CPSC 320 Sample Solution, Asymptotic Analysis

September 21, 2016

1 Comparing Orders of Growth for Functions

For each of the functions below, give the best Θ bound you can find and then arrange these functions by increasing order of growth. Note that the last two are challenge problems.

SOLUTION:

Original function	Good Θ bound	Notes
$\ln n$	$\Theta(\lg n)$	$\log n$, $\lg n$, and $\ln n$ differ by a constant factor
$\frac{n}{\log n}$	$\Theta(\frac{n}{\lg n})$	But that does **not** make $\log n$ a constant factor!
$55n + 4$	$\Theta(n)$	
$1.5n \lg n$	$\Theta(n \lg n)$	
$2n \log(n^2) = 4n \log n$	$\Theta(n \lg n)$	(same "rank" as previous line)
$n + n^2$	$\Theta(n^2)$	
$(n \lg n)(n + 1) = n^2 \lg n + n \lg n$	$\Theta(n^2 \lg n)$	
$\sqrt{n}^{\sqrt{n}} = (n^{\frac{1}{2}})^{\sqrt{n}}$	$\Theta(n^{\frac{\sqrt{n}}{2}})$	(we can argue about the "best" form of this)
2^n	$\Theta(2^n)$	
$1.6^{2n} = (1.6^2)^n = 2.56^n$	$\Theta(2.56^n)$	
$n!$	$\Theta(n!)$	
$(n + 1)!$	$\Theta((n + 1)!) = \Theta(n \times n!)$	

Note: $\lg n = \log_2 n$, $\ln n = \log_e n$, and $\log n = \log_{10} n$.

For the comparison of $n^{\frac{\sqrt{n}}{2}}$ and 2^n, one approach is to make them look more similar. $f(x) = 2^{\lg f(x)}$. So, $n^{\frac{\sqrt{n}}{2}} = 2^{\lg n^{\frac{\sqrt{n}}{2}}} = 2^{\frac{\sqrt{n}}{2} \lg n}$. Now, if we take the ratio of these two functions: $\frac{2^n}{2^{\frac{\sqrt{n}}{2} \lg n}} = 2^{n - \frac{\sqrt{n}}{2} \lg n}$. $n - \frac{\sqrt{n}}{2} \lg n \in \Theta(n)$; so, $2^{n - \frac{\sqrt{n}}{2} \lg n}$ goes to infinity, and 2^n dominates our big nasty function. Just for fun, note that that means our big nasty function is **sub-exponential** yet **super-polynomial**.

2 Functions/Orders of Growth for Code

Give and briefly **justify** good Θ bounds on the worst-case running time of each of these pseudocode snippets dealing with an array A of length n. Note: we use 1-based indexing; so, the legal indexing of A is: $A[1], A[2], \ldots, A[n]$.

SOLUTION NOTES: I've annotated the pseudocode below with notes and put my conclusions on worst-case running time below. Note: any line of code that is executed takes $\Omega(1)$ time (i.e., at least some constant amount of time to process); so, anything I annotate as $O(1)$ is in $\Theta(1)$.

Finding the maximum in a list:

```
Let max = -infinity          O(1)
For each element a in A:     n iterations
```

```
  If max < a:                    O(1)  could make this true often, but with..
    Set max to a                 O(1)  no asymp diff to loop body runtime anyway
Return max                     O(1)
```

SOLUTION: n iterations of a loop whose body takes constant time to run—regardless of the conditional contained inside—plus some constant-time setup and teardown. Overall, this is $\Theta(n)$.

"Median-of-three" computation:

```
Let first = A[1]                           O(1)
Let last = A[n]                            O(1)
Let middle = A[floor(n/2)]                 O(1)

If first < last And first < middle:        O(1)
  return first                               O(1)
Else If middle < first And middle < last:  O(1)
  return middle                              O(1)
Else:                                      O(1)
  return last                                O(1)
```

SOLUTION: Nothing exciting happening here. Every single operation is $O(1)$; so, it doesn't matter what happens in the conditional, the whole thing takes $\Theta(1)$ time.

Counting inversions:

```
Let inversions = 0                     O(1)
For each index i from 1 to n:          n iterations
  For each index j from (i+1) to n:      n-i iterations
    If a[i] > a[j]:                        O(1)
      Increment inversions                   O(1)
Return inversions                      O(1)
```

SOLUTION: The inner loop takes constant time per iteration. I've seen many times the pattern of $n - i$ iterations of an inner loop within an outer loop that lets i range over n values. So, I might jump to saying this is $\Theta(n^2)$, but let's get there step by step.

We can express the runtime as a sum:

$$\begin{aligned}
\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n}(n-i) \\
&= (\sum_{i=1}^{n} n) - (\sum_{i=1}^{n} i) \\
&= n^2 - \frac{n(n+1)}{2} \\
&= n^2 - \frac{n^2+n}{2} \\
&= n^2/2 - n/2 \\
&\in \Theta(n^2)
\end{aligned}$$

It's interesting to note also that in the worst case, where the array is in reverse-sorted order, we increment `inversions` $\Theta(n^2)$ times as well, which means there can be $\Theta(n^2)$ inversions in an array of length n. We'll end up coming back to that.

3 Progress Measures for While Loops

Assume that `FindNeighboringInversion(A)` consumes an array `A` and returns an index `i` such that `A[i] > A[i+1]` or returns `-1` if no such inversion exists. Let's work out a bound on the number of iterations of the loop below in terms of n, the length of the array `A`.

```
Let i = FindNeighboringInversion(A)
While i >= 0:
  Swap A[i] and A[i+1]
  Set i to FindNeighboringInversion(A)
```

1. **Give and work through two small inputs** that will be useful for studying the algorithm. (What is "useful"? Try to find one that is simply common/representative and one that really stresses the algorithm.)

   SOLUTION: Let's try this as a "representative" input `[8, 2, 3, 9, 7]`. And, how about this to stress the algorithm, because it has the maximum possible number of inversions (each pair of elements is inverted with respect to each other) `[4, 3, 2, 1]`?

   The algorithm is "non-deterministic" in the sense that it doesn't specify which inversion to select at each step. I'll try the leftmost and think back to this decision as I solve future parts, in case it's important.

   Then, for the first example:

   ```
   [8, 2, 3, 9, 7]
   [2, 8, 3, 9, 7]
   [2, 3, 8, 9, 7]
   [2, 3, 8, 7, 9]
   [2, 3, 7, 8, 9]
   ```

   That happens to be the same number of steps as the number of inversions (4) in the input.

   For the second:

   ```
   [4, 3, 2, 1]
   [3, 4, 2, 1]
   [3, 2, 4, 1]
   [2, 3, 4, 1]
   [2, 3, 1, 4]
   [2, 1, 3, 4]
   [1, 2, 3, 4]
   ```

   Again, that happens to be the same number of steps as the number of inversions (6)! (Cool, but perhaps totally irrelevant.)

2. **Define an inversion** (not just a neighboring one), and **prove that if an inversion exists at all, a neighboring inversion exists**.

   SOLUTION: Well, here's the definition of a neighboring inversion "an index `i` such that `A[i] > A[i+1]`". The neighboring part is because we compare index `i` to index `i+1`. So, let's let it be any two indexes, but insist their elements are out of order with respect to the indexes: "indexes `i` and `j` where `i < j` but `A[i] > A[j]`".

   Let's double-check that that makes sense. It means one element comes before another in the array and yet is larger, which seems reasonable. It applies to the 8 and 3 in our first small example but not to the 8 and 9. It applies to every pair of elements in our second example.

3. Give **upper- and lower-bounds on the number of inversions** in A.

   SOLUTION: By our definition, an inversion requires two distinct elements of A. There are $\binom{n}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$ pairs. Can they all be inverted? In our second example, we do achieve $\frac{4^2}{2} - \frac{4}{2} = 8 - 2 = 6$ inverted pairs. If we extend our second example to an array of length n in reverse-sorted order, then any pair of elements `A[i]` and `A[j]` with `i < j` will indeed be out-of-order with respect to each other.

   So, the upper-bound is $\binom{n}{2} \in O(n^2)$.

   An already-sorted array will have no inversions. So, the lower bound is 0.

4. Give a "measure of progress" for each iteration of the loop in terms of inversions. (I.e., how can we measure that we're making progress toward terminating the loop?)

   SOLUTION: In our examples above, we took exactly as many steps as there were inversions. That's because each swap patched an inversion without "disturbing" any others (because all elements besides the two swapped were still in the same order with respect to each other).[1] So, one measure of progress would be the number of remaining inversions, which goes down by 1 at each iteration.

5. Give an upper-bound on the number of iterations the loop could take.

   Since we reduce the number of inversions by 1 in each iteration and the number of inversions is at most $\binom{n}{2} \in O(n^2)$, then that is also an upper-bound on the number of iterations of the loop.

   (Note that the loop ends when the number of inversions is equal to zero. This may be non-obvious, since it's not immediately clear that *an inversion implies a neighboring inversion.* One good, simple argument of this point is: if there are no neighboring inversions, then the array is clearly in sorted order; so, there are no inversions. We can also make an inductive argument (on the "width" of the inversion) that this is true. If an inversion isn't already neighboring, consider the index `j-1`. Since the inversion isn't neighboring, `i < j-1 < j`. If `A[i] > A[j-1]`, then by induction there's a neighboring inversion. Otherwise, `A[i] < A[j-1]` and we know `A[i] > A[j]`; so, `A[j-1] > A[j]`, and there's a neighboring inversion at `j-1`.[2])

6. Prove that this algorithm sorts the array A (i.e., removes all inversions from the array).

   SOLUTION: The loop terminates in a finite number of steps with no inversions (including neighboring inversions), which is exactly the characteristic of a sorted array.

[1] Note that this is true regardless of the arbitrary choice we made to address the leftmost neighboring inversion first.

[2] I'm assuming distinct elements, but the proof proceeds essentially the same with duplicates allowed.