HW4 Solution Sketches

Feb. 28, 1999

9.1-3 Consider the tree that you get by using the comparison model. Each node has at most two children. (If the test performed at the node is $a_i > a_j$, then one child corresponds to the test being true, the other corresponds to it being false.) In this tree, there can be at most $2^{cn+1}$ leaves at depth $< cn$. Since each different input goes down a different path (and hence ends up at a different leaf), there can be at most $2^m$ inputs for which the running time is $< cn$. For sufficiently large $n$, we must have $n! / 2 > 2^{cn}$; we also have $n! / n > 2^{cn}$ and $n! / 2^n > 2^m$ (see below). It follows that there is no comparison sort where even a fraction $1/2^n$ of the inputs run in time $cn$. To see that $n! / 2^n > 2^m$ for $n$ sufficiently large (the other two cases follow from this one), assume without loss of generality that $c > 1$. Multiply both sides by $2^n$; then take lg of both sides. Thus, it is sufficient to show that $\log(n!) = \log(n) + \log(n-1) + \cdots + \log(2) + \log(1) > (c+1)n$. If $n' > 2^{2c}$, then $\log(n') > 2c$. Thus, $\sum_{k=1}^n \log(k) = \sum_{k=1}^{2^k} \log(k) + \sum_{k=2^k}^n \log(k) > 2c(n-2^{2c})$. It is easy to see that if $(c-1)n > 2^{2c-2c+1}$, then $2c(n-2^{2c}) > (c+1)n$.

[Grader: David Welte]

9.1-4 If you can do a 3-way comparison, you get a ternary tree instead of a binary tree. A ternary tree of height $h$ has at most $3^h$ leaves, so using the same argument as on p. 174, for a decision tree of height $h$ to do sorting, we must have $n! < 3^h$, so $h > \log_3(n!)$. Now standard properties of logs show that $\log_3(b) = \log(b) / \log(c)$ (since if $\log_3(b) = x$, then $c^x = 2^{x \log c} = b$, so $\log(b) = x \log c$ and $x = \log b / \log c$). Thus, $h > \log n! / \log 3 > n \log n / 2 \log 3 = \Omega(n \log n)$.

[Grader: David Welte]

9.2-2 Suppose $A[i] = A[j]$ and $i < j$. We need to show that $A[i]$ appears lower in the sorted list than $A[j]$. To make sense out of this, we need to assume that there is satellite data, or other information in $x$ besides the number that is being sorted (e.g., when we’re applying CountingSort to RadixSort, we’re sorting on one of the digits). To make this precise, suppose we’re sorting elements $x$ by their keys $k[x]$. So we really want to show that if $A[i] = x$, $A[j] = y$, $i < j$, $k[x] = k[y]$, $B[i'] = x$, and $B[j'] = y$, then $i' < j'$. This basically follows from lines 10 and 11 in the code for CountingSort on p. 176. For suppose that after executing line 7 of CountingSort, $C[k[x]] = m$; that means that there are $m$ elements with keys less than or equal to $k[x]$. Suppose that there are $m'$ elements with key $k[x]$. Lines 9-11 ensure that they go into slots $B[m - m' + 1..m]$ in the order they occurred in array $A$, which is just what we want for stability. For in executing lines 9-11, we start at the top end of the array and work our way down. The last element with key $k[x]$ will be first one encountered with that key, and it will
go in slot $B[C[x]]$, which is $m$. At that point, $C[k[x]] \leftarrow C[k[x]] - 1$, so the next element from the top with key $k[x]$ will go in slot $B[m - 1]$, and so on. [The proof here doesn’t have to be too formal.]

[Grader: Rami Baalbaki]

9.3-2 Merge sort and insertion sort are stable, heap sort and quick sort aren’t. [Proofs are not required here - 1 point for each correct response.] Here’s one way to make a sorting scheme stable. Given an array $A$, replace $A[i]$ by the pair $(A[i], i)$ for $i = 1, \ldots \text{length}[A]$. That is, we tag the $i$th element in the array by $i$. Call the resulting array $A'$. This tagging process takes time $O(n)$ and additional space $O(n \lg n)$. (Note that writing the number $n$ in binary takes space $\lg n$.) Now we compare elements lexicographically: that is $A'[i] < A'[j]$ if either $A[i] < A[j]$ or $A[i] = A[j]$ and $i < j$. Next apply your favorite comparison-based sorting algorithm to $A'$ using the lexicographic order. In this case, the comparisons take a little longer because you’re comparing “bigger” elements, although the order of magnitude doesn’t change. Finally, delete the second element that you added (another $O(n)$ steps).

The same idea works even if the sorting scheme isn’t comparison-based. Then we would just sort $A'$ by the first component, carrying along the second component, and then sort the elements with the same first component by their second component. This takes additional time $O(n \lg k)$, where $k$ is the biggest number of items with the same first component.

[Grader: Ronnie Choy]

9.3-3 We show by induction that after the $i$th pass through the algorithm, the elements are sorted by their $i$ lowest-order digits. The base case $i = 1$ is immediate from the algorithm. Clearly after the first pass through, the elements are sorted by the least-significant digit. Suppose that after the $i$th pass through the algorithm, the elements are order by their lowest $i$ significant digits. Consider what happens on the $i$th pass. The algorithm orders the elements on the $i$th significant digit. We need to check that, not only are the elements ordered on the $i$th digit, but they’re actually ordered on the all the lowest $i$ digits. The only problem comes if we have two elements with the same $i$th digit. We need to make sure that they’re in order. But this follows immediately from stability, since they were in order before. [I’d actually prefer a little more detail than this.]

[Grader: Indranil Gupta]

10.3-1 Using the same techniques as in the text, if we divide the inputs into groups of $2k + 1$, there will be roughly $n/(2k + 1)$ groups. If $q$ is the median of the medians, in half of those groups, that is, roughly $n/(4k + 2)$ groups, the median will be smaller than $q$. Thus, in $n/(4k + 2)$ of the groups, there will be $k + 1$ elements smaller than $q$. This means that there will be at least $(k + 1)n/(4k + 2)$ elements smaller than $q$ (I’m ignoring constants here). Similarly, there will be at least $(k + 1)n/(4k + 2)$ elements larger than $q$. Thus, when we apply the algorithm recursively, we apply it to at most $n - (k + 1)n/(4k + 2) = (3k + 1)n/(4k + 2)$. (In the case $k = 2$ done in the book, we get $7n/10$.) Thus, we get the recurrence $T(n) \leq T((3k + 1)n/(4k + 2)) + T(n/(2k + 1))$. The algorithm will work in linear time as long as $(3k + 1)/(4k + 2) + 1/(2k + 1) = (3k + 3)/(4k + 2) < 1$. Thus is false if $k = 1$ (i.e., if we have groups of 3), but true if $k > 1$ (in particular, it is true if $n = 3, 5, and 7$).
10.3-3 To make quicksort run in time $O(n \lg n)$, we simply choose the median and pivot around the median at each step. This guarantees we split the array in half. The running time in this case is

$$T(n) = O(n) + 2T([n/2]),$$

since finding the median can be done in time $O(n)$. By the master theorem, $T(n) = O(n \lg n)$.

**Common mistakes:** In the running-time analysis, some students thought the total time is the sum of the extra time for finding the medians and the time for quick-sorting and they claimed $T(n) = O(n) + O(n \lg n)$. Actually, the partition of each subarray need to call the SELECT procedure to find the median. So the time for finding the medians is not $O(n)$.

[Grader: Lantian Zheng]

10.3-9 Let $y_1, \ldots, y_n$ be the $y$ coordinates of the wells. Let $y_{med}$ be the median of the coordinates. We claim that the main pipeline should run horizontally at $y = y_{med}$. (If $n$ is even, it doesn’t matter which of the two medians is chosen.) By results of Section 10.3, this can be determined in linear time.

To see that the median is the right answer, suppose for simplicity that $n = 2k + 1$ is odd, $y_{k+1}$ is the median, the first $k$ wells have $y$-coordinate less than the median, so $y_1, \ldots, y_k$ are all $< y_{k+1}$, and the last $k$ wells have $y$ coordinate greater than the median, so $y_{k+2}, \ldots, y_{2k+1}$ are all $> y_{2k+1}$. (This assumption just makes it easier to write up the proof.) First suppose that the main pipeline goes through $y = y_{med}$. Pair up the wells at $y_j$ and $y_{k+1+j}$, for $j = 1, \ldots, k$. Notice that the total length of the spur going to $y_j$ and the spur going to $y_{k+1+j}$ is $y_{k+1+j} - y_j$. Indeed, this is true as long as the main line goes between $y_j$ and $y_{k+1+j}$. If the main line is not between $y_j$ and $y_{k+1+j}$, the sum of the spurs is greater than $y_{k+1+j} - y_j$. By putting the main line at $y = y_{med}$, we do not need a spur line to $y_{k+1}$ at all. Thus, by putting the main line at $y = y_{med}$, the sum of the length of the spur lines is $\sum_{j=1}^{k} (y_{k+1+j} - y_j)$. Any other placement makes it worse, since the sum of the spurs to $y_j$ and $y_{k+1+j}$ is still at least $y_{k+1+j} - y_j$ and now we need to add the length of the the spur to $y_{k+1}$.

If $n$ is even, the same argument shows that we can put the mainline anywhere between the two median values.

[Grading: 1 point for saying to put it at the median, 1 point for observing that therefore the placement can be found in linear time, and 5 points for showing that this is the right choice.]

**Common mistakes:**

1. Some students forgot to prove that the median is the right choice.

2. A few students tried to prove it by contradiction, but what they proved is that if moving the pipeline a little bit away the median then the result location is worse. From this, we can only conclude that the median is a local minimum. If moving the pipeline for a long distance, then it’s difficult to show that the result location is not as good as the median.
3. Several students claimed that the average of \( y_1, y_2, \ldots, y_n \) is the optimal location. This is not right. A simple counter example: \( n = 3, y_1 = 1, y_2 = y_3 = 10 \).

[Grader: Lantian Zheng]

11.2-2 Push becomes List-Insert and Pop becomes List-Delete\((L, \text{head}[L])\). Finally Stack-Empty becomes a test to see if head\(L\) = nil. [Deduct one point if they don’t mention Stack-Empty.]

[Grader: Randy Fernando]

11.2-5 We can use a doubly-linked list. If each of \( S_1 \) and \( S_2 \) is represented by a doubly-linked list, then we can get a list representing \( S_1 \cup S_2 \) by linking the tail of the list representing \( S_1 \) to the head of the list representing \( S_2 \). [For full credit, you should write pseudocode for Union. 2 points were deducted if you just used English here.]

**Comments on 11.2-2 and 11.2-5:**

- Remember that “next” is used for elements of lists, not for lists. For example, \( \text{next}[S] \) where \( S \) is a list doesn’t make sense, unless we make some sort of special definition.

- Remember that the code to link the two lists should actually return a list! Many people forgot this, and the grading was pretty lenient in the sense that we tried to figure out which list was intended (normally the first one). Please make sure not to make this kind of mistake again, because you will definitely lose more points.

- Try to think through your solutions and see if all the pointers make sense in the final result. In some cases, this wasn’t the case.

- Try to pick the simplest data structure that will solve the problem. Even if this is not explicitly stated in the question, it is always part of the problem requirements. Pretend that you actually had to program the solution... you should always pick the easiest thing that works and satisfies the problem’s explicit requirement such as space used and running time. Many people tried to link circular lists with sentinels when a double-linked list or a singly-linked list (modified to have a pointer straight to the tail) would worked much more easily!

- Use standard notation, or if you feel some pressing need to use some new notation, make sure to define it clearly. In general, stick to what has been discussed in class or in the book. The occasions will be rare, if at all, where new notation needs to be defined.

[Grader: Randy Fernando]

12.2-4 Keeping the list in sorted order shouldn’t affect successful searches. An unsuccessful search takes time about half the length of the list on average; we can quite once we’ve gone past the element we’re searching for, and on average, the element we’re searching for will go around the middle of the list. Insertion will take longer: on average, it will take time about half the length of the list (since we have to run along the list to find the right place to do insertion),
whereas insertion at the head takes time $O(1)$ independent of the length of the list. Note that for deletion, we’re given a key, not a pointer (unlike the standard deletion in linked lists). If the key we want to delete is actually in the list, on average, we have to search through about half the list to find it, whether or not the list is in sorted order. On the other hand, if the key is not in the list, having the list sorted saves a factor of two on average, just as with unsuccessful search. Is it worth sorting? Probably not, unless the lists are long and we expect to have quite a few unsuccessful searches in our application. [Grading: an informal argument as I’ve done it is OK for this one.]

[Grader: Rami Baalbaki]

12.3-1 Given a key $k$, for each element in the list, check if $h(k)$ is the hash value for that element. If so, check if $k$ is the key for that element. While this approach has the overhead of needing to compute $h(k)$ and check hash values, if the keys are long, testing hash values for equality is much faster than testing keys for equality, so this approach should be faster.

[Grader: Ronnie Choy]