

15-451 Algorithms, Fall 2008

Homework # 2

Solutions

Problems:

1. [median of two sorted arrays] Let A and B be two sorted arrays of n elements each. We can easily find the median element in A — it is just the element in the middle — and similarly we can easily find the median element in B . (Let us define the median of $2k$ elements as the element that is greater than $k - 1$ elements and less than k elements.) However, suppose we want to find the median element overall — i.e., the n th smallest in the *union* of A and B . How quickly can we do that? You may assume there are no duplicate elements.

Your job is to give tight upper and lower bounds for this problem. Specifically, for some function $f(n)$,

- (a) Give an algorithm whose running time (measured in terms of number of comparisons) is $O(f(n))$, and
- (b) Give a lower bound showing that any comparison-based algorithm must make $\Omega(f(n))$ comparisons in the worst case.

In fact, see if you can get rid of the O and Ω to make your bounds *exactly* tight in terms of the number of comparisons needed for this problem.

Some hints: You may wish to try small cases. For the lower bound, you should think of the output of the algorithm as being the location of the desired element (e.g., “ $A[17]$ ”) rather than the element itself. How many different possible outputs are there?

Solution: The upper and lower bounds are exactly $\lceil \log_2 2n \rceil$. For brevity, however, we will assume here that n is a power of 2. Also, to make the math simpler, let's index the arrays beginning with 1.

- (a) Compare $A[n/2]$ to $B[n/2]$. Say that $A[n/2] < B[n/2]$. Notice that the only elements that can possibly be smaller than $A[n/2]$ are $A[1] \dots A[n/2 - 1]$ and $B[1] \dots B[n/2 - 1]$ (which is a total of just $n - 2$ elements). So, $A[n/2]$ is at most the $(n - 1)$ st smallest overall. This means we can delete $A[1] \dots A[n/2]$ as possible answers. Also, $B[n/2]$ must be at least the n th smallest overall since $A[1] \dots A[n/2]$ and $B[1] \dots B[n/2 - 1]$ are smaller than it. This means that nothing larger than it could possibly be the right answer, so we can delete $B[n/2 + 1] \dots B[n]$. So, we recursively find the $(n/2)$ th smallest element in the union of the arrays $A' = A[n/2 + 1] \dots A[n]$ and $B' = B[1] \dots B[n/2]$. We used one comparison to cut the problem size in half. As a base case, when $n = 1$, we take one final comparison to find the element we want. So this solves to $T(n) = 1 + \lg n = \lg(2n)$.
- (b) For the lower bound, notice that any of $A[1] \dots A[n]$ or $B[1] \dots B[n]$ could be answers. For instance $A[i]$ could be the answer if the lowest $n - i$ elements of B are less than it, and the remaining elements in B are greater than it. So, consider $2n$ pairs of arrays (A, B) ,

each requiring a different answer, and think of these as $2n$ “scenarios”. Now, consider the decision tree where each node has associated with it all the scenarios consistent with the results of comparisons made so far. This tree must have $2n$ leaves, since if two different scenarios are consistent with all the comparisons so far, the algorithm cannot yet be done (since each scenario requires a different answer). This means the depth must be at least $\lceil \log_2 2n \rceil$.

Another way to do the argument: if the algorithm makes at most t comparisons, then there are at most 2^t possible outputs it can produce (since there are at most 2^t possible series of answers possible to the comparisons made). So, if $2^t < 2n$ then there must be some location never output by the algorithm, which contradicts the fact that there exist inputs for which each of the $2n$ locations are the correct answer.

2. [tight upper/lower bounds] Consider the following problem.

INPUT: n^2 distinct numbers in some arbitrary order.

OUTPUT: an $n \times n$ matrix of the inputs having all rows and columns sorted in increasing order.

EXAMPLE: $n = 3$, so $n^2 = 9$. Say the 9 numbers are the digits 1, ..., 9. Possible outputs include:

1 4 7	or	1 4 5	or	1 3 4	or	...
2 5 8		2 6 7		2 5 8		
3 6 9		3 8 9		6 7 9		

It is clear that we can solve this problem in time $O(n^2 \log n)$ by just sorting the input (remember that $\log n^2 = O(\log n)$) and then outputting the first n elements as the first row, the next n elements as the second row, and so on. Your job in this problem is to prove a matching $\Omega(n^2 \log n)$ *lower bound* in the comparison-based model of computation.

For simplicity, you can assume n is a power of 2.

Some hints: Show that if you could solve this problem using $o(n^2 \log n)$ comparisons (in fact, in less than $n^2 \lg(n/e)$ comparisons), then you could use this to violate the $\lg(m!)$ lower bound for comparisons needed to sort m elements. You may want to use the fact that $m! > (m/e)^m$. Also, recall that you can merge two sorted arrays of size n using at most $2n - 1$ comparisons.

Solution: Let’s assume we are able to solve the “Matrix sorting” problem using less than $n^2 \lg(n/e)$ comparisons. We will show that this would allow us to sort n^2 elements in less than $\lg((n^2)!)$ comparisons, violating our sorting lower bound.

Here is how we would do the sorting. We begin by putting the n^2 elements into an n -by- n matrix and running our fast matrix-sorting algorithm. We then repeatedly merge the rows of the matrix as follows. First, we pair up the rows, merging rows $2i - 1$ and $2i$ ($i = 1, \dots, \frac{n}{2}$), thereby obtaining a new, $\frac{n}{2} \times 2n$ matrix (which will obviously have its rows sorted). Then, we pair up the rows of this matrix, merging rows $2i - 1$ and $2i$ ($i = 1, \dots, \frac{n}{4}$), and obtaining a new matrix of size $\frac{n}{4} \times 4n$ (which, again, will have its rows sorted). We repeat the procedure until

we finally merge the two rows of a $2 \times \frac{n^2}{2}$ matrix and get the sorted sequence of n^2 numbers. (We are using the fact that n is a power of 2, so in every step we can pair up the rows.)

The total number of comparisons this algorithm uses for merging is at most:

$$\frac{n}{2}(2n - 1) + \frac{n}{4}(4n - 1) + \frac{n}{8}(8n - 1) + \dots + 2\left(\frac{n^2}{2} - 1\right) < n^2 \lg n.$$

Adding the above to the $n^2 \lg(n/e)$ comparisons we supposedly used for the matrix-sorting problem, we get a total of less than $n^2 \lg(n^2/e) = \lg((n^2/e)^{n^2}) < \lg((n^2)!) comparisons, violating our sorting lower bound.$

3. [amortized analysis] Suppose we have a binary counter such that the cost to increment or decrement the counter is equal to the number of bits that need to be flipped. We saw in class that if the counter begins at 0, and we perform n increments, the amortized cost per increment is just $O(1)$. Equivalently, the total cost to perform all n increments is $O(n)$.

Suppose that we want to be able to both increment *and* decrement the counter.

- (a) Show that even without making the counter go negative, it is possible for a sequence of n operations starting from 0, allowing both increments and decrements, to cost as much as $\Omega(\log n)$ amortized per operation (i.e., $\Omega(n \log n)$ total cost).

Solution: Let $2^{b+1} \leq n \leq 2^{b+2}$. Use 2^b operations to transform the number to a 1 followed by $b - 1$ 0's. This takes at least 2^b work. Then alternate decrementing and incrementing the number for the remaining $\geq 2^b$ steps. Each of these operations takes $(b - 1)$ work. So, the total work over all these operations is at least $b(2^b)$.

Thus the amortized cost per operation is at least $b2^b/n \geq b/4 = \Omega(\log n)$.

- (b) To reduce the cost observed in part (a) we'll consider the following *redundant ternary number system*. A number is represented by a sequence of *trits*, each of which is 0, +1, or -1. The value of the number represented by t_{k-1}, \dots, t_0 (where each $t_i, 0 \leq i \leq k - 1$ is a trit) is defined to be

$$\sum_{i=0}^{k-1} t_i 2^i.$$

For example, $\boxed{1} \boxed{0} \boxed{-1}$ is a representation for $2^2 - 2^0 = 3$.

The process of incrementing a ternary number is analogous to that operation on binary numbers. You add 1 to the low order trit. If the result is 2, then it is changed to 0, and a carry is propagated to the next trit. This process is repeated until no carry results. Decrementing a number is similar. You subtract 1 from the low order trit. If it becomes -2 then it is replaced by 0, and a borrow is propagated. Note that the same number may have multiple representations (e.g., $\boxed{1} \boxed{0} \boxed{1} = \boxed{1} \boxed{1} \boxed{-1}$). That's why this is called a *redundant* ternary number system.

The cost of an increment or a decrement is the number of trits that change in the process. Starting from 0, a sequence of n increments and decrements is done. Give a clear, coherent proof that with this representation, the amortized cost per operation

is $O(1)$ (i.e., the total cost for the n operations is $O(n)$). Hint: think about a “bank account” or “potential function” argument.

Solution: Note that when the number is incremented or decremented, some 1s may change to 0 and some -1 s may change to 0, but a 1 will never directly change to -1 or vice versa. This means that if \$ 2 is paid each time a 0 is changed (to 1 or -1), there will be enough money in the bank to pay for converting it back to a 0. (Think of having a separate bank account for each digit.) The second fact is that in any increment or decrement, at most one 0 changes (to either 1 or -1). Therefore, the amortized cost per operation is at most 2.