15-451 Algorithms, Spring 1997
Midterm Exam Solutions

March 11, 1997

Problem 1

a) True. For $1 \leq i < n$, check if $A[i] \leq A[i+1]$ is true. An array is unsorted if and only if there exists some i for which the above comparison fails. Each comparison takes constant time and there are $n-1$ of them. Hence the total time taken is $O(n)$.

b) **(i)** True. $\log_2 n = \log_{10} n / \log_{10} 2$. Hence, the two functions differ by a multiplicative constant $(1/\log_{10} 2)$ which can be absorbed by the $O()$ notation.

(i) False. $n^{\log_2 n} = n^{\log_{10} n / \log_{10} 2} = (n^{\log_{10} n})^c$, where c is the constant $1/\log_{10} 2$. A multiplicative constant in the exponent cannot be absorbed by the $O()$ notation. Many had derived an incorrect expression $n^{\log_{10} n} n^c$ instead of the last expression above and got the right answer for the wrong reason.

c) False. An individual FIND can take $O(\log n)$ steps in the worst case, it is only the average over all the unions and finds that takes $O(\log^* n)$. Many had got this one wrong.

d) True. We can show that we can use this procedure to sort n elements. We do a pre-processing step in which we replicate each of the n elements 10 times and form n lists of 10 identical elements each. This takes $O(n)$ time. Then use the procedure to merge the n (trivially) sorted lists into one sorted list. In the post-processing step, startin from the first element and read off every tenth element which gives us the sorted list of the original n elements. This last step also takes $O(n)$ time. Since we know that every comparison-based sorting algorithm (no matter how clever) takes time that is bounded below by $\Omega(n \log n)$ in the worst case and the pre- and post-processing steps above take only $O(n)$ which is smaller than $\Omega(n \log n)$, the merging step must take at least $\Omega(n \log n)$ time.

Many had misunderstood the procedure as merging 10 lists of n elements each, which is $O(n)$. Others had presented an actual algorithm that does the merging and got an upper bound of $O(n \log n)$ which is not what we are after. You should show that ANY algorithm that implements the procedure has to have the stated lower bound.

e) True. We can do a binary search for x in the array in $O(\log n)$ time, and the same for y. Then take the difference of the two indices found to obtain the number of keys in between. This takes constant time, so the total is still $O(\log n)$.

More rigorous: Find indices $0 \leq i, j \leq n$ such that $A[i] < x \leq A[i+1]$, $A[j] \leq y < A[j+1]$, where we consider $A[0] = -\infty$, $A[n+1] = +\infty$. The required number is $j - i$.

f) True. Just return the maximum of $A[2]$ and $A[3]$.

More rigorous: For any element $A[i]$ in the heap, the path up to the root contains elements in nondecreasing order. For $i \geq 1$, this path will go through either $A[2]$ or $A[3]$, which is therefore at least as large as $A[i]$. The maximum of the two will therefore be at least as large as any of $A[2..n]$.

Problem 2

Since *all* nodes in an AVL tree have to be balanced, we can rephrase the definition recursively: a tree is an AVL tree if and only if its left and right subtrees are both AVL trees and their heights are at most one apart (empty trees are AVL). Our algorithm implements this definition: it descends the tree recursively starting from the root and returns for every node whether the subtree rooted at it is an AVL tree or not; if yes, it also returns the node's height. For an empty tree, return "yes" and a height of zero. If the tree is nonempty, compute the answer for its left and right subtrees. If any of them is not AVL, return "no", else compare their heights. If their absolute difference is more than 1, return "no", else return "yes" and the largest of the two heights plus 1 (to account for this node). The algorithm visits each node exactly once and performs a constant amount of work at each node. It is therefore $O(n)$.

The only way to find out whether each node is balanced is to visit its subtrees, and therefore the node itself. Visiting all nodes requires at least $\Omega(n)$ work, so no algorithm can do better.

Common Errors

- Making unnecessary assumptions about the data structure. Your algorithm should not depend (unless necessary) on the fact that trees are represented with arrays, pointers or something else. Even more important, you should *not* assume that the data structure already contains auxiliary fields that for some reason you have decided to use.
- Confusing height with size (number of nodes). The AVL property is about the *heights* of the trees, not about their sizes. (There is a relation between the two, shown in Theorem 4.1 in the book, but this is not directly relevant to the problem).
- Trying to establish a *global* property of the tree (specifically, that in an AVL tree any two leaves have depths in the tree that are at most 2 apart). This is false. To convince yourself, try to show the following (useful as an exercise about induction): for any $n \geq 1$ there is an AVL tree of height n that has leaves at any depth between $\lfloor n/2 \rfloor + 1$ and n. In general, since trees are recursive structures, it is useful attempt to find a simple recursive solution first.

Problem 3

The selection algorithm is a variant of quicksort designed to find just the k^{th} largest element of a set. This is a generalization of the median finding algorithm and is described in section 6.5.2 of Manber. The main difference between quicksort and selection is that in quicksort we recursively process both sides of the partition, but in selection we need only process one side. Although complete sorting requires $O(n \log n)$ time, we can solve the selection problem in $O(n)$ if we can find a pivot element that provides a "good" split, which is one that is guaranteed to have at least pn elements on each side, where p is a fraction less than 1/2. The algorithm in this problem is designed to find such a pivot.

Deriving the Recurrence

We begin by dividing the n elements into sets of size 7, which can be done in $O(n)$ time by a variety of methods. We then wish to find the middle element of each set of 7, which can be done with a constant number of comparisons for each set, or $O(n)$ operations in all. The next step requires finding the median of the $n/7$ middle elements. This we can do by recursively calling our program on these elements. Because $T(n)$ is the running time for the algorithm on n elements, this recursive call will take $T(n/7)$ time.

Finally, we will partition the elements around our selected pivot and recursively search on either the left or right side, depending on where the k^{th} element will be found. The cost of this operation will depend on the worst case size of the larger partition. Assume for now that all elements are distinct. Our pivot was the median of the $n/7$ middle elements and was therefore greater than $n/14$ middle elements. Each of these was larger than 3 other elements in their respective set of 7. Therefore, the pivot is larger than at least $(1+3)n/14$ or $2n/7$ elements and the right partition can have at most $5n/7$ elements. A similar argument applies to the left partition. Therefore, the recursive call will involve at most $5n/7$ elements.

Adding up the cost of finding the median of the middle elements, the main recursize call and the other $O(n)$ terms gives:

$$T(n) \leq T(n/7) + T(5n/7) + cn$$

If our set included duplicate elements, we could leave any duplicates of the pivot out of both the left and the right partition. This would only make them smaller so the analysis still holds.

Common Errors

Because T is the time it takes to run this program on some input, $T(n/7)$ and $T(5n/7)$ could only reflect recursive calls of the program. If they were any other operations on $n/7$ or $5n/7$ items, they could not involve T. A number of people seemed to confuse $T(n/7)$ with $O(n/7)$. Others misread the algorithm as creating 7 sets of $n/7$ items. If we were to recursively run the program on these sets, it would cost $7T(n/7)$. Although this is the wrong interpretation, it's worth noting that *in a recurrence relation we cannot drop constant factors on recursive expressions.* For example, $T(n) = 2T(n-1) + c$ is exponential in n, but $T(n) = T(n-1) + c$ is $O(n)$.

Notice that we aren't too concerned here whether n/7 is an integer and we neglect the use of rounding symbols $\lfloor\rfloor$ or $\lceil\rceil$. In general, such considerations don't affect the asymptotic analysis of an algorithm and are best ignored. However, if we were actually programming the algorithm, this would be an important consideration. This is one of the many differences between programming and theory.

Solving the Recurrence

The base case for such a problem is not very interesting. For a fixed size input, any deterministic algorithm that is guaranteed to terminate will do so in constant time. Therefore, for small n, the algorithm will run in cn time.

Assume $T(m) \leq 7cm$ for all $m < n$. We will prove that $T(n) \leq 7cn$.

$$\begin{aligned} T(n) \leq& \quad T(n/7) + T(5n/7) + cn \\ T(n) \leq& \quad 7c(n/7) + 7c(5n/7) + cn \\ T(n) \leq& \quad cn + 5cn + cn \\ T(n) \leq& \quad 7cn \end{aligned}$$

Common Errors

This is one of those problems that looks easy once you see the answer. The key was the use of strong induction. If we assume that the inequality holds for all inputs of size less than n, it is easy.

Some people solved for $T(7n)$ based on $T(n)$ and $T(5n)$ which is just as good as what we've done here. Others tried to prove the recurrence holds for $T(n+1)$. This either lead to incorrect proofs or unnecessarily complicated ones.

Many people tried to expand out the terms in the recurrence. This technique is useful for generating a guess. In this case, however, the guess ($7cn$) was provided. Expanding this particular recurrence generally just resulted in a big mess. Once one is experienced with recurrences it is easier to generate and test a guess than it is to expand terms.

Problem 4

This problem is solved by dynamic programming. There are two (possibly more) solutions. The one I had in mind was to define

$T(i,j) =$ the least cost way of arriving at gas station i with j gallons of gas remaining in the tank.

Then we have

$$T(1,0) = 0, \ \ T(1,j) = \infty \ \text{ for } \ j \geq 1.$$

Also, for $i \geq 2$, to get to the i th station with j gallons in the tank, we must have earlier got to the $i-1$ th station with some s gallons in the tank ($0 \leq s \leq \min(j+1, b)$), bought $j-s+1$ gallons at station $i-1$ and driven to station i. This gives us the dynamic programming recurrence

$$T(i,j) = \min_{0 \leq s \leq \min(j+1,b)} T(i-1,s) + (j-s+1)c_{i-1} \ \text{ for } i = 2, 3, \ldots \text{ and } j = 0, 1, \ldots b,$$

where c_i is the price per gallon of gas at station i. We can now compute a table of $T(i,j)$ in increasing order of i and j . Each entry takes $O(b)$ time for a total of $O(nb^2)$. This computes only the costs. [I gave full points if you got upto this point.]

To find the actual amounts to buy at each station, we keep track of a quantity $P(i,j)$ which is the amount of gas we buy at gas station $i-1$ in the least cost way of reaching station i with j gallons of gas. When we compute $T(i,j)$, we set $P(i,j)$ to be $j-s+1$ for the s that attains the minimum. We can finally backtrack from $T(n/30, 0)$ to find the amounts of gas bought.

There is a better solution which is a little more complicated. For this, define $S(i,j)$ to be the least cost of being at station i with j gallons in the tank at some time (this includes the possibility that we are in the process of filling gas at station i). Then before we got to this state (of being at station i with j gallons in tank), the immediate earlier state was : either we were in station i with $j-1$ gallons of gas in the tank or we were in station $i-1$ with $j+1$ gallons in the tank.

Thus, we now have the recurrence

$$S(i,j) = \min(S(i,j-1) + c_i, S(i-1,j+1)).$$

This needs only $O(1)$ per entry for a total of $O(nb)$.

Common Errors

Many people said : look b gas stations ahead and fill up enough to get to the nearest one with lesser cost; if all of them have a higher cost, fill up the tank here. This does not work because if for example the cost at stations $2, 3, \ldots b$ were all higher than the one at station 1, we will fill up at station 1. Then we will run out of gas at station b and have to fill at least 1 gallon there. Now, it may be that the cost per gallon of gas at station b is more than b times that at any other station, in which case this is not a good strategy. There are some more complicated versions of this algorithm (for example, this particular example can be taken care of) , but none seems to work in general.

Some people had a correct solution by considering the quantity : $R(i, j)$ which is the least cost way of reaching the end starting at gas station i with j gallons in the tank.