HW1 Solutions

February 11, 2001

Note that Bo graded the first seven problems and Don graded the rest. Please check with the grader first if you have questions.

1.2-1 The Selection-Sort algorithm is straightforward except for one thing: if the elements in A are not all different, you have to somehow tag that you've removed one and not another. One way to do this is by replacing an element you've removed by a NIL or ∞ (which is what I do in the code below). This, of course, means that you end up "destroying" A. Alternatively, you could use a third array, consisting of flags (the flag in slot i is on if you removed element i from array A). It is also possible to sort in place, rearranging the elements in A so that they're in sorted order (although the problem, for some reason, didn't want this). The best case and worst-case running times for selection sort are both $\Theta(n^2)$. Ideally, the analysis should look like that for insertion sort on page 8 (although it's easier). There's one more subtlety. For a **for** loop that looks like "**for** $j \leftarrow 1$ **to** n", do we do this line n times or $n+1$ times (that is, do we have to test when $j = n+1$). That depends in part on the implementation of **for**. (Either version was accepted – these details don't affect the big O analysis.) Here's some pseudocodt:

```
Selection-Sort(array A, array B, n)
 for count ← 1 to n do
    min ← ∞;
    for index ← 1 to n do
       if A[index] < min then
          min ← A[index];
          min_index ← index;
    B[count] ← min;
    A[min_index] ← ∞;
```

Since both loops of the algorithm always execute n times independent of input, this algorithm is $\Theta(n^2)$ for both the best and worse cases.

Some common problems:

1. Many students neglected the case of duplicate array entries.
2. Many students simply wrote down an answer to the running times. You should always include an argument (ideally an analysis) with your answers.

3. Some students just write down one running time without specifying if it is meant to be the same for both cases. You shouldn't expect the grader to always automatically complete your answer for you!

4. Some students are practically doing sort-in-place with array A (i.e. in the ith pass, look through A[i+1..n], find the smallest element, swap it with A[i]), and then copy A[i] to B[i]). This does not exactly violate the requirement of this problem (although it comes close ...), but you should think of the problems involved if you are not allowed to do this. Also note that if you are doing sort-in-place, it is inefficient to swap A[i] every time you find a smaller element when looking through A[i+1..n]. It's better to remember the index of the smallest element in A[i+1..n], and swap A[i] with that element only once. This will not bring down the order of the running time, but is still worth doing.

1.2-4 Of course, the straightforward algorithm computes x, x^2, ..., x^n separately, which is where the $\Theta(n^2)$ comes from. Here it is:

```
val ← A[0]                                  ; handle case x^0
for  j ← 1 to length[A] − 1
   do  multiplier ← 1
       for  k ← 1 to  j                     ; Evaluate x^j
            multiplier ← multiplier ∗ x
       val ← val + multiplier ∗ A[j]
```

Now here's a $\Theta(n)$ algorithm using Horner's Rule:

```
val ← 0
for  j ← length[A] − 1 to 1
   do  val ← (val + A[j]) ∗ x
val ← val + A[0]
```

1.2-3 The outline for this solution is to start by having a counter for each input array starting at the beginning of each. Then the value at each counter is compared and the lower value is put into the next free spot in an output array. Then that counter is advanced. When one counter is at the end of its array, the other array is appended to the output. I try to use CLRish notation, such as arrays starting at 1.

```
Merge(array A, int p, int q, int r)
    i ← p
    j ← q+1
    next ← 1
    array Out[(r − q) + 1]
    while (i ≤ q) or (j ≤ r)
        if j > r then
            Out[next] ← A[i]
            i ← i + 1
```

```
        else-if i ≥ q then
            Out[next] ← A[j]
            j ← j + 1
        else-if A[i] < A[j] then
            Out[next] ← A[i]
            i ← i + 1
        else
            Out[next] ← A[j]
            j ← j + 1
        end-if
        next ← next + 1
    end-while
    A[p...r] ← Out
end-Merge
```

Some general notes on the first 3 problems

1. It's probably good practice to initialize variables before using them (even if CLR doesn't always do this!).
2. Be careful with arrays. When you create an array of length n in C or Java, the index available is between $0..n-1$, not $0..n$ or $1..n$. This would seem obvious if given as a seperate question, but one tends to forget it for time to time (especially since in CLR arrays go from 1 to n). With Java, you're much better off, since it performs arraybound checking dynamically for you. But with C, you might get really weird errors, and it might take you a long time debugging to spot that it is acutally this misuse of index that causes the error.

1.4-2 $n = 15$, since if $n = 15$, then $100n^2 = 22,500$ and $2^15 = 32,768$ (according to my calculator), while if $n = 14$, then $100n^2 = 19,600$ and $2^14 = 16,384$. Note that in this problem, n must be an integer.x

2.1-1 Since f and g are asymptotically nonnegative, there is some n_0 such that $f(n) > 0$ and $g(n) > 0$ for $n > n_0$. Clearly $\max(f(n), g(n)) \leq f(n) + g(n)$ for all n_0. Thus, $\max(f(n), g(n))$ is $O(f(n) + g(n))$. Now we need to show that $\max(f(n), g(n)) \geq c(f(n) + g(n))$ for some constant c and all $n \geq n_0$. Take $c = 1/2$. This works for all n. To see this, suppose $f(n) \geq g(n)$. Then $\max(f(n), g(n)) = f(n)$ and $1/2(f(n) + g(n)) \leq 1/2(f(n) + f(n)) = f(n)$. Thus, $\max(f(n), g(n)) \geq 1/2(f(n) + g(n))$. An identical argument works if $f(n) \leq g(n)$.

Comment: Many of you gave zero as the constant in the lower bound. From the definition of theta notation, the constants in the lower bound and the upper bound must be positive. Also in this question, the correct range of the constant in the lower bound is $0 < c \leq 1/2$. Anything between $1/2$ and 1 will not work. Consider the case where $f(n) = g(n)$. Then $c(f(n) + g(n)) = 2cf(n) > f(n)$ for $c > 1/2$.

2.1-4

1. With $c \geq 2$ and $n_0 \geq 0$, we have that for all $n \geq n_0$, $2^{n+1} \leq c * 2^n$. Thus $2^{n+1} = O(2^n)$ by the definition of O.

2. $2^{2n} \neq O(2^n)$ - following proof is by contradiction.

   Suppose $2^{2n} = O(2^n)$. Then, as per the definition of O, there are constants c and n_0 such that for all $n \geq n_0$, $2^{2n} \leq c * 2^n$.

   Now, $2^{2n} \leq c * 2^n$ implies that $2^n \leq c$, and that $n \leq logc$, meaning that for $n > logc$, the inequality ($2^{2n} \leq c * 2^n$) would not hold, a contradiction.

   Hence, $2^{2n} \neq O(2^n)$.

Comments on grading scheme:

1. Each of the parts was graded for 4 points.

2. For each of the parts: 1 point if the answer is right (yes/no), 2 points if the answer is right and the explanation 'looks' good but is incomplete (does not use the definition of O), and 4 points if the answer and explanation are both right.

3.2-1

$$\begin{aligned}
\sum_{k=1}^{n} 1/k^2 &\leq 1 + \sum_{k=2}^{n} 1/k^2 \\
&\leq 1 + \int_1^n 1/x^2 \, dx \\
&\quad \text{(since } 1/k^2 \text{ is a monotonically decreasing function)} \\
&\leq 1 + [-1/x]_1^n \\
&\leq 1 + [1 - 1/n] \\
&\leq 2 - 1/n \\
&\leq 2
\end{aligned}$$

Common mistakes:

1. *Changing from summation to integral:* Notice first that $1/k^2$ is a monotonically decreasing function, not monotonically increasing - you have to apply the right formula from CLR (the one for monotonically decreasing functions, not the one for monotonically increasing functions). If you have started off applying the wrong formula, you get 1 point out of 5.

2. *Last step:* Many students the upper bound at $(2 - 1/n)$ and not followed up by saying that this is ≤ 2. $(2 - 1/n)$ is not a constant. 2 is the constant required. 1 point deducted for this.

4.1-1 There is a subtlety here: it doesn't tell you what $T(1)$ is. The point is that it doesn't matter. A totally correct solution would start out by saying "Suppose $T(1) = c$. Then $T(n) \leq c + 2\lg n$." It is not hard to prove $T(n) \leq c + 2\lg n$ by strong induction [note it's strong induction, not regular induction; see below], although even here there are some subtleties. It is easy to check that this works for $n = 1$ and 2. Now suppose that $n \geq 3$ and we have the result for all $n' < n$:

$$\begin{aligned}
T(n) &= T(\lceil n/2 \rceil) + 1 \\
&\leq c + 2\lg(\lceil n/2 \rceil) + 1 && \text{induction hypothesis} \\
&\leq c + 2\lg((n+1)/2) + 1 && \text{since } \lceil n/2 \rceil \leq (n+1)/2 \\
&\leq c + 2\lg(n+1) - 2 + 1 \\
&\leq c + 2\lg(n+1) - 1 \\
&\leq c + 2\lg(n) && \text{since } \lg(n+1) \leq \lg(\sqrt{2}n) = \lg(n) + 1/2 \text{ if } n \geq 3
\end{aligned}$$

Common mistakes:

1. A few students didn't prove the base case, and more student made the invalid assumption $T(1) = 1$ in the proof of base case.

2. A lot students made a wrong assumption: $\lceil \frac{n}{2} \rceil \leq \frac{n}{2}$

3. The inductive statement should be $T(n) \leq c + 2\lg n$ or $T(n) \leq c\lg n$, but some students took $T(n) = O(\log n)$ as the statement.

4. Recall that in strong induction, if $P(k)$ is the induction statement (which in this case is the statement "$T(k) \leq cn\lg k$), you get to assume that $P(k)$ holds for all $k < n$ and you must prove $P(n)$ holds (or, equivalently, you get to assume $P(k)$ holds for all $k \leq n$ and you must prove $P(n)$). This is in contrast to regular induction, for which you assume $P(n)$ and prove $P(n+1)$. In this case you need strong induction since when your proving $P(n)$ you use $P(\lceil n/2 \rceil)$. More generally, for full credit in an induction proof, be sure to do the following:

   (a) State exactly what $P(k)$ – the statement that you're trying to prove – is. $P(k)$ should be an English sentence (or at least, be readable as an English sentence) and have a k in it. In this case, $P(k)$ is "$T(k) \leq 2k\lg k + T(1)$".

   (b) Make it clear whether you're using regular induction or strong induction.

   (c) Do the base case.

   (d) Do the inductive step.

4.3-1 For full credit, you need to show what the a, b $f(n)$, ϵ, and n_0 are in the master theorem. For (a), we have $a = 4$, $b = 2$, and $f(n) = n$, so $n^{\lg_b a} = n^2$ and for $\epsilon \leq 1$, $n = O(n^{2-\epsilon})$. Thus, part (1) of the Master Theorem applies and $T(n) = \Theta(n^2)$. For (b), $f(n) = n^2$ and part (2) of the Master Theorem applies; thus, $T(n) = \Theta(n^2 \lg n)$. Finally, for (c), $f(n) = n^3$ and part

(3) of the Master Theorem applies since $4f(n/2) = n^3/2 \leq f(n)$ and $n^3 = \Omega(n^{2+\epsilon})$ for $\epsilon \leq 1$. Thus, $T(n) = \Theta(n^3)$.

Common mistakes:

1. Some students forgot to show what's the value of ϵ.
2. Quite a few students forgot to verify the condition in (3): $af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large n.

5.2-1 To show that $\subseteq$ is a partial order, you need to show that $\subseteq$ is reflexive and transitive. To show that it is not a total order, you need to show that two sets are incomparable with respect to $\subseteq$ (for example, $\{1\}$ and $\{2\}$). It's not enough to say that there are two incomparable sets; you must explicitly give two.

Some comments and observations:

- It is important to give a concrete counterexample to show that you are not just "hand waving". (You lost a point if you didn't.)
- you show reflexivity, antisymmetry, and transitivity, you need to say explicitly that it is because of these three properties that the subset relation is a partial order. Otherwise the proof doesn't really have much meaning.
- Avoid saying that things are "obvious" because it will normally get you into trouble on a proof. "Trivial" is a technical word that can be used, for example, for facts that are true by definition, or often for base cases of proofs by induction.
- Remember, you can't *prove* with an example, because that would only show a single case for which your proof works. To prove something you need to show that it holds for *all* cases. However, you can *disprove* with a single counterexample.

5.2-3 Any examples will do here, as long as they're described explicitly. A wide variety of creative answers were given, some of which could have been thought through a little more carefully because they really stretched the imagination. (They were entertaining, however.) Here are some easy examples that work:

(a) reflexive and symmetric but not transitive: "is friends with" on the set of all people (assume one is one's own friend)

(b) reflexive and transitive but not symmetric: "greater than or equal to" on the set of integers

(c) symmetric and transitive but not reflexive: $\{(a, b) : a$ and b are people, a is a sibling of b or $a = b$, and a and b have siblings$\}$. (Assume you cannot be your own sibling.) Note that this is not reflexive because of a has no siblings, then (a, a) is not in the relation. However, it is easy to see that it is symmetric and transitive.

Some comments and observations:

- Remember if you make up a relation you also need to give the set that you're dealing with. Some relations have certain properties only for certain sets.
- If you need to make a certain assumption (such as "assuming one cannot be one's own sibling"), make sure to write it down. As long as your assumption makes sense, you'll get credit for the example.

5.3-2 $f(x) = x + 1$ is injective for both $\mathbf{N}$ and $\mathbf{Z}$, but surjective only if the domain and codomain are $\mathbf{Z}$, since there is no x such that $f(x) = 0$ if the domain is $\mathbf{N}$. Thus, it is a bijection for $\mathbf{Z}$, but not for $\mathbf{N}$. (It is not enough just to say yes or no; there needs to be an explanation.)

5.4-5 You need to show that reachability is reflexive, symmetric, and transitive in undirected graphs. It's not symmetric in directed graphs, but it is reflexive and transitive, but not necessarily symmetric. (One point was deducted if there was no counterexample showing it is not symmetric.)

Comments:

(a) People were confused between reflexivity and symmetry.

R is reflexive if for all a, we have (aRa) (alternatively, if you're used to thinking of a relation as a set $((a, a) \in R$ for all a in the domain.

R is symmetric if for all a, b, we have that if (aRb) then (bRa) (i.e., if $(a, b) \in R$, then $(b, a) \in R$.

Finally, R is transitive if for all a, b, c, if (aRb) and (bRC), then (aRc). These relations are defined over a set (i.e., a domain).

(b) PROOF BY EXAMPLE IS NOT ACCEPTABLE. We were somewhat lenient this time.

(c) For graphs, you don't need self loops for the "is reachable" function to be reflexive. All you need to say is that if you are at a node you can just not move. That means node a is reachable from itself.

(d) A counterexample to show reachability in digraphs is not symmetric. This is trivial. Suppose there are two nodes, A and B, and only one edge, from A to B. A is reachable from B but B is not reachable from A.

5.5-4 This is a straightforward induction. Deduct one point if it's not set up right.

Let N= # of nodes in the tree
Let L= # of leaves in the tree
Let $N2$= # of nodes with outdegree 2.

Claim: $N2 = L - 1$. We prove this by induction on N:

Base Case: $N = 1$. It follows that $L = 1$ and $N2 = 0$, which satisfies $N2 = L - 1$.

Induction Hypothesis: Suppose the result holds for $N = k$. We need to prove that for any binary tree with $N = k + 1$, we have $N2 = L - 1$.

Assume we have a binary tree T with $k+1$ nodes and L leaves and $N2$ degree-2 nodes. We must have $L \geq 1$. Remove one of the leaves. Let T' be the resulting tree. T' has k nodes. Suppose it has L' leaves and $N2'$ degree-2 nodes. By the induction hypothesis, T' has $N2' = L' - 1$. Now there are two cases. (1) The leaf that you removed was the only child of its parent in T (that is, the parent is a degree-1 node in T). In that case, the parent is a leaf in T'. That means T and T' have the same number of leaves and the same number of degree-2 nodes (i.e., $L = L'$ and $N2 = N2'$), so the result follows from the induction hypothesis. (2) The parent of the leaf you removed had two children (i.e., it is a degree-2 node in T). That means the parent is a degree-1 node in T' and T' has one less leaf than T, so $L' = L - 1$ and $N2' = N2 - 1$. Again, it follows from the induction hypothesis that $N2 = L - 1$.

Some common problems:

1. Many students confused what they were inducting upon throughout the proof. For instance, many students started by inducting of the number of degree-2 nodes in the base case, but during the inductive step, switched to inducting on the number of leaves. Explicitly stating what you are inducting on (by way of an induction hypothesis) might have helped many of the students who became confused.

2. Several students' inductive steps failed to take in account all cases.

6.1-3 Since rotations of any seating are considered to be the same seating, we can look at the problem from the perspective of wherever professor 1 is sitting. Call his seat "seat 1". Then any one of the remaining $(n-1)$ professors can sit next to him in seat 2, and then any of the remaining $(n-2)$ professors can sit in seat 3, etc. This gives a total of $(n-1)!$ ways to arrange the professors.

Common mistake: $n!$ would be the answer only if different rotations mattered. It is not correct in this case to say "There are n choices for the first seat, $(n-1)$ choices for the second, etc." because then we are double-counting configurations that shouldn't be counted under the assumption that rotation of a seating doesn't change it.

6.1-4 There are two ways to get three numbers to sum to an even number:

1. all three numbers are even,

2. two of the numbers are odd, and one is even.

Obvious Note: There are 50 even and 50 odd numbers in the set of all the numbers between 1 and 100, inclusive.

Let E denote the set of even numbers between 1 and 100. Let O denote the set of odd numbers between 1 and 100.

Case 1: There are $C(50, 3)$ ways to pick 3 even numbers from E.

Case 2: There are $C(50, 2)$ ways to pick 2 odd numbers from O, and 50 ways to pick 1 even number from E. So there are $50 \times C(50, 2)$ of choosing 2 odd numbers and 1 even number.

Since we are looking for the total number of ways of choosing these numbers, we sum the two cases together. So, there are $C(50, 3) + 50 \times C(50, 2)$ total ways.