ADTs (now in Chapter 7 but harking back to Chapter 6)

An **Abstract Data Type** (**ADT**) is more than just a "data object" or some objects combined into a "data structure". It also specifies:

1. The operations provided to clients (end-clients and/or library coders) for building and manipulating the objects;
2. Logical requirements and guaranteed properties of those operations; and
3. Efficiency specifications (at least relative) for those operations.

The text only mentions point 1 when it defines ADTs in its short section 7.1, but it addresses point 3 right away, and it covers point 2 first at the level of **unit testing** of individual pieces of code (section 7.7). I tend to approach point 2 first at the level of the whole ADT analytically, per what is called design by contract. Here is a small example that is reflected accurately by the text's little box on "Comparators" on page 143 in chapter 4. Scala inherits from Java an ADT called `Comparator[T]` that requires the parametric type `T` to implement < and = comparisons. It provides an operation `x.compareTo(y)` that is basically always implemented to have the results

$$x.compareTo(y) \ = \begin{cases} +1 & if\ x\ >\ y \\ 0 & if\ x\ =\ y \\ -1 & if\ x\ <\ y \end{cases}$$

However, the **contract** only guarantees that the value is a positive integer if $x\ >\ y$ and a negative integer if $x\ <\ y$, not the specific integers $+1$ and -1. You may be able to see this, including the Java `@Contract` attribute, if you write a line of code with (say) `true.compareTo(false)` and hover over the `compareTo` (or right-click it?) in your IDE. It is common---but risky---to use bexp`.compareTo(false)` as a way of converting a Boolean expression to have value 0 when false and $+1$ if true.

The one thing that an ADT is not supposed to specify is *how* the data type is implemented. That is the "Abstract" aspect. The library designer must be free to change the implementation, so long as nothing in the API or contract or complexity specs changes. (All three can be enhanced.) Achieving certain promised combinations of logic and performance for a given API, however, often dictates much of the implementation. *That correspondence is the main subject of this course.*

Set and Map as ADTs

The three most basic operations provided by all kinds of `Set` objects are *creating* sets by various means, *quick lookup* of individual objects (i.e., testing whether a given object is in the set) and *traversing* through all objects in the set. A Mutable Set adds operations to modify a Set after it is created by adding and/or removing elements. It may---or may not---share implementation details with the basic (immutable) Set. One key logical property in the contract for a Set is:

- A `Set` never has two copies of the same value. If the argument type `A` (which I will also call the "client type" or the "foreground type") is a *value type*, it automatically has an `equals` comparison defined, and the == operator calls it. Then the logical requirement "x is not the same as y" for any two elements x,y in the Set is implemented as `!x.equals(y)`.

If `A` is a reference type, things may default to reference equality. For a particular argument type such as `String`, you can always put it in a "wrapper" class where you write your own (override of the default) `equals` method. For instance (`.../ScalaSamples/Sets.scala`, modified a little from https://stackoverflow.com/questions/7681183/how-can-i-define-a-custom-equality-operation-that-will-be-used-by-immutable-set)

```
case class MyString(val name: String) {     //example of "composition not inheritance"

  override def equals(other: Any) = other match {     //KWR: match used as a type-test
    case that: String => that.name.equalsIgnoreCase(this.name)
    case _ => false
  }
  override def hashCode = name.toUpperCase.hashCode
}

val mySet = Set(MyString("conservative"), MyString("conversative"), MyString("Conservative"))
mySet: ...immutable.Set[MyString] = Set(MyString("conservative"), MyString("conversative"))
```

Here are some further ADT properties of interest:

- If the foreground type `A` in `Set[A]` has < as well as ==, then is a traversal guaranteed to be in a totally sorted order?
- If not, can we at least say that if a `Set` object `foo` is created *the same way* by two different lines of code (or the same line at two different times in the execution, or by two different threads at the same time), which then traverse the set, will the two transversals give the same sequence of elements?

A **Map** is actually just a more useful kind of Set that returns a value when you successfully look up an element. So `Map[A,B]` gives values of type `B` when you find an element of type `A`. The key logical property of a Map is that **the same element always gives the same value**. You can emulate a `Set[A] s` via a `Map[A,Boolean] f` such that for all objects `a`, if `s.contains(a)` then `f(a) = true`, else `f(a) = false`.

The one downside of doing this is that with the Map, you have to store all the `false` values for the *non-elements*, whereas with the Set, you can ingore non-elements entirely. Partly for this reason, Scala actually builds-in the parenthesis syntax you'd expect for maps directly into the `Set` API, so you can write `s(a)` instead of `s.contains(a)`. The parens in turn are short for what is formally the `apply` method. [There is actually an internal difference in how `s.contains(a)` and `s(a)` are implemented in the Scala library, but we can ignore it.] Apart from this difference, implementations of Map follow almost all the same issues as implementationf of Set.

Time Complexity and O-Notation

The number n will generally stand for "the total number of unit-size data points." When we have a single main container, such as a List or Array or Set or Map, we identify n with the container's `.size`. Even when the individual data objects have varying sizes, such as with `String` lines of different lengths, this is still often an accurate assumption (maybe blank lines offset long ones).

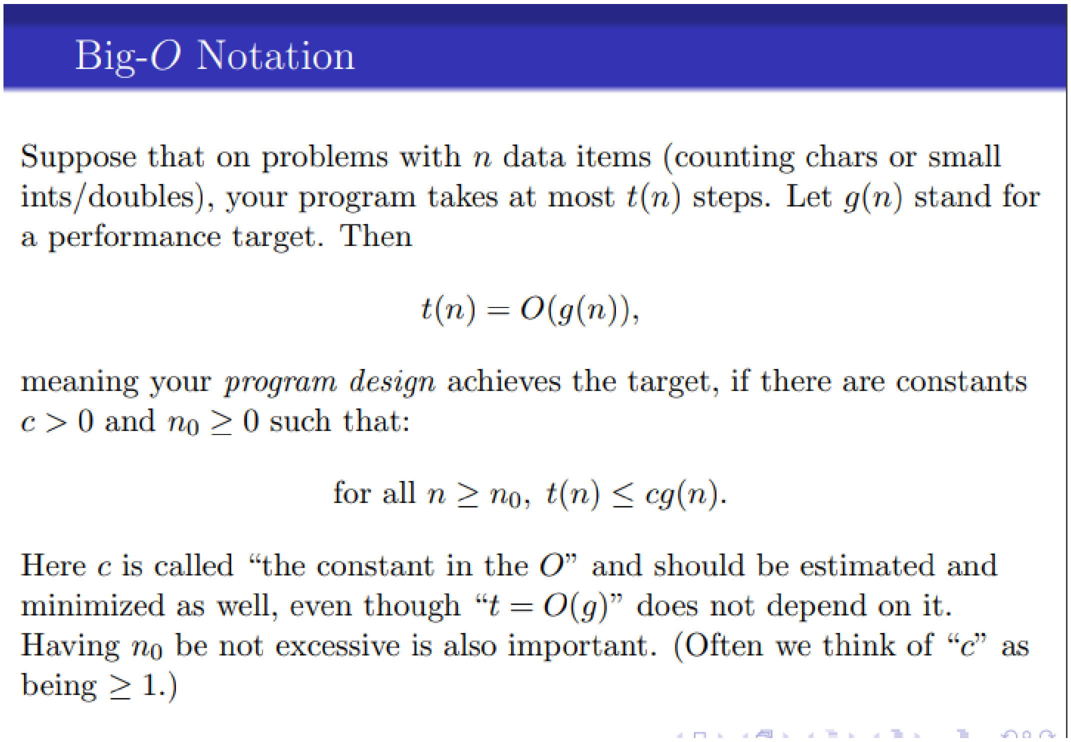
Sometimes "n" is sub-divided. In our case, we have an array of m lines, where each line is a list of (variable) length ℓ. We can think of n as approximately equal to $m \times \ell$ here, but part of the reality with a long file like `JustHamlet.txt`---where all the lines are cut off at 80 chars---is that $\ell \ll m$. Depending on the context, we might regard m as being the "n". One thing for sure: if you use operations on the lines that collectively take time proportional to m^2, it will be a bigger time hit than if you do operations within each line that take time proportional to ℓ^2.

We also have a dictionary of size $N = 460,000$ words about. If the best we could say was that each lookup of a word took time of order N, things could get slow. But happily the `Set` data structure (in Scala) fulfils a contract that the lookup time is of order "vanishingly less than" N.

The concepts "time at most order-of", "time proportional to", and "vanishingly smaller than" are necessarily rough. We can, however, give a precise mathematical definition of them in a way that incorporates their roughness:

The key definition is: Given two numerical functions $f(n)$ and $g(n)$,

- $f(n) = O(g(n))$ if there are constants c and n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.
- $f(n) = o(g(n))$ if the limit of $f(n)/g(n)$ goes to 0 as n goes to infinity.



Big-O Notation

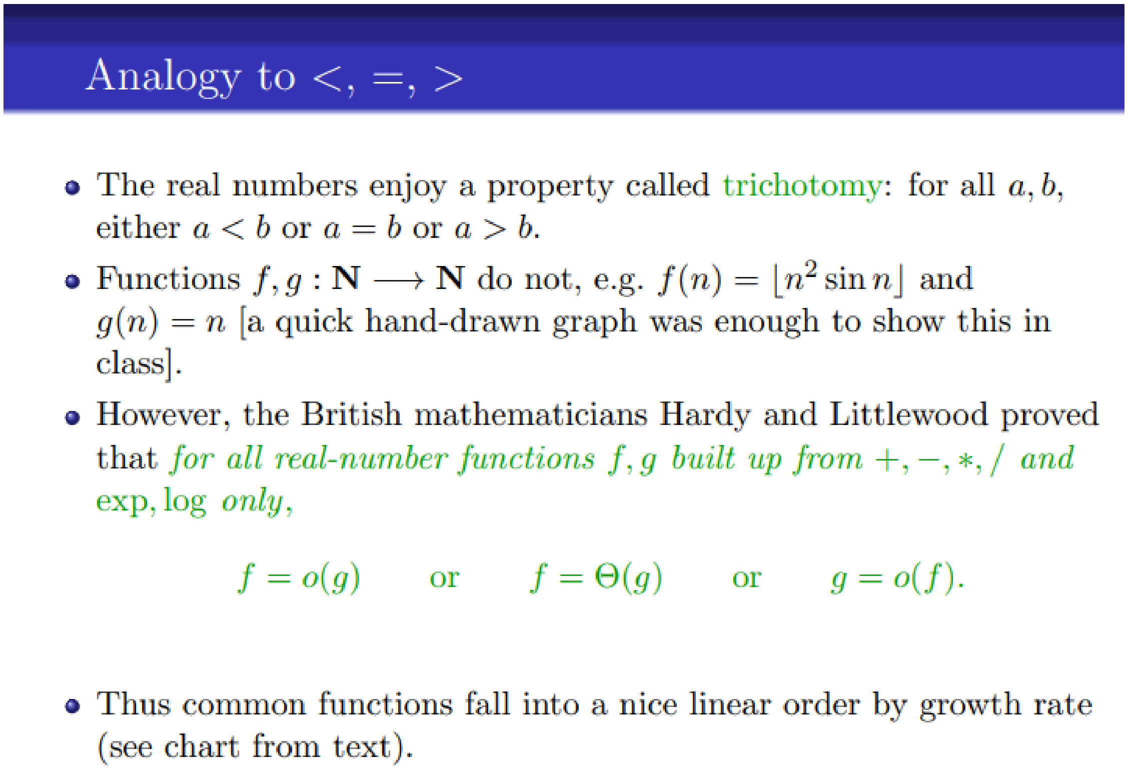
Suppose that on problems with n data items (counting chars or small ints/doubles), your program takes at most $t(n)$ steps. Let $g(n)$ stand for a performance target. Then

$$t(n) = O(g(n)),$$

meaning your *program design* achieves the target, if there are constants $c > 0$ and $n_0 \geq 0$ such that:

$$\text{for all } n \geq n_0,\ t(n) \leq cg(n).$$

Here c is called "the constant in the O" and should be estimated and minimized as well, even though "$t = O(g)$" does not depend on it. Having n_0 be not excessive is also important. (Often we think of "c" as being ≥ 1.)



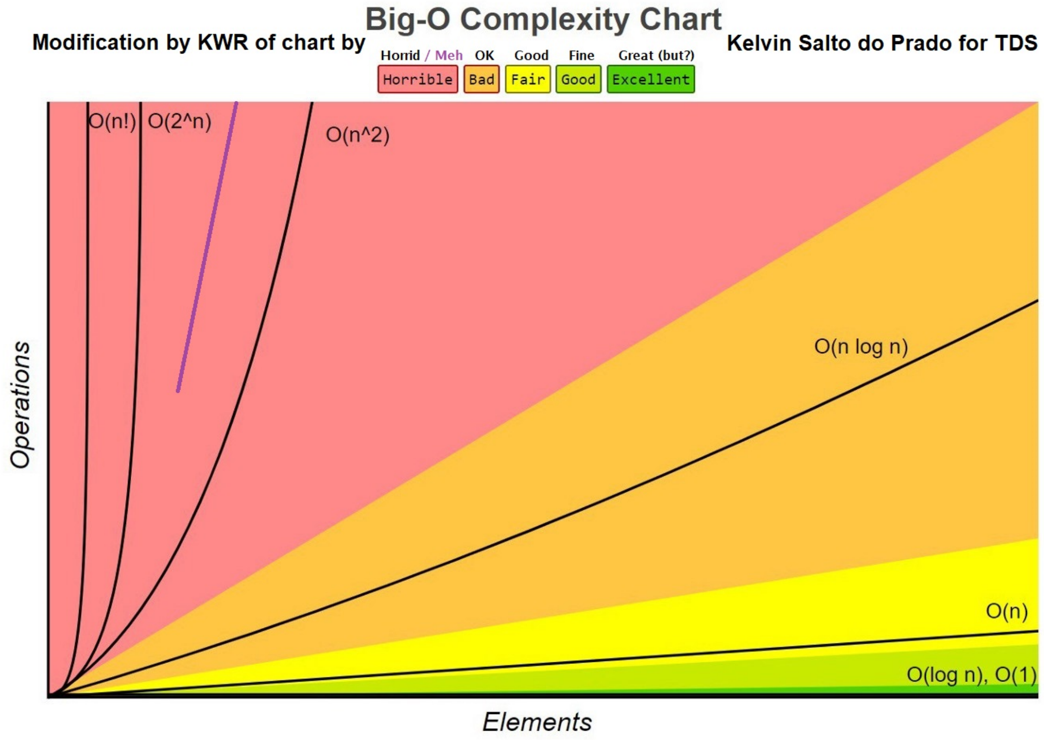
Analogy to $<, =, >$

- The real numbers enjoy a property called trichotomy: for all a, b, either $a < b$ or $a = b$ or $a > b$.
- Functions $f, g : \mathbf{N} \longrightarrow \mathbf{N}$ do not, e.g. $f(n) = \lfloor n^2 \sin n \rfloor$ and $g(n) = n$ [a quick hand-drawn graph was enough to show this in class].
- However, the British mathematicians Hardy and Littlewood proved that *for all real-number functions* f, g *built up from* $+, -, *, /$ *and* $\exp, \log$ *only*,

$$f = o(g) \qquad \text{or} \qquad f = \Theta(g) \qquad \text{or} \qquad g = o(f).$$

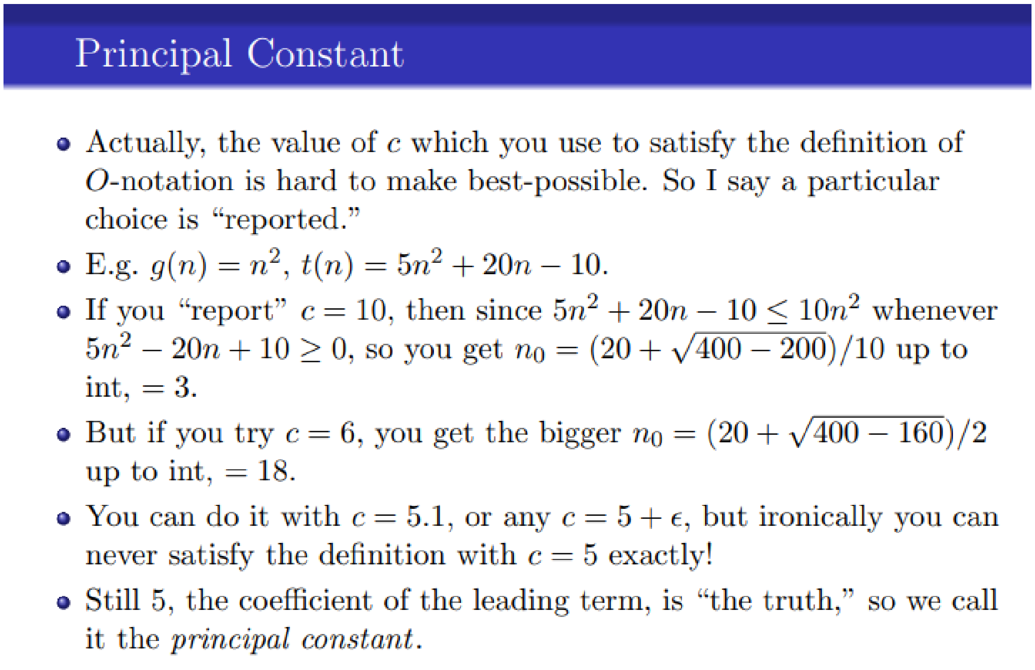
- Thus common functions fall into a nice linear order by growth rate (see chart from text).

That referred to my C++ text in 2014. Here is a mod of the chart I showed in class instead:



Big-O Complexity Chart

Modification by KWR of chart by Kelvin Salto do Prado for TDS



Principal Constant

- Actually, the value of c which you use to satisfy the definition of O-notation is hard to make best-possible. So I say a particular choice is “reported.”
- E.g. $g(n) = n^2$, $t(n) = 5n^2 + 20n - 10$.
- If you “report” $c = 10$, then since $5n^2 + 20n - 10 \leq 10n^2$ whenever $5n^2 - 20n + 10 \geq 0$, so you get $n_0 = (20 + \sqrt{400 - 200})/10$ up to int, $= 3$.
- But if you try $c = 6$, you get the bigger $n_0 = (20 + \sqrt{400 - 160})/2$ up to int, $= 18$.
- You can do it with $c = 5.1$, or any $c = 5 + \epsilon$, but ironically you can never satisfy the definition with $c = 5$ exactly!
- Still 5, the coefficient of the leading term, is “the truth,” so we call it the *principal constant*.

My lecture then covered extensive examples of curves, tradeoffs, and the role of the principal constant in the graphs of Jim Marshall from a course at Sarah Lawrence: http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/index.html