Amortized complexity in layman's terms?

Can someone explain amortized complexity in layman's terms? I've been having a hard time finding a precise definition online and I don't know how it entirely relates to the analysis of algorithms. Anything useful, even if externally referenced, would be highly appreciated.

algorithm amortized-analysis

edited Feb 26 '13 at 0:45
Rafał Rawicki
15.2k ● 3 ● 44 ● 68

asked Feb 26 '13 at 0:41
Bob John
1,325 ● 6 ● 27 ● 47

2 en.wikipedia.org/wiki/Amortized_complexity – Oliver Charlesworth Feb 26 '13 at 0:44

stackoverflow.com/questions/11102585 stackoverflow.com/questions/12659931 stackoverflow.com/questions/14002391 programmers.stackexchange.com/questions/161404 – BlueRaja - Danny Pflughoeft Feb 26 '13 at 16:33

6 Answers

Amortized complexity is the total expense per operation, evaluated over a sequence of operations.

The idea is to guarantee the total expense of the entire sequence, while permitting individual operations to be much more expensive than the average.

Example:
The behavior of C++ `std::vector<>`. When `push_back()` increases the vector size above its pre-allocated value, it doubles the allocated length.

So a single `push_back()` may take `O(N)` time to execute (as the contents of the array are copied to the new memory allocation).

However, because the size of the allocation was doubled, the next `N-1` calls to `push_back()` will each take `O(1)` time to execute. So, the total of `N` operations will still take `O(N)` time; thereby giving `push_back()` an amortized cost of `O(1)` per operation.

Unless otherwise specified, amortized complexity is an asymptotic worst-case guarantee for any sequence of operations. This means:

- Just as with non-amortized complexity, the big-O notation used for amortized complexity ignores both fixed initial overhead and constant performance factors. This means, for the purpose of evaluating big-O amortized performance, you can generally assume that any sequence of amortized operations will be "long enough" to amortize away a fixed startup expense. Specifically, for the `std::vector<>` example, this is why you don't need to worry about whether you will actually encounter `N` additional operations: the asymptotic nature of the analysis already assumes that you will.
- Besides arbitrary length, amortized analysis doesn't make assumptions about the sequence of operations whose cost you are measuring -- it is a worst-case guarantee on *any possible sequence* of operations. No matter how badly the operations are chosen (say, by a malicious adversary!), an amortized analysis must guarantee that a long enough sequence of operations may not cost consistently more than the sum of their amortized costs. This is why (unless specifically mentioned as a qualifier) "probability" and "average case" are not relevant to amortized analysis -- any more than they are to an ordinary worst-case big-O analysis!

edited Sep 13 at 3:16

answered Feb 26 '13 at 1:18
comingstorm
17.5k ● 1 ● 22 ● 47

3 Sometimes an example is all one needs! :) – akki Nov 11 '16 at 7:09

> In an amortized analysis, the time required to perform a sequence of data-structure operations is averaged over all the operations performed... Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case.

(from Cormen et al., "Introduction to Algorithms")

That might be a bit confusing, since it says both that the time is averaged, and that its not an average-case analysis. So let me try to explain this with a financial analogy (indeed, "amortized" is a word most commonly associated with banking and accounting.)

Suppose that you are operating a lottery. (Not buying a lottery ticket, which we'll get to in a moment, but operating the lottery itself.) You print 100,000 tickets, which you will sell for 1 currency unit each. One of those tickets will entitle the purchaser to 40,000 currency units.

Now, assuming you can sell all the tickets, you stand to earn 60,000 currency units: 100,000 currency units in sales, minus the 40,000 currency unit prize. For you, the value of each ticket is 0.60 currency units, amortized over all the tickets. This is a reliable value; you can bank on it. If you get tired of selling the tickets yourself, and someone comes along and offers to sell them for 0.30 currency units each, you know exactly where you stand.

For the lottery purchaser, the situation is different. The purchaser has an expected loss of 0.60 currency units when they purchase a lottery ticket. But that's probabilistic: the purchaser might buy ten lottery tickets every day for 30 years (a bit more than 100,000 tickets) without ever winning. Or they might spontaneously buy a single ticket one day, and win 39,999 currency units.

Applied to datastructure analysis, we're talking about the first case, where we amortize the cost of some datastructure operation (say, insert) over all the operations of that kind. Average-case analysis deals with the expected value of a stochastic operation (say, search), where we cannot compute the total cost of all the operations, but we can provide a probabilistic analysis of the expected cost of a single one.

It's often stated that amortized analysis applies to the situation where a high cost operation is rare, and that's often the case. But not always. Consider, for example, the so-called "banker's queue", which is a first-in-first-out (FIFO) queue, made out of two stacks. (It's a classic functional data-structure; you can build cheap LIFO stacks out of immutable single-linked nodes, but cheap FIFOs are not so obvious). The operations are implemented as follows:

```
put(x):  Push x on the right-hand stack.
y=get(): If the left-hand stack is empty:
           Pop each element off the right-hand stack and
             push it onto the left-hand stack. This effectively
             reverses the right-hand stack onto the left-hand stack.
         Pop and return the top element of the left-hand stack.
```

Now, I claim that the amortized cost of `put` and `get` is `O(1)`, assuming that I start and end with an empty queue. The analysis is simple: I always `put` onto the right hand stack, and `get` from the left hand stack. So aside from the `If` clause, each `put` is a `push`, and each `get` is a `pop`, both of which are `O(1)`. I don't know how many times I will execute the `If` clause -- it depends on the pattern of `put`s and `get`s -- but I know that every element moves exactly once from the right-hand stack to the left-hand stack. So the total cost over the entire sequence of n `put`s and n `get`s is: n `push`es, n `pop`s, and n `move`s, where a `move` is a `pop` followed by a `push`: in other words, 2n `put`s and `get`s result in 2n `push`es and 2n `pop`s. So the amortized cost of a single `put` (or `get`) is one `push` and one `pop`.

Note that banker's queues are called that precisely because of the amortized complexity analysis (and the association of the word "amortized" with finance). Banker's queues are the answer to what used to be a common interview question, although I think it's now considered too well-known: Come up with a queue which implements the following three operation in amortized O(1) time:

1) Get and remove the oldest element of the queue,

2) Put a new element onto the queue,

3) Find the value of the current maximum element.

edited Feb 26 '13 at 1:43

answered Feb 26 '13 at 1:28

rici

124k • 12 • 100 • 163

+1 for clearly stating that probabilities have no impact on amortized complexity, +1 for the clear examples, +1 for quoting a reference work (Cormen). – Georges Dupéron Jun 7 '13 at 13:23

The principle of "amortized complexity" is that although something may be quite complex when you do it, since it's not done very often, it's considered "not complex". For example, if you create a binary tree that needs balancing from time to time - say once every `2^n` insertions - because although balancing the tree is quite complex, it only happens once in every n insertions (e.g once at insertion number 256, then again at 512th, 1024th, etc). On all other insertions, the complexity is O(1) - yes, it takes O(n) once every n insertions, but it's only `1/n`

probability - so we multiply O(n) by 1/n and get O(1). So that is said to be "Amortized complexity of O(1)" - because as you add more elements, the time consumed for rebalancing the tree is minimal.

edited Feb 26 '13 at 1:29

answered Feb 26 '13 at 0:45

Mats Petersson
101k ● 7 ● 70 ● 152

What does "large enough" have to do with it? There are extraneous details here and the key concept of multiplying by a probability is left out. – Potatoswatter Feb 26 '13 at 1:19

Unqualified amortized performance guarantees have nothing to do with probability -- they are absolute guarantees for any sequence of operations. When talking about probabilistic performance, a term of art indicating "expected" or "average-case" performance should be used. – comingstorm Feb 26 '13 at 1:25

I've rephrazed it a little bit to remove superfluous "large enough" (by which I meant that a small tree may well be rebalanced quite frequently, but a larger one isn't being rebalanced very often - but I agree it wasn't a very good way to put it, because the effort also goes up as it grows) – Mats Petersson Feb 26 '13 at 1:31

Amortized complexity is not the same as average-case complexity, so, as @comingstorm says, probabilities don't come into it. To apply amortized analysis to rebalancing binary trees, you'd have to demonstrate that (worst-case) rebalancing contributes a constant time to every insert. – rici Feb 26 '13 at 1:36

@comingstorm Thanks, I've fixed my answer. – Potatoswatter Feb 26 '13 at 2:21

Amortized means divided over repeated runs. The worst-case behavior is guaranteed not to happen with much frequency. For example if the slowest case is O(N), but the chance of that happening is just O(1/N), and otherwise the process is O(1), then the algorithm would still have amortized constant O(1) time. Just consider the work of each O(N) run to be parceled out to N other runs.

The concept depends on having enough runs to divide the total time over. If the algorithm is only run once, or it has to meet a deadline each time it runs, then the worst-case complexity is more relevant.

edited Feb 26 '13 at 2:20

answered Feb 26 '13 at 0:47

Potatoswatter
102k ● 12 ● 184 ● 333

It is somewhat similar to multiplying worst case complexity of different branches in an algorithm with the probability of executing that branch, and adding the results. So if some branch is very unlikely to be taken, it contributes less to the complexity.

answered Feb 26 '13 at 0:47

perreal
64.7k ● 6 ● 89 ● 119

Say you are trying to find the kth smallest element of an unsorted array. Sorting the array would be O(n logn). So then finding the kth smallest number is just locating the index, so O(1).

Since the array is already sorted, we never have to sort again. We will never hit the worst case scenario more than once.

If we perform n queries of trying to locate kth smallest, it will still be O(n logn) because it dominates over O(1). If we average the time of each operation it will be:

(n logn)/n or O(logn). So, Time Complexity/ Number of Operations.

This is amortized complexity.

I think this is how it goes, im just learning it too..

answered Jun 21 '14 at 7:26

user2968401
577 ● 1 ● 8 ● 24
