15-451 Algorithms, Spring 2007

Homework # 2 **due: Week February 12 Mon-Thurs, 2007**

Ground rules:

- This is an oral presentation assignment. You should work in groups of three. At some point before **Sunday, February 11 at 11:59pm** your group should sign up for a 1-hour time slot on the sign-up sheet on the course web page. Sign-ups will close at that time.
- Each person in the group must be able to present every problem. The TA/Professor will select which student presents which problem. The other group members may assist the presenter.
- 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. (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 nth 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?

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 *either* all rows, *or* all 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          1 4 5          3 2 1
2 5 8    or    3 6 7    or    4 6 5    or   ...
3 6 9          2 8 9          8 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 $n^2 \lg(n/2e)$ 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.

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).

   (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 $t_{k-1}, \ldots, t_0$ (where each $t_i, 0 \le i \le k-1$ is a trit) is defined to be

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

   For example, $\boxed{1}\boxed{0}\boxed{\text{-1}}$ 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., $\boxed{1}\boxed{0}\boxed{1} = \boxed{1}\boxed{1}\boxed{\text{-1}}$). 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.