CSE 101 Homework 1 Answers

1. (Exercise 1.3-1)

2. Let $T(n)$ be the time to sort a list of $n$ elements using the recursive version of insertion sort. This time includes $T(n-1)$ time for the recursive call to sort the first $n-1$ elements, plus the time to insert the last element into this sorted sequence using a loop like that in lines 2-8 of the insertion sort algorithm on Page 8. This loop takes at most $n-1$ iterations, requiring $O(n)$ time, so we have:

$$T(n) = T(n-1) + O(n)$$

When the array is reverse-sorted to begin with, this loop proceeds all the way down to the first element, taking $\Theta(n)$ steps. The time for this loop can be no worse, so this is the worst case that we wish to analyze here. For this worst case:

$$T(n) = T(n-1) + \Theta(n)$$

This recurrence may be solved using the iteration method; this solution is given on Page 156 of the textbook: $T(n) = \Theta(n^2)$.

3. For $n = 14$, $100n^2 = 19600$ and $2^n = 16384$, so the $2^n$ algorithm is still faster. But when $n = 15$, $100n^2 = 22500$ and $2^n = 32768$; so $n = 15$ is the smallest value for which the $100n^2$ algorithm is faster.

4. (Exercise 8.1-1)

5. PARTITION takes time $\Theta(n)$ on an array of size $n$, and divides it into two pieces, of size $q$ and $n-q$; recursive calls on these take time $T(q)$ and $T(n-q)$. Since each of the two pieces has size at least 1, $1 \leq q \leq n-1$. The best-case running time can be no better than that obtained by choosing $q$ to minimize the total time taken by the recursive calls. So, letting $T(n)$ denote the best-case running time for QUICKSORT:

$$T(n) = \min_{1 \leq q \leq n-1} (T(q) + T(n-q)) + \Theta(n)$$

Using the substitution method to solve this recurrence, we want to show that $T(n) \geq cn \lg n$ for some constant $c$. For $n = 1$ this is simply $T(n) \geq 0$, which is trivially true, establishing the base case. Now we substitute this into the recurrence:

$$T(n) \geq \min_{1 \leq q \leq n-1} (cq \lg q + c(n-q) \lg(n-q)) + \Theta(n)$$

$$T(n) \geq c \min_{1 \leq q \leq n-1} (q \lg q + (n-q) \lg(n-q)) + \Theta(n)$$

We need to minimize the expression $q \lg q + (n-q) \lg(n-q)$ over this range of $q$. Taking the first derivative with respect to $q$ yields:

$$\lg q + \frac{q}{q} - \lg(n-q) - \frac{n-q}{n-q} = \lg q - \lg(n-q)$$

This expression is equal to 0 when $q = n - q$ (that is, $q = \frac{n}{2}$), and is not equal to 0 anywhere else in this range. To see that this point is a minimum (as opposed to a maximum or a saddle point), note that the derivative $\lg q - \lg(n-q)$ is positive for $q > n-q$ (that is, $q > \frac{n}{2}$), so the function increases as $q$ increases from $q = \frac{n}{2}$; and, the derivative is negative for $q < \frac{n}{2}$, so the function also increases as $q$ decreases from $q = \frac{n}{2}$. This shows that $q = \frac{n}{2}$ is the minimum, so we have:

$$T(n) \geq c(\frac{n}{2} \lg \frac{n}{2} + (n - \frac{n}{2}) \lg(n - \frac{n}{2})) + \Theta(n)$$

$$T(n) \geq cn \lg \frac{n}{2} + \Theta(n)$$

$$T(n) \geq cn \lg n - cn + \Theta(n)$$

We can choose $c$ small enough so that $cn$ is dominated by $\Theta(n)$, giving:

$$T(n) \geq cn \lg n$$

Thus, the best-case running time of QUICKSORT is $\Omega(n \lg n)$.

6. Problem 8-3 (a) We prove correctness by induction on $n$, the number of elements in the array $A$ passed to STOOGE-SORT. For the base case, consider $n = 1$ and $n = 2$. For $n = 1$, line 1's check fails so line 2 does not do anything, line 3's check passes so the function immediately returns; arrays of size 1 are correctly left untouched. For $n = 2$, lines 1-2 put the pair in proper order if they are initially out of order, then lines 3's check passes so the function returns; arrays of size 2 are put in proper order.

   Now, assuming that STOOGE-SORT correctly sorts arrays of size $< n$ (this is the induction hypothesis), show that it correctly sorts arrays of size $n$ (for $n > 2$).

   Let $A_0$ denote $A$ after line 2 is executed. Line 2 may exchange two elements of the input, but in any case $A_0[i \ldots j]$ contains a permutation of the input - so if we correctly sort this part of $A_0$ then we have correctly sorted the input. Since $n > 2$, line 3's check fails and the function does not terminate at line 4.

   Define $k$ according to line 5. We now want to show that lines 6-8 produce a properly sorted permutation. Consider the $k+1$ largest-valued elements in $A$ (if this set is not uniquely defined due to duplicate elements, the following argument will still apply to *some* set of $k+1$ largest-valued elements). Before line 6, some of these largest-valued elements are among $A[j-k+1 \ldots j]$, and the remaining largest-valued elements are among $A[1 \ldots j-k]$. After line 6, $A[1 \ldots j-k]$ is sorted correctly by the induction hypothesis, so these remaining largest-valued elements must be among $A[i+k \ldots j-k]$ (since there are at most $k+1$ such remaining largest-valued elements, and $A[i+k \ldots j-k]$ has at least $k+1$ elements). Thus, all of the $k+1$ largest elements are among $A[i+k \ldots j]$ after line 6. Line 7 then correctly sorts $A[i+k \ldots j]$ (by the induction hypothesis), so the $k+1$ largest elements are among $A[j-k \ldots j]$ and are in correct sorted order at that point. The remaining smaller elements are properly placed in $A[1 \ldots j-k-1]$, and after line 8 is executed, these are correctly sorted as well. At this point, the entire array is correctly sorted.

   Thus, arrays of size $n$ are also correctly sorted. By induction, STOOGE-SORT correctly sorts arrays of all sizes.

   (b) Say STOOGE-SORT requires $T(n)$ time to sort an array of size $n$. During a call to STOOGE-SORT, lines 1-5 take only $\Theta(1)$ time. Each recursive call in lines 6-8 is done on an array of size $\lceil \frac{2n}{3} \rceil$. So, we have the recurrence:

$$T(n) = 3T(\lceil \frac{2n}{3} \rceil) + \Theta(1)$$

   We can apply the master theorem (page 62) with $a = 3$ and $b = \frac{3}{2}$. Since $f(n)$ is $\Theta(1)$, it is $O(n^{\log_{\frac{3}{2}} 3 - \epsilon})$, since this is about $n^{2.7}$ if we pick $\epsilon = 0.1$ (other values of $\epsilon$ would also work, as long as they aren't too large). So, case 1 of the master theorem applies, giving:

$$T(n) = \Theta(n^{\log_{\frac{3}{2}} 3}) \approx \Theta(n^{2.71})$$

   So, it is even slower than insertion sort's worst case.

7. (Exercise 9.2-1)

8. SELECT has worst-case running time that is $O(n)$. So, modify the PARTITION function used by QUICKSORT to first call SELECT to find the median of the array (in $O(n)$ time), then partition around this element (in $\Theta(n)$ time). The total time required by this modified PARTITION is not (asymptotically) worse; it is still $\Theta(n)$. But this modified PARTITION will *always* give balanced

partitions of $\frac{n}{2}$ elements each (rounded up or down). As shown on page 157, this gives a running time for QUICKSORT that is $O(n \lg n)$; this is now a worst-case running time since the use of SELECT guarantees that it is always achieved.

One additional detail: if there are duplicates of the median, there is no guarantee that the partition will be exactly balanced as described above. There are a variety of ways to handle this. One simple hack is to (temporarily) add small, differing amounts to each duplicate of the median - small enough so that none of their values becomes larger than the next larger value in the original array (so that the overall sorted order is not disrupted). Choose these small, differing values in increasing order, so that we know what their median is. This all takes only linear time, so the $\Theta(n)$ time for PARTITION is unaffected. We now know the *unique* median of the modified array, so we get our completely balanced partition around this, and then change the temporarily modified values back to their original value (the original duplicated median value).

9. We modify the SELECT algorithm described on Page 190 to get WEIGHTED-SELECT below. $X$ and $W$ hold the elements and weights specified in the problem, and $p$ and $r$ specify the boundaries of the portion of these arrays we are to use; we are selecting from elements $X[p \ldots r]$. Given a desired target weight $t$ with $0 \leq t \leq 1$, WEIGHTED-SELECT finds an element $x_i$ satisfying $\sum_{x_i < x_k} w_i \leq t$ and $\sum_{x_i > x_k} w_i \leq 1 - t$. So calling WEIGHTED-SELECT with $t = \frac{1}{2}$ would give the weighted median.

   Note that WEIGHTED-SELECT simply calls SELECT on line 3 to find a good element (the median) about which to partition. Since we already know that SELECT takes $O(n)$ time in the worst-case, there is no need to use the clever recursive method from inside SELECT for this purpose.

   Note that the MODIFIED-PARTITION function called on line 4 partitions around $x$, as described in step 4 on Page 190; it also carries the weights around with the elements, so $W$ must be passed in. Also note that the arguments to SELECT are specified in the same order as they were for RANDOMIZED-SELECT on Page 187.

   The remainder of the function is similar to SELECT, except that it uses weight rather than number of elements to decide which partition to recurse into. Lines 5-7 count up the total weight $w_l$ on the low side of the partition. If $t$ is larger than this total weight, then we must be looking for an element on the high side of the partition, so we recurse there (appropriately reducing our target weight since we are removing the elements on the low side of the partition from future consideration). Otherwise, the element we seek is on the low side of the partition, so we recurse there.

```
WEIGHTED-SELECT(X,W,p,r,t)
1   if p = r
2       then return X[p]
3   x ← SELECT(X,p,r,⌊(r−p+1)/2⌋)
4   q ← MODIFIED-PARTITION(X,W,p,r,x)
5   w_l ← 0
6   for i ← 1 to q
7       do w_l ← w_l + W[i]
8   if t ≤ w_l
9       then return WEIGHTED-SELECT(X,W,p,q,t)
10      else return WEIGHTED-SELECT(X,W,q+1,r,t − w_l)
```

   We now analyze the time $T(n)$ required to compute the weighted median of $n$ elements. Lines 1-2, 5, and 8 take constant ($\Theta(1)$) time, line 3 takes $O(n)$ time (page 191), line 4 takes $\Theta(n)$ time (page 155; passing in the element about which to partition does not change this running time), and lines 5-7 loop through at most $n$ elements doing constant work with each for a total $O(n)$ time. The total for lines 1-8 is thus $\Theta(n)$. Since we partition around element number $\lfloor \frac{n}{2} \rfloor$ (and we obtain exactly this specified partition, since all elements are distinct as given in the problem), lines 8-9 recurse on $\lceil \frac{n}{2} \rceil$ or $\lfloor \frac{n}{2} \rfloor$ elements.

To get an upper bound on running time, we now have the recurrence:

$$T(n) \leq T(\lceil \frac{n}{2} \rceil) + \Theta(n)$$

Use the substitution method to show that $T(n) \leq cn$. Pick $c$ large enough so this is true for $n = 1$. Now substitute into the recurrence:

$$T(n) \leq c(\lceil \frac{n}{2} \rceil) + \Theta(n)$$

$$T(n) \leq c(\frac{n}{2} + 1) + \Theta(n)$$

$$T(n) \leq cn + (\Theta(n) + 1 - \frac{c}{2}n)$$

Pick $c$ large enough so that $\frac{c}{2}n$ dominates $\Theta(n) + 1$, giving:

$$T(n) \leq cn$$

The lower bound is very similar. We have:

$$T(n) \geq T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$$

Use the substitution method to show that $T(n) \geq cn$. Pick $c$ small enough so this is true for all $n \leq N_0$ ($N_0$ is a constant defined below). Now substitute into the recurrence:

$$T(n) \geq c(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$$

$$T(n) \geq c(\frac{n}{2} - 1) + \Theta(n)$$

$$T(n) \geq cn + (\Theta(n) - (\frac{c}{2}n + 1))$$

We can pick $c$ small enough so that $\frac{c}{2}n + 1$ is dominated by $\Theta(n)$ for $n > N_0$, by picking $N_0$ sufficiently large (this serves to define $N_0$). So:

$$T(n) \geq cn$$

Combining these, $T(n) = \Theta(n)$.