HW1 CLRS Questions: Answers

November 3, 2012

Section 2.2., p.29. *2.2-1.* Express $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ notation.

The answer's going to be $\Theta(n^3)$. To work it out systematically, the easiest thing to do is to use the *Seven Rules for Big-O and* Θ at `http://web.cs.wpi.edu/~guttman/cs2223/seven_rules.pdf`. Writing e for the original expression, rule (2) applied repeatedly tells us

$$\Theta(e) = \Theta(\max(n^3/1000,\ -100n^2,\ -100n,\ 3)).$$

Applying rule (1) to each of these, this will equal $\Theta(\max(n^3,\ n^2,\ n,\ 1))$. Now we can apply rule (4) to see that[1]

$$\Theta(1) < \Theta(n) < \Theta(n^2) < \Theta(n^3).$$

So $\Theta(\max(n^3,\ n^2,\ n,\ 1)) = \Theta(n^3)$.

2.2-2. Selection sort grabs the minimum in the whole array and exchanges it with `a[1]`, then the minimum within `a[2]..a[#a]`, exchanging it with `a[2]`, etc. So the code in Fig. 1 will do it.

What invariant is true at the top of the outermost loop? When i has a particular value at the top of the loop, two things hold:

1. the part of the array from $a[1]$ to $a[i-1]$ inclusive is already sorted;
2. for every element x in the part of the array from $a[1]$ to $a[i-1]$ and y in the part of the array from $a[i]$ to $a[\#a]$, $x \leq y$.

Let's write n for $\#a$. Then the invariant is true when $i = 1$ because $a[1]$ to $a[0]$ is empty, so it comes for free. When the loop index increases to some new value $i+1$, the sorted portion increases because the element we just put

[1] also using the fact that $1 = n^0$.

```
function selection_sort (a)
   for i = 1, #a-1 do
      local min = a[i]
      local min_loc = i
      for j = i+1, #a do
         if a[j] < min          -- less than min?
         then
            min = a[j]          -- remember it
            min_loc = j
         end
      end
      -- swap em!  This works in Lua
      a[i], a[min_loc] = a[min_loc], a[i]
   end
end
```

Figure 1: Selection sort

into $a[i]$ was (by part 2) greater than or equal to anything already in the sorted part up to $i-1$. And it was chosen to be minimal in the remainder.

We can stop after $i = n-1$ because the last thing (by the invariant part 2) is already greater than anything earlier.

Looking at the two nested loops, you see that we have to walk through all the entries in $a[i+1]$ to $a[n]$ once for each i. So the execution time will be:

$$\sum_{1 \le i \le n} c \cdot (n-i).$$

This c reflects the proportion of the cases where we have to update the minimum, since in that case we do some more work. However, if c_{worst} is the amount of work we do when we have to do the updates, and c_{best} is the amount we do when we *don't*, we always have c reflecting some mixture of those two cases. So $c_{\text{best}} \le c \le$ worst. Now

$$\sum_{1 \le i \le n} c \cdot (n-i) = \sum_{0 \le i \le n-1} c \cdot (i) \quad \in \Theta(\sum_{0 \le i \le n-1} i).$$

Finally, $\sum_{0 \le i \le n-1} i$ equal to $n \cdot (n-1)/2$.

2.2-3. We're asked to assume that the target value is found in each of the n slots equally often. When the object is in slot i, linear search looks in the

first i slots and then finds it. So the expected work is

$$\sum_{1 \le i \le n} i/n \quad = \quad (n+1) \cdot n/2n \quad = \quad (n+1)/2.$$

So the expected work is about half the number of slots.

2.2-3. When we're programming the algorithm, we can work out the answer to one input problem. We then write the program to check for this input. When it's found, we just give the pre-computed answer. Otherwise, we do the normal computation.

At run-time, if we get the selected input, we have to do just the work to recognize that it *is* the input we hard-coded in. This is $O(c_0)$ where c_0 is the input length. Assuming that the solution is of length c_1, the work we do is $O(c_0 + c_1) = O(1)$.

So "best case execution cost" is pretty meaningless.

Ch. 2. *2-3. (a)* It's linear in n, so $\Theta(n)$.

(b) The naïve polynomial evaluation algorithm would use n multiplications to get x^n, then another to get $a_n x^n$, etc. This leads to an algorithm in $\Theta(n^2)$, i.e. significantly worse than Horner's rule.

2-4. (a) 6 has an inversion with 8, and 1 has an inversion with each predecessor.

(b) The reversed array $\{n, n-1, \ldots, 2, 1\}$ has $n(n-1)/2$ inversions.

(c) Insertion sort removes inversions from left to right. At any stage, when looking at the array element $a[i]$, it has to take a step for every inversion remaining between it and its predecessors. Ultimately, it does an amount of work proportional to the number of inversions plus the length of the array.

Sec. 3.1. *3.1-1.* To prove: $\max(f(n), g(n)) \in \Theta(f(n) + g(n))$. We are allowed to assume that f and g are asymptotically non-negative, meaning that there's an N_0 such that for all $n > N_0$, $f(n) \ge 0$ and $g(n) \ge 0$.

Letting $n > N_0$, we certainly have $\max(f(n), g(n)) < f(n) + g(n)$, so the constant 1 works in establishing $\max(f(n), g(n)) \in O(f(n) + g(n))$. But any constant > 2 works in establishing $f(n) + g(n) \in O(\max(f(n), g(n)))$, since the maximum is at least half the total.

3.1-4. $2^{n+1} = 2 \cdot (2^n)$, so the constant 2 works for c for any N_0.

On the other hand, $2^{2n} = 2^n \cdot 2^n$. So suppose we want to make $2^{2n} \le c \cdot 2^n$. Dividing through by 2^n, we have $2^n \le c$. But no c is greater than *all* the numbers 2^n for $n > N_0$.

Ch. 3. *3-1. (a)* If $k \geq d$, we want to show that $p(n) \in O(n^k)$. So we need to choose N_0, c so that $p(n) \leq c \cdot n^k$ for all $n > N_0$.

First, you can choose $N_0 > \max a_1, \ldots, a_d$. This ensures that the leading term of $p(n)$ is the largest. Now choose $c = (d+1)a_d$. So $p(n)$ is a sum of $d+1$ values, of which $a_d \cdot x^d$ is the largest. But $(d+1)a_d x^k$ is the sum of $d+1$ values, all equal to this largest term if $k = d$, and larger otherwise.

3-1. (b) We want to show $n^k \leq c \cdot (p(n))$ when $k \leq d$. But if we $c \geq 1/a_d$, this holds.

3-1. (c) Combine (a) and (b).

3-1. (d) By (c), we know that $p(n) \in \Theta(n^d)$. So it's enough to show that $n^d \in o(n^k)$ when $d < k$. We must show that for every c, there is an N_0 such that for $n > N_0$ we have $n^d < c \cdot n^k$. Dividing through, this means:

$$1 < c \cdot n^{k-d}, \quad \text{equivalently,} \quad 1/c < n^{k-d}.$$

So we can let N_0 be any number greater than the $(k-d)^{\text{th}}$ root of $1/c$.

3-1. (e) By (c), we know that $p(n) \in \Theta(n^d)$. So it's enough to show that $n^d \in \omega(n^k)$ when $k < d$. But applying part (d) with k, d reversed, we know exactly that.

3-2. You can use the Seven Rules.

(a) Rule 5 says that the answer is o. $f \in O(g)$ is also true (it says less).

(b) Rule 6 says that the answer is o. $f \in O(g)$ is also true (it says less).

(c) Note that $n^{\sin n} \in O(n^1)$ since $\sin n \leq 1$. So the answer is ω. $f \in \Omega(g)$ is also true (it says less).

(d) Here the answer is ω and Ω.

(e,f) Here the answer is o and O.

3-3. See sheet in `http://web.cs.wpi.edu/~guttman/cs2223/hw1_3-3.pdf`.