15-451 Algorithms, Fall 2004

Homework # 2 **due: Tue-Wed, Sept 28-29, 2004**

Groundrules:

- This is an oral presentation assignment. As stated in the course information handout, you should work in groups of three. At some point before **Sunday Sept 26 at midnight** you should sign up for a 1-hour time slot on the signup sheet on the course web page.
- You are not required to hand anything in at your presentation, but you may if you choose. If you do hand something in, it will be taken into consideration (in a non-negative way) in the grading.

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. (If n is even, let us define the median as the element just to the left of the middle.) However, suppose we want to find the median element overall — i.e., the nth smallest in the *union* of A and B. How quickly can we do that? For purposes of defining what we mean, you may assume there are no duplicate elements.

   Your job is to give tight upper and lower bounds for this problem.[1] 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, your 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?

2. [lower bounds for deterministic quicksort] We saw in class that if we always choose the leftmost element as the pivot for quicksort then it will take time $\Omega(n^2)$ if the input is already sorted.[2]

---

[1] Historical note: this fact — the matching upper and lower bounds — was something that led the [BFPRT] authors to investigate and solve the linear-time median-finding problem discussed in class.

[2] Technical detail: assume that the partitioning (rearranging) procedure doesn't change the relative order of elements that go into the same bucket. E.g., if the original array was $[3, 5, 0, 4, 2, 6, 1]$, and we use the leftmost element as pivot, then we will get $[0, 2, 1, 3, 5, 4, 6]$.

Your goal in this problem is to show that for *any* deterministic pivot-selection procedure, so long as the pivot is chosen by examining only a constant number of elements, you can reverse-engineer an input that makes quicksort take $\Omega(n^2)$ time.

(a) Show how to construct an array of n elements (for any given value of n) such that the strategy "pick the middle element $A[n/2]$ as pivot" will have $\Omega(n^2)$ running time.

(b) Now show how we can do the same thing for any pivot-selection strategy S that examines only a constant number of elements. Specifically, given the length n of the array, S chooses an index i_1 to examine. Based on what it sees ($A[i_1]$), it then chooses a new index i_2 to examine. Based on what it sees there, it then chooses a new index i_3 and so on up to i_c where c is some constant (like 5). S then chooses one of $A[i_1], A[i_2], \ldots, A[i_c]$ as the pivot. Show how given access to S you can reverse-engineer an n-element array such that quicksort using S takes time $\Omega(n^2)$.

3. [amortized analysis] Suppose we have a binary counter such that the cost to increment 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 increment the counter n times, 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).

(b) To fix the problem from part (a), 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 $t_{k-1}, \ldots, t_0$ (where each t_i is a trit) is defined to be

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

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 are 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.