big-Θ, big-O, and big-Ω: being precise about imprecision

COMP 550.002

January 27, 2018

Reading for Jan 29–31:
ClRS 3 (skimming 3.2) with the caveats below,
start of 4, then 4.3-4.4. Skim 4.2 & 4.5.
Quicksort 7 through 7.3.

After this week's reading and lectures, you should be able to
define big-Θ, big-O, and big-Ω, and apply them to the worst- and best-case running times of algorithms, including Binary Search, Insertionsort, Mergesort and Quicksort, and
convincingly present the solution of recurrence relations for these and other divide and conquer algorithms.

CLRS is encyclopedic, but at some places in chapters 3–4 it is not as precise or clear as I would like.

1. Definitions of big-Θ, big-O, and big-Ω, p. 44-48, should put all their quantifiers first.

   Here is how I write the **definition of big-Θ**:

   $$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0, \forall n \geq n_0, (0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}.$$

   That is, you choose the positive constants, which must work for all $n \geq n_0$.

   Placing all quantifiers first makes the definition easier to negate. Function $f(n)$ **is not in** $\Theta(g(n))$ iff, no matter what $c_1, c_2 > 0$ are chosen, you can find arbitrarily large n so that either $c_1 g(n) > f(n)$ or $f(n) > c_2 g(n)$. That is, $f(n)$ never becomes trapped between $c_1 g(n)$ and $c_2 g(n)$.

2. On p.48–49, big-Θ, big-O, and big-Ω are defined for functions of n, but applied to "running times," which are functions of instances, and not of n.

   It makes sense to define the **running time of an algorithm** A **on input instance** I as $A(I)$, and define a **size of an instance** as an integer $|I|$ as the number of bits or numbers or pointers to write down the instance. The **worst-case running time** $A_{worst}(n) = \max_{|I|=n} A(I)$ and **best-case running time** $A_{best}(n) = \min_{|I|=n} A(I)$ do become functions of n. (If we have a probability distribution ρ on instances, we can also define average-case running time $A_{\overline{\rho}} \sum_{|I|=n} \rho(I) A(I)$.)

   CLRS admits, "Technically it is an abuse to say that the running time of insertion sort is $O(n^2)$, since for a given n the actual running time varies depending depending on the particular input of size n." But then they ask in exercise 3.1-6 (p.53) to "Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$." This is not possible without a definitions of Θ for "running times." (I think their statement about O and Ω is a reasonable definition for **Algorithm** A **runs in** $\Theta(n)$ **steps**.)

3. They sometimes state recurrences with asymptotic (big-O or big-Θ) notation (p.66). By definition these say nothing when $n \leq n_0$ for some chosen constant n_0.

   I find CLRS p.79 misleading when it says, "although asymptotic notation subsumes constant multiplicative factors, recursive notation such as $T(n/2)$ does not." Asymptotic notation hides constant factors (since they depend on the implementation anyway), but these need to be made explicit in a recurrence, even if we use lazy binding to avoid picking values until the end of our analysis.

Recurrence relations should ideally be stated without asymptotic notation. They should **always** be solved by replacing asymptotic notation with particular chose constants. We'll work through many examples this week.

4. One final one: Permutations are defined for sets (B.3, p.1167), but most used for sequences, first on page 16 with insertion sort.

   The CLRS definition for sets is brief: "A bijection from a set A to itself is sometimes called a permutation." What it means is that our function $\sigma\colon A \to A$ sends every element of A to exactly one element of A. In logic we have to break this up into σ being surjective (aka onto), $\forall a' \in A, \exists a \in A, (f(a) = a')$, and injective (aka one-to-one, or more accurately, two-to-two), $\forall a, b \in A, (f(a) = f(b) \longrightarrow a = b)$. For finite sets, either one of these is enough, because of the pigeonhole principle. For example, one of the six permutations of $\{1, 2, 3\}$ is $f(1) = 2$, $f(2) = 1$, $f(3) = 3$.

   A *permutation of a sequence* $(a_1, a_2, \ldots, a_n)$, to CLRS, is a permutation of the set of indices $\sigma\colon 1..n \to 1..n$ to give the new sequence $(a_{\sigma(1)}, a_{\sigma(2)}, \ldots, a_{\sigma(n)})$. Permutation σ tell where the value at each original index should go. There is an inverse permutation $F = \sigma^{-1}$ that tells where each final value comes from. You can get σ from F easily: for $i = 1..n$, $\sigma[F[i]] = i$.

   See Wikipedia for more on permutations: `http://en.wikipedia.org/wiki/Permutation`

Insertion sort(A)
Input: An array $A[1..n]$ of n numbers
Output: Values of $A[1..n]$ permuted so $A[1] \le A[2] \le \cdots \le A[n]$, and the *from indices* $F[1..n]$. The element ending in A at index i was originally from $F[i]$.
Inv 0: $F[1..n]$ are maintained so $F[k]$ is the original position in A of the item currently at k.
Inv 1: At 1 with loop value j, the values $A[1] \le A[2] \le \cdots \le A[j-1]$ originally came from $F[1..j-1]$, which is a permutation of $1..j-1$ and $A[j..n]$ and $F[j..n]$ are untouched.
Inv 2: At 2, $A[1..i]$ is in sorted order and *key*, the former value of $A[j]$, is $<$ former values of $A[i+1..j-1]$, which are now moved to $A[i+2..j]$, as reflected in F.

```
   for j = 1 to A.length, F[j] = j; // Initialize F
1: for j = 2 to A.length;
       key = A[j]; i = j − 1;
2:     while i > 0 & key < A[i]
           A[i + 1] = A[i]; // Moved from i to i + 1, so
           F[i + 1] = F[i]; i = i − 1; // save orig pos
       endwhile
       A[i + 1] = key; // As we store key in A,
       F[i + 1] = j;          // Save where key is from in F.
   endfor
```

Algorithm 1: Modified insertion sort of $A[1..n]$ records in $F[1..n]$ where elements were originally.

To show that insertion sort really makes a permutation, we could keep track of where each element comes from. We add this to the invariant. Since F points to the original data location, we could avoid moving the data in A if we compare the key to $A[F[i]]$. Invariants help us do this correctly.