Counting The Number Of Operations, Reading

What led us to suspect duplicate checking in the DVC schedule history solution, instead of sorting, was a math calculation we did. Without even bothering with details like how long (how many CPU cycles) it takes to do a compare vs doing a swap, or objects vs concatenated strings, we came up with a number -- it was 2,450,000,000. What we *didn't* do was to write this as a formula, with the data set size as a variable -- that's what would have helped us see how the runtime would be expected to scale with data set size.

Data Set Size, n

The variable name that is typically used to represent the data set size is "n". In the case of the DVC schedule, n is about 70,000. So our math should have been something like this:

1. There are n values that we have to compare a term-section combination to values in a list.
2. The list size starts at zero and ends at about n, for an average list size of about n/2.
3. There are just a few duplicates, so just about *every* value has to go through the entire list to prove that it's not a duplicate.

That's:

`n * (n/2) compare cycles` or $n^2/2$ or $f(n) = \frac{n^2}{2}$

...where **f(n) is a function in n** that expresses how many compare cycles it takes to check for duplicates. Each compare cycle involves a compare of values with ==, an increment of a loop counter, and a compare of a loop counter. If you look inside the value comparison, of you're comparing strings (whether as attributes in a struct or concatenated), there's a library function that's comparing the chars in the strings until a mismatch is found or until the end of the string is reached. So there's really some sort of **constant multiplier** that should be applied to the formula, if f(n) is to represent the number of actual CPU operations being performed. That's tough to even estimate, because how can we know how many chars on average will be compared in string compares? The best we can do is some multiplier -- call it "c".

$$f(n) = c \cdot \frac{n^2}{2}$$

The Complete Picture, f(n)

But there's more to the solution for the whole program. There's the opening of the file, the reading and parsing of its lines, and the sorting of the subject-count objects. Let's look at these:

Opening the file does take an indeterminate number of operations, but it's not dependent on the data set size. So that contributes another constant -- call it c_1.

Reading a line and parsing it also takes a hard to calculate number of operations per line, and there are n lines to process. So that contributes $c_2 \cdot n$, with yet another constant.

The sorting involves nested for-loops that each have about 100 cycles -- that's the number of subject codes in the data set. It's not strongly dependent on n, if n is really large. The inner loop averages 50 cycles, starting at about 100 and ending at 1. The outer loop has 100 cycles. So that's 5,000 compares, total. If half the time there are swaps to do, that's 2,500 swaps (guessing here...). So this contribution is $c_3 \cdot 5000 + c_4 \cdot 2500$ with even more constants. Put all these together we get this:

$$f(n) = c \cdot \frac{n^2}{2} + c_1 + c_2 \cdot n + c_3 \cdot 5000 + c_4 \cdot 2500$$

Simplifying The Formula For Large n

That does not even account for the loop to output the results, and already it's complicated. Plus it's full of estimates and unknown constant multipliers. As it stands, it's **not real easy to** use this to (1) **communicate** the performance characteristics of our solution to other programmers, or to (2) **compare** its scaling characteristics to alternative solutions.

But **there *is* useful information** in there. So what programmers have done is to devise a way to take formulae like the one we developed above, and express them in a more meaningful way that can be used to communicate and compare. The first step is to simplify the formula.

Notice that our formula is a polynomial, which is typical for such formulae. Recognizing that the only reason we're interested in doing this at all is the scaling issue when n is large, we can make one really big simplification -- *drop the "lower order polynomial terms"*. The n^2 term will dominate at some point as n becomes large. So we simply ignore the n^1 and n^0 terms, and we're back to:

$$f(n) = c \cdot \frac{n^2}{2}$$

But wait -- there's more. "c" is already an indeterminate constant, so there's no reason for the divisor "2". Indeterminate divided by 2 is still indeterminate. So we're at $c \cdot n^2$ at this point -- and we're still not quite done.

What we *really* care about is how the function *varies* with n, so the constant is inconsequential -- drop it, and we get:

$$f(n) \sim n^2$$

Nested For-Loop Sorting

In our solution, the sorting was not dependent on the data set size -- the number of cycles was dependent on some other number. But in standard nested for-loop sorting, where the loop limit n *is* the data set size, we have this code:

```
for (int i = 0; i < n; i++)
  for (int j = i + 1; j < n; j++)
    if (a[j] < a[i])
      swap(a[i], a[j]);
```

Applying what we learned above, the inner loop limit averages n/2 cycles, and it's done n times. That's $n^2/2$ object compares, plus some percentage of that involve swaps."j" is compared and incremented $n^2/2$ times. "i" is compared and incremented n times. You can see where this is going, either by writing it out or by reasoning it out -- it's $n^{2:}$

$$f(n) \sim n^2$$

Searching An Array Or Linked List

Given a data set of size n, and a for-loop to look at each value and check for a match, we start with this code:

```
for (i = 0; i < n; i++)
  if (a[i] == matchThisValue)
    break;
```

On the average, the match will be found half-way through, so that's n/2 compares. Ignoring any constant multiplier to account for how many CPU operations actually get done, we get:

$$f(n) \sim n$$

Searching An Array Or Linked List, Best Case

But sometimes we're lucky -- there are what we call "best cases". In searching, the best case is that we find the match right away! That formula has no n in it at all -- just a constant. That's expressed as:

$$f(n) \sim 1$$

Something to think about -- is there a best case for nested for-loop sorting? If so, what is it and how would the formula change (if at all)?

Searching An Array Or Linked List, Worst Case

On the other hand, the worst case is that we don't find a match -- that's what happens most of the time in our duplicate checking. For that we have to do n compares every time, instead of an average of n/2. So how does that affect its scaling? It's still this:

$$f(n) \sim n$$

It may take twice as long -- that's where the constant multiplier comes in -- but *it scales exactly the same way!*

Scaling Is Not About The Multiplier

It's not to say that faster processors and more efficient coding are not important to performance -- they certainly are. But in terms of evaluating and comparing how different *algorithms* scale, these factors are ignored. It's all about the algorithm.