Musing Mortoray

EFFICIENCY

What is amortized time?

ON 2014-08-11 • (1 COMMENT)

Amortized time is often used when stating algorithm complexity. Instead of giving values for worst-case performance it provides an average performance. This is appropriate in many domains, but we must be careful. This article is a short introduction to the topic of amortized analysis.

The aggregate method

Amortized time looks at an algorithm from the viewpoint of total running time rather than individual operations. We don't care how long one `insert` takes, but rather the average time of all the calls to `insert`.

$$c'_i = \frac{1}{N} \sum_{i=1}^{N} c_i$$

The amortized cost of an insert equals the sum of all insert costs divided by the number of inserts. It's the arithmetic mean cost.

For example, let's assign our cost for a data collection to be the number of times an object has to be copied. We do 100 inserts on the collection and measure 500 total copy operations. That gives us an amortized cost of 5. It's quite possible that most inserts do only 1 copy, but that some of shuffle things around doing 10 or more copies.

Applicability

Amortized cost analysis is good when we're trying to understand a series of operations, in particular a large number of them. As programmers we often deal with data in bulk and perform batch operations. The time it takes to perform just one operation is of little significance when we're doing it thousands of times.

The caveat here is that amortized costs can hide performance issues. It's nice to think of some collections as offering amortized `O(1)` insertion, but often a `O(N)` operation is hiding, waiting to pounce on the unsuspecting caller. This can often be the root cause of inexplicable spikes in response time. Put that one long operation in a synchronized section and suddenly all other threads are blocked as well.

This problem doesn't diminish the value of amortized analysis, it just means we need to be attentive. It's important to know whether a big O cost is the amortized, or the maximum cost. Quite often programming literature fails to identify amortized costs. Hash maps are often victims here, with insertion labelled simply as `O(1)`, rather than an *amortized* `O(1)` with a proper limit of `O(N)`.

If we dig deeper into performance analysis we'll also find the aggregate method to be a bit limiting. Fortunately we have two extra tools at our disposable. We'll take a brief look at those now.

The accounting method

Amortized cost can also be treated as a form of credits and debits. This is called the accounting method. Each time you perform an operation you put a credit into the machine. The machine uses up these credits while performing the operation. The amortized cost is the amount of credit needed for each operation such that the balance never falls below zero.

Consider the aggregate example, where 100 inserts resulted in 500 copies. We can say the insert costs 5 credits, where each copy will use up 1 credit. Say the first five inserts each do one copy, so they insert 5 credits and use up 1, leaving 20 credits in the machine. Now the sixth operation has the 20 balance, plus 5 new credits, so it can perform up to 25 copies to remain above 0.

The accounting method requires us to determine how much credit and debit is required per operation. It is a different way of looking at the problem. It makes it easier to reason about individual amortized operations and sequences of different operations, like `insert` and `remove`.

The potential method

A third way to look at this problem is from a view of potential. Instead of maintaining an explicit credit balance, the structure of the system itself encodes a virtual balance: the potential. The amortized cost of an operation is the amount it increases this potential.

$$c_i' = c_i + \phi(D_i) - \phi(D_{i-1})$$

The amortized cost equals the actual cost plus the difference in potential.

An intuitive example here is with maps and trees. We know that as you insert into a tree you will occasionally need to rebalance it. Rebalancing can be a costly operation, for a red-black tree this is `O(log n)`. But we also know that after rebalancing the tree is in a better state. The next rebalancing will not require as many steps. For a red-black tree this results in an amortized time of `O(1)`. In terms of potential, the basic inserts increase the potential and the rebalancing decreases the potential.

Like the accounting method this is just another approach to amortized analysis. It doesn't matter what approach you take, the result should be the same. One method may however be easier to use than the other.

> If you're looking for more details, like how to actually perform this analysis, search for the term "amortized analysis". I've found this explanation (https://duckduckgo.com/l/?kh=-1&uddg=http%3A%2F%2Fwww.cs.princeton.edu%2F~fiebrink%2F423%2FAmortizedAnalysisExplained_Fiebrink.pdf) by Rebecca Fiebrink to be quite helpful.

I'm a programmer who has worked on many high performance systems, from trading systems, to responsive web servers, to video games. Follow me on Twitter (http://twitter.com/edaqa) to hear more about my work. If your project needs that extra speed boost, or a reduction in resources, then don't hesitate to enlist my help (https://mortoray.com/about/?src=perf).

1 reply »

Pingback: Algorithm Analysis: amortized time | Revo