Illinois Institute of Technology
Department of Computer Science

Solutions to First Examination

CS 430 Introduction to Algorithms
Spring, 2012

11:25am–12:40pm, Wednesday, February 8, 2012
104 Stuart Building

Exam Statistics

56 students took the exam. The range of scores was 15–92, with a mean of 53.73, a median of 54, and a standard deviation of 16.26. Very roughly speaking, if I had to assign final grades on the basis of this exam only, above 70 would be an A (10), 55–69 a B (17), 41–54 a C (19), 20–40 a D (7), below 20 an E (3).

Problem Solutions

1. (a) Every time; that is, n times.

   (b) $1/n$ because each of the n indices is equally likely to be the last in the random order, including the index at which the maximum element occurs.

   (c) From the previous part, the expected number of executions is given by $e_n = e_{n-1} + 1/n$, $e_1 = 1$. Thus e_n is the nth harmonic number $H_n = 1 + 1/2 + \cdots + 1/n = \ln n + o(1)$.

2. A correct algorithm must report the majority color, but how can you do that using only comparisons $x_i : x_j$? In other words, you cannot look at an element and see that it is red or blue, only that its color matches or does not match the color of another element. Thus you must return an element of the majority color—there is no other way to report the majority! Many students tried to report the majority as "the other color" or something equivalent; this was not acceptable. Furthermore, no data movement is necessary or desirable.

   Here is a simple algorithm that always uses $n - \nu(n)$ questions, where $\nu(n)$ denotes the number of 1-bits in the binary representation of n: Initially, put each of the n elements in a different bin; then, at each step, select one representative from each of two equal-sized bins and compare their colors. If they are the same color, merge these two bins; otherwise discard both bins (as per the hint, this does not affect the majority color, if any). The sizes of the bins are always powers of 2, so if there are no two bins of equal size, the largest bin must contain more elements than the rest of the bins combined—any element from that bin is of the majority color. If there are no bins left, there was no majority. In the worst case, all comparisons result in identical colors and the n elements will be distributed into bins of unequal size, all powers of 2; this is just the binary representation of n, so there are $\nu(n)$ bins. The number of tests that went into the formation of a bin of size 2^i is clearly $2^i - 1$,

and the total of $n - \nu(n)$ comparisons follows. It turns out that this algorithm is *optimal* by an information theoretic argument; see "Determining the majority," *Info. Proc. Let.*, vol. 47 (1993), pp. 253–255.

A slightly inferior method (also more subtle, but easier to write in code) was proposed in "A Fast Majority Vote Algorithm," *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Kluwer, 1991, pp. 105–117:

```
 1: c ← 0
 2: for i ← 1 to n do
 3:    if c = 0 there are equal numbers of red/blue items among x_1, ..., x_{i-1};
 4:    otherwise there are c more of the color of x_j than of the other color
 5:    if c = 0 then
 6:       j ← i
 7:       c ← 1
 8:    else if x_i = x_j then
 9:       c ← c + 1
10:    else
11:       c ← c − 1
12:    end if
13: end for
14: if c = 0 then
15:    report there is no majority
16: else
17:    report the majority color is the color of x_j
18: end if
```

This makes as few as $n/2$ color comparisons in line 8 (for what input?) or as many as $n - 1$ (again, for what input?).

An even worse linear-time algorithm is to compare x_1 with each of the remaining $n - 1$ elements, counting the number that are equal. This always uses $n - 1$ comparisons and gives us enough information to say whether x_1 is of the majority or not, or if there is a tie (no majority). To report the majority color when it is *not* x_1, you need to have kept track of an element not matching x_1.

3. The recurrence for the worst case of Lucky Quicksort is

$$T(n) = T(n/4) + T(3n/4) + cn,$$

for some constant c and n sufficiently large, say $n \geq n_0$. $T(n) = O(1)$ for $n < n_0$. This is the fourth example on page 13 of the notes on recurrence relations (Lecture 2, January 11); you could cite the argument given there based on recursion trees or argue directly as follows:

We show that for K large enough, $T(n) \leq Kn \log n + cn$ for $n \geq n_0$ by induction. The statement clearly holds of for $n < n_0$. By induction, then,

$$T(n) \quad \leq \quad [K(n/4)\log(n/4) + cn/4] + [K(3n/4)\log(3n/4) + 3cn/4] + cn$$

$$\begin{aligned} &= K(n/4)(\log n - \log 4) + K(3n/4)(\log n - \log 4/3) + 2cn \\ &= Kn\log n - Kn[(3/4)(\log 4/3) + (1/4)\log 4] + 2cn. \end{aligned}$$

Thus we choose K such that

$$-K[(3/4)(\log 4/3) + (1/4)\log 4] + 2c \leq c,$$

or

$$K \geq \frac{c}{(3/4)(\log 4/3) + (1/4)\log 4}.$$

4. (This is problem 12.1-5 on page 289.)

   Given a binary search tree for n elements, an inorder traversal, which takes $O(n)$ time, produces the sorted list of elements without using any comparisons. This means that once the binary search tree has been formed, the sorted order is completely known. Thus, having an $o(n \log n)$ algorithm for creating a binary search tree means that there is an $o(n \log n)$ algorithm for sorting, that is impossible since there is a lower bound of $\Omega(n \log n)$ for sorting.

   Note that the question did *not* ask for a direct proof of that lower bound, only for a way to use the bound for sorting for that particular problem.

5. (This is an expanded version of problem 13.1-7 page 312.)

   The smallest possible number of internal nodes is when all the nodes are black (the root may be red but it doesn't matter because the root is not counted (see the definition of a black weight of a tree). In that case the number of internal nodes is $2^k - 1$. The tree must be full because in a RB-tree every simple path from a node to a descendent leaf contains the same number of black nodes.

   The largest number of internal nodes is when the root is black and the colors of nodes in each path alternate, that is, each black node (which is not a leaf) has two red sons and each red node has two black sons. The number of nodes in that case is $2^{2k} - 1$.

   Starting with a red root would yield $2^{2k-1} - 1$ internal nodes.