WIKIPEDIA

Amortized analysis

In computer science, **amortized analysis** is a method for analyzing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute. The motivation for amortized analysis is that looking at the worst-case run time *per operation*, rather than *per algorithm*, can be too pessimistic.[1]

While certain operations for a given algorithm may have a significant cost in resources, other operations may not be as costly. Amortized analysis considers both the costly and less costly operations together over the whole series of operations of the algorithm. This may include accounting for different types of input, length of the input, and other factors that affect its performance.[2]

Contents

History

Amortized analysis initially emerged from a method called aggregate analysis, which is now subsumed by amortized analysis. The technique was first formally introduced by Robert Tarjan in his 1985 paper *Amortized Computational Complexity*,[3] which addressed the need for a more useful form of analysis than the common probabilistic methods used. Amortization was initially used for very specific types of algorithms, particularly those involving binary trees and union operations. However, it is now ubiquitous and comes into play when analyzing many other algorithms as well.[2]

Method

Amortized analysis requires knowledge of which series of operations are possible. This is most commonly the case with data structures, which have state that persists between operations. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost.

There are generally three methods for performing amortized analysis: the aggregate method, the accounting method, and the potential method. All of these give correct answers; the choice of which to use depends on which is most convenient for a particular situation.[4]

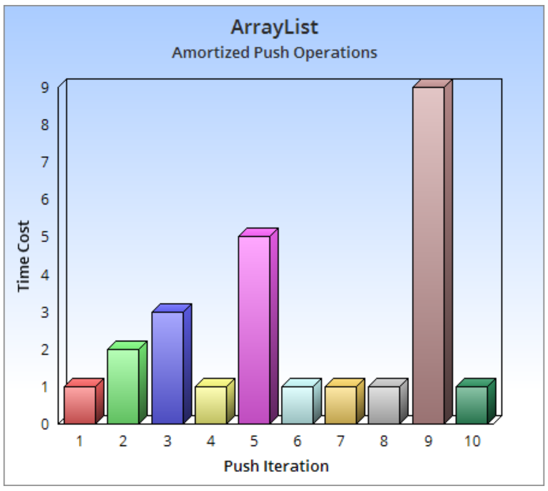
- Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the amortized cost to be $T(n) / n$.[4]
- The accounting method is a form of aggregate analysis which assigns to each operation an *amortized cost* which may differ from its actual cost. Early operations have an amortized cost higher than their actual cost, which accumulates a saved "credit" that pays for later operations having an amortized cost lower than their actual cost. Because the credit begins at zero, the actual cost of a sequence of operations equals the amortized cost minus the accumulated credit. Because the credit is required to be non-negative, the amortized cost is an upper bound on the actual cost. Usually, many short-running operations accumulate such a credit in small increments, while rare long-running operations decrease it drastically.[4]

- The potential method is a form of the accounting method where the saved credit is computed as a function (the "potential") of the state of the data structure. The amortized cost is the immediate cost plus the change in potential.[4]

Examples

Dynamic Array

Consider a dynamic array that grows in size as more elements are added to it, such as `ArrayList` in Java or `std::vector` in C++. If we started out with a dynamic array of size 4, it would take constant time to push four elements onto it. Yet pushing a fifth element onto that array would take longer as the array would have to create a new array of double the current size (8), copy the old elements onto the new array, and then add the new element. The next three push operations would similarly take constant time, and then the subsequent addition would require another slow doubling of the array size.



Amortized Analysis of the Push operation for a Dynamic Array

In general if we consider an arbitrary number of pushes $n + 1$ to an array of size n, we notice that push operations take constant time except for the last one which takes $O(n)$ time to perform the size doubling operation. Since there were $n + 1$ operations total we can take the average of this and find that pushing elements onto the dynamic array takes: $\frac{nO(1)+O(n)}{n+1} = O(1)$, constant time.[4]

Queue

Shown is a Ruby implementation of a Queue, a FIFO data structure:

```
class Queue
  def initialize
    @input = []
    @output = []
  end

  def enqueue(element)
    @input << element
  end

  def dequeue
    if @output.empty?
      while @input.any?
        @output << @input.pop
      end
    end

    @output.pop
  end
end
```

The enqueue operation just pushes an element onto the input array; this operation does not depend on the lengths of either input or output and therefore runs in constant time.

However the dequeue operation is more complicated. If the output array already has some elements in it, then dequeue runs in constant time; otherwise, dequeue takes $O(n)$ time to add all the elements onto the output array from the input array, where n is the current length of the input array. After copying n elements from input, we can perform n dequeue operations, each taking constant time, before the output array is empty again. Thus, we can perform a sequence of n dequeue operations in only $O(n)$ time, which implies that the amortized time of each dequeue operation is $O(1)$.[5]

Alternatively, we can charge the cost of copying any item from the input array to the output array to the earlier enqueue operation for that item. This charging scheme doubles the amortized time for enqueue, but reduces the amortized time for dequeue to $O(1)$.

Common use

- In common usage, an "amortized algorithm" is one that an amortized analysis has shown to perform well.
- Online algorithms commonly use amortized analysis.

References

- Allan Borodin and Ran El-Yaniv (1998). *Online Computation and Competitive Analysis* (http://www.cs.technion.ac.il/~rani/book.html). Cambridge University Press. pp. 20,141.

1. "Lecture 7: Amortized Analysis" (https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0206.pdf) (PDF). Carnegie Mellon University. Retrieved 14 March 2015.
2. Rebecca Fiebrink (2007), *Amortized Analysis Explained* (http://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf) (PDF), retrieved 2011-05-03
3. Tarjan, Robert Endre (April 1985). "Amortized Computational Complexity" (http://www.cs.duke.edu/courses/fall11/cps234/reading/Tarjan85_AmortizedComplexity.pdf) (PDF). *SIAM Journal on Algebraic and Discrete Methods*. **6** (2): 306–318. doi:10.1137/0606031 (https://doi.org/10.1137/0606031).
4. Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis" (http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec20-amortized/amortized.htm). Cornell University. Retrieved 14 March 2015.
5. Grossman, Dan. "CSE332: Data Abstractions" (http://courses.cs.washington.edu/courses/cse332/10sp/lectures/lecture21.pdf) (PDF). *cs.washington.edu*. Retrieved 14 March 2015.