Illinois Institute of Technology
Department of Computer Science

Solutions to First Examination

CS 430 Introduction to Algorithms
Spring, 2009

11:25am–12:40pm, Wednesday, February 23, 2009
104 Stuart Building

1. **Randomized Min**

   Consider the following algorithm to find the minimum element in an unordered array of n elements:

   ```
   1: RandomMin(A[1..n])
   2: m ← ∞
   3: for i ← 1 to n in random order do
   4:    if A[i] < m then
   5:       m ← A[i]
   6:    end if
   7: end for
   8: return m
   ```

   Notice that the **for** loop (line 3) takes the numbers $1, 2, \ldots, n$ in random order.

   (a) In the worst case, how many times does RandomMin execute line 5?

   (b) What is the probability that nth (last) iteration executes line 5?

   (c) Analyze the expected number of executions of line 5.

   Solution.

   (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 minimum 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. **Unusual Sorting Algorithm**

   A TA in CS 430 suggested the following unusual algorithm to sort n elements in an array:

   ```
   1: TaSort(A[0..n − 1])
   2: if n = 2 and A[0] > A[1] then
   3:    swap A[0] ↔ A[1]
   4: else
   ```

```
5:    if n > 2 then
6:       m ← ⌈2n/3⌉
7:       TaSort(A[0..m − 1])
8:       TaSort(A[n − m..n − 1])
9:       TaSort(A[0..m − 1])
10:   end if
11: end if
```

(a) Prove that TaSort correctly sorts its input.

(b) When the TA first proposed the algorithm, she mistyped line 6 using the floor instead of the ceiling operation. The algorithm failed to sort correctly; explain why.
(*Hint*: Consider $n = 4$.)

(c) State (using precise floor/ceiling operations) the recurrence relation and initial values describing the number of comparisons $A[0] > A[1]$ performed in line 2 of the algorithm in the worst case. Then, ignoring the floor/ceiling operations, solve the recurrence.

Solution.

(a) The first recursive call sorts the lower 2/3 of the array; the second then sorts the upper 2/3 of the array, leaving the upper 1/3 in its correct sorted order. That second recursive call messes up the bottom 2/3, however, so that portion needs to be resorted, which is done in the third recursive call. This is the idea behind an inductive proof, in which the base case is $n = 2$ (which is correctly handled by lines 2–3). The induction (and correctness!) depends on the lower 2/3 and the upper 2/3 of the array overlapping by $\lceil n/3 \rceil$.

(b) With floor instead of ceiling in line 6, there is no overlap for $n = 4$ elements: $m = 2$, so the recursive calls sort $A[0..1]$, $A[2..3]$, and then $A[0..1]$ again so the sort will fail if the elements in $A[0..1]$ are larger than those in $A[2..3]$.

(c) $T(2) = 2$, $T(n) = 3T(\lceil 2n/3 \rceil) + c$, for some constant c. Ignoring the ceiling this becomes $T(n) = 3T(2n/3) + c$. We can solve this by the "Master Theorem" in CLRS or by substitution and the operator method of the class notes: Substitute $n = 2(3/2)^k$ and let $t_k = T(n)$. This gives $t_0 = 2$ and $t_k = 3t_{k-1} + c$. The annihilator for that is $(E-3)(E-1)$ so the solution is $t_k = a3^k + b$ for some constants a and b. That is, $T(n) = t_k$ where $k = \log_{3/2}(n/2) = \log_{3/2} n + O(1)$, so that $T(n) = a3^{\log_{3/2} n} + O(1) = \Theta(n^{\log_{3/2} 3})$. Since $(3/2)^2 = 2.25$, $\log_{3/2} 3 > 2$, so this sorting algorithm is worse that an n^2-sort.

3. **Special Hardware Priority Queue**

Distressed by the results of the previous problem, the TA has developed a hardware priority queue for his computer. The priority queue device can store up to p records, each consisting of a key and a small amount of satellite data (such as a pointer). The computer to which it is attached can perform Insert and ExtractMin operations on the priority queue, each of which takes $O(1)$ time, no matter how many records are stored in the device. The TA wishes

to use the hardware priority queue to help implement a sorting algorithm on his computer. He has n records stored in the primary memory of his machine. If $n \leq p$, the TA can certainly sort the keys in $O(n)$ time by first inserting them into the priority queue, and then repeatedly extracting the minimum. Design an efficient algorithm for sorting $n > p$ items using the hardware priority queue. Analyze your algorithm in terms of both n and p.

Solution. A simple idea, worth almost full credit, is to divide the list input into n/p lists of length p, sort the lists using the queue in $n/p \times \Theta(p) = \Theta(n)$ time, and then merge the lists in $\Theta(n \lg(n/p))$ time. The total time for this algorithm is $\Theta(n \log(n/p))$.

But this approach uses the hardware only for sorting sub-lists; the hardware does not help with merging. A better idea is to use the hardware to implement a p-way merge, that runs in $\Theta(n)$ time (for p lists containing n elements total), and use it the same way merge sort uses its $\Theta(n)$ time 2-way merge.

```
1: SORT(A)
2: if A has length 1 then
3:    return
4: else
5:    Divide A into p equal sub arrays A_1 ... A_p
6:    SORT(A_1) ... SORT(A_p)
7:    p-way merge A_1 ... A_p
8: end if
```

The p-way-merge works as follows

(a) Start by moving the smallest element from each list into the hardware queue, along with an indicator as to which list it came from.

Time: p elements, $O(1)$ time to INSERT each, so $O(p)$ total time.

(b) EXTRACTMIN to get the smallest element. Say it came from the ith sorted list. Move the next element (if any) from the ith list into the queue, to replace the one that was just extracted. Repeat until all n elements have been extracted.

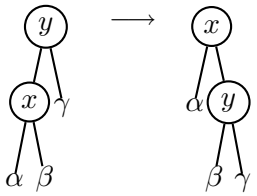
Time: $O(1)$ to EXTRACTMIN and to INSERT, so $O(n)$ total time.

The time of the p-way-merge is thus $O(p+n) = O(n)$. The overall running time is thus given by the recurrence $T(n) = pT(n/p) + O(n)$ which has solution $T(n) = O(n \log_p n)$. This time is asymptotically better than the $\Theta(n \log(n/p))$ of the simple approach, because we are not treating p as a constant.

4. **Augmented Binary Search Trees**

Suppose we augment a red-black tree so each node has a field giving the height of its subtree. Show that a rotation in such a tree can be done in time $O(1)$, correctly maintaining the height fields.

Solution. This follows from Theorem 14.1, but it is easy to show directly. We'll look only at a right rotation; the left rotation analysis is identical:



Before the rotation, the heights fields are correct in each node; moreover, we can get the heights of the subtrees α, β, and γ by following the corresponding pointer, if it is not empty—if it is empty, that height is zero. After the rotation, the height fields are all correct (because they did not change), except those in the nodes x and y. The height of the subtree rooted at y in the new subtree is $\text{height}(y) = 1 + \max(\text{height}(\beta), \text{height}(\gamma))$ and the height of the subtree rooted at x is then $\text{height}(x) = 1 + \max(\text{height}(\alpha), \text{height}(y))$.

5. **Maximum External Path Length in Binary Search Trees**

   Given a binary tree with n internal nodes and $n + 1$ leaves, what is the maximum possible external path length? Prove your answer.

   Solution. The maximum EPL is $1 + 2 + 3 + \cdots + n + n = n(n+1)/2 + n = (n^2 + 3n)/2$ for a tree that is has exactly one internal at each of depths 1, 2, ..., n (which means one leaf at each depth 1, 2, ..., $n - 1$ and two leaves at depth n). A proof by contradiction proves that this is the maximum possible: Consider the tree with the largest EPL and look at its longest path from the root to a leaf, say at depth L. If $L = n$ we are done because the tree is as described (that leaf must have a sibling leaf also). If $L < n$, some internal node on that path has non-empty sibling subtree S (otherwise, where could the remaining $n - L$ nodes not on that path be!); suppose that the root of S is at depth ℓ, $\ell < L$. Swap S with the deepest leaf (at depth L, by assumption); the effect of this change on the EPL is to add depth $L - \ell$ to each of the $|S| > 0$ leaves. Because $L - \ell > 0$, the EPL is thus increased by $|S|(L - \ell) > 0$, a contradiction.