $I = O(kn)$, then this will imply that the running time of insertion sort will be $O(kn)$.

How can we bound I? If we focus on any particular element in our array, say $A[i]$, let us consider which other elements in the array might potentially form an inversion with this element. Due to the restriction that every element may be at most k positions away from its position in the final sorted ordering, we can deduce that any element forming an inversion with $A[i]$ must lie somewhere in the range $A[i-2k]$ to $A[i+2k]$. Every element therefore belongs to at most $O(k)$ inversions, so we conclude that $I = O(kn)$.

(b) For this part, we use a decision tree argument similar to the one used in Problem 1. The only difference in this case is the number of leaves in the decision tree. Recall that the leaves of the decision tree correspond to different allowable permutations of the input array. Thus our problem reduces to finding a lower bound on the number of possible permutations, given that any element may move at most k positions away from it's true position.

In order to arrive at a lower bound, we divide up the array into n/k sub-arrays, each of size k. Since all $k!$ permutations within each sub-array are "legal", we get the following lower bound on the number of permutations (leaves).

$$\#\text{ leaves} \geq (k!)^{n/k}$$

Since the worst case running time of any sorting algorithm is the height of it's decision tree, we have:

$$\begin{aligned}
\text{Running time} &\geq \log(\#\text{ leaves}) \\
&= \log(k!)^{n/k} \\
&= n/k \log k! \\
&= \Omega(n \log k) \quad \text{[Using Stirling's approximation]}
\end{aligned}$$