Amortized Analysis

The previous topics – divide-and-conquer, dynamic programming, greedy algorithms – were algorithm design approaches. This topic offers a different outlook on the process of algorithm analysis. If your program contains a nice structure like nested for loops or a while loop that repeatedly does the same one or two operations, you are going to use the traditional "counting" method to get a worst-case upper bound. Or if your program uses recursion, you may be able to find and solve a recurrence relation to get a worst-case upper bound. Amortized analysis is for when your program does a sequence of a few operations that are low-cost, but once in a while (in a "semi-random" way) it does an expensive operation.

Example: Every night you stack your dirty dinner dishes in the sink; the time for this task is about 30 seconds. But once in a while during the week, whenever you run out of clean dishes, you have to wash all the dirty dishes, which takes 20 minutes. If you do a simplistic worst-case analysis of after-dinner time spent per week, you would take the 20 minutes and multiply it by the 7 passes through a one-week loop = 140 minutes. But most of the time, you don't spend 20 minutes, so this is a very high over-estimate. You would get a better estimate using amortization.

1. What is "amortized analysis"?

Dictionary definition of "amortize": **1 :** to pay off (as a mortgage) gradually, usually by periodic payments of principal and interest. **2 :** to gradually reduce or write off the cost or value of (as an asset). Buried in this word is its original Latin root of "mort" – death – therefore the word literally means "to kill off". So an amount is being decreased by a sequence of payments or write-offs over a period of time. Also, an amount could be increased over a period of time.

How does this relate to analysis of algorithms? Consider two types of analyses that we are familiar with:

Worst-case: max number of work units

$$\text{Average} = \sum_{\text{all cases}} (\text{work for this case}) * (\text{probability of this case})$$

Example: Sequential search on a list of n elements. Work unit = comparison of target value against list value.

Worst case (target not in list or equal to last element in list): n comparisons

Average (assume target is in the list and is equally likely (this is "probability") to be any element in the list):

Location of target in list	Number of comparisons required
1 (first element)	1
2	2
3	3
…	…
n (last position in list)	n

Average work = $1*\frac{1}{n}+2*\frac{1}{n}+\cdots+n*\frac{1}{n}=\frac{1+2+\cdots+n}{n}=\frac{1}{2}(n+1)$

end of Example

In an amortized analysis, we look at the worst case work for various operations but consider the algorithm running over a period of time, so that operations are done in sequence. An operation may be "expensive" in the worst case, but as part of a sequence of operations, it may only occur infrequently, so that its overall cost is "amortized." Unlike an average-case analysis, there is no probability involved. We must know the sequence of operations - we are not estimating the probability of various situations occurring.

There are three techniques used in amortized analysis:

- Aggregate method
- Accounting method
- Potential method

2. Aggregate method

The aggregate method is sort of a brute-force method, but fairly easy to understand. We try to find an expression T(n) for the worst-case total time taken by a sequence of n operations (perhaps more than one type of operation). Then the amortized cost per operation is T(n)/n.

Example:

Consider three operations to be done on a stack:

push(x) – push an item x onto the top of the stack
pop(x) – remove the top stack element x and return it
multipop(k) – pop the k top objects from the top of the stack
or pop all objects if stack contains less than k objects

These are three different operations, but we can estimate the cost or work of each:

cost of push = 1
cost of pop = 1
cost of multipop(k) = min(k, size of stack) so multipop(n) on a stack with n elements does n units of work

Now consider an initially empty stack on which a sequence of n push, pop, and multipop operations is done. The stack size during this sequence is at most n, so the single most expensive operation possible is multipop(n), which would be O(n) work units [this is "big oh" – upper bound]. If you simply consider that some random sequence of n of these three operations is done, the worst-case – the maximum number of work units – would be given by doing the most expensive operation, multipop(n), n times, which would be $n*O(n) = O(n^2)$. But this is way too big an upper bound because in fact it's not possible to do n multipop(n) operations in sequence.

In any such sequence, you can only pop an object once for each time it has been pushed. In n operations you can push at most n elements, so the total number T(n) of push and pop operations, including the pops done in multipop, is O(n). [If n = 4, you can do push, push, multipop(2), or push, push, push, pop, or push 4 times, etc.] Therefore the amortized worst-case cost per operation in this scenario is O(n)/n = O(1). By "averaging" over a sequence of operations, we find that the "average" cost per operation is low. In the long run, this process behaves as though each operation is O(1), so again (reversing this division), over n operations, the total cost is O(n).

3. The accounting method

In the accounting method, an amortized cost value is assigned to each operation. If the amortized cost of an operation exceeds the actual cost, the difference can be used as a "credit" by other operations whose amortized cost is less than their actual cost. But it must be the case that over any sequence of operations, the total amortized cost is greater than or equal to the total actual cost because the amortized cost is supposed to be an upper bound on the actual cost. Therefore the total "credit" available at any point in the sequence must be nonnegative.

Example:

Again consider the three stack operations. As before, the actual costs of each are:

cost of push = 1
cost of pop = 1
cost of multipop(k) = min(k, size of stack)

We assign an amortized cost to each operation as follows:

amortized cost of push = 2
amortized cost of pop = 0
amortized cost of multipop(k) = 0

It is clear that push operations build up credit because their amortized cost exceeds their actual cost, while pop operations of either type eat up credit because their amortized cost is less than their actual cost.

When an item x is pushed onto the stack, 1 unit of its amortized cost is used to "pay" for the actual cost and 1 unit of credit is associated with x. This credit will pay for a pop operation to remove x. After a sequence of 1 push and 1 pop, the total credit remaining is 0. For any sequence of push, pop, or multipop operations, the credit will never be negative because a multipop operation pops items that are on the stack and each item has 1 unit of credit associated with it to pay for its pop.

The amortized cost for each type of operation is a constant, i.e., O(1), so the total amortized cost over a sequence of n operations, which is an upper bound for the actual cost, is O(n).

Note that in the aggregate method, we analyze the entire sequence of operations, then divide to find the amortized cost per operation, which will be the same for any operation. In the accounting method, we assign an amortized cost to each operation (may vary depending on the operation), check that the credit never goes negative, and then compute the cost of the entire sequence of operations.

4. The potential method

In the accounting method, a "credit" is associated with each individual item in a data structure. The potential method is similar except that a "credit" or "potential" is associated with the entire data structure at each point in the sequence of operations.

We can represent the sequence of data structures as the n operations are performed by

$D_0, D_1, \ldots D_n$

where D_0 is the initial data structure and D_i is the data structure after performing i operations. $\Phi(D_i)$ represents the potential associated with D_i. There is an actual cost c_i for each of the i operations, and there is an amortized cost m_i for each operation. The amortized cost m_i is obtained from the equation

$$m_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

In other words, the amortized cost is the actual cost plus the change in potential. If the change is positive, then the amortized cost exceeds the actual cost and the excess increases the overall potential of the data structure. If the change is negative, then the actual cost exceeds the amortized cost and the data structure potential has been decreased to pay for the actual cost. Summing over the sequence of n operations, the total amortized cost is

$$\sum_{i=1}^{n} m_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + (\Phi(D_n) - \Phi(D_0))$$

which is the sum of the actual costs plus the potential difference across the whole sequence. Because again the amortized cost is supposed to be an upper bound on the actual cost, we must ensure that

$$\Phi(D_n) - \Phi(D_0) \geq 0 \text{ or } \Phi(D_n) \geq \Phi(D_0)$$

This condition is often achieved by defining $\Phi(D_0)$ to be 0, and $\Phi(D_i) \geq 0$ for all i ≥ 1. This says that at any point the potential of the data structure is nonnegative.

Example:

Again using our stack example, the actual costs are

cost of push = 1
cost of pop = 1
cost of multipop(k) = min(k, size of stack)

and we have to define the potential function Φ. The data structure is the stack. We start with an empty stack and define $\Phi(D_0)$ to be 0. For i ≥ 1, define $\Phi(D_i)$ to be the number of items currently on the stack. This clearly satisfies the requirement of $\Phi(D_i) \geq 0$ for all i ≥ 1.

Specifically, if the ith operation is a push operation, then one item has been added to the stack, so

$$\Phi(D_i) - \Phi(D_{i-1}) = 1$$

The amortized cost is then

$m_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$

If the ith operation is a pop operation, then one item has been removed from the stack, so

$$\Phi(D_i) - \Phi(D_{i-1}) = -1$$

and the amortized cost is

$m_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$

If the ith operation is a multipop operation, then min(k, size of stack) items have been removed from the stack, so

$$\Phi(D_i) - \Phi(D_{i-1}) = -\text{ min(k, size of stack)}$$

and the amortized cost is

$m_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \min(k, \text{size of stack}) - \min(k, \text{size of stack}) = 0$

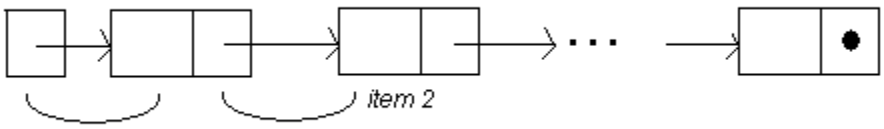
As in the accounting method, the amortized cost of each type of operation is a constant, O(1), so the total amortized cost over a sequence of n operations is O(n).

5. So what's the point?

An amortization analysis can be difficult because we have to look at an entire sequence of operations. What is the payoff? By applying an amortization analysis, we may be able to improve on the worst-case analysis of an algorithm where one particular "expensive" operation may make a standard worst-case analysis too pessimistic. Thus we may be willing to use data structures/algorithms that we would otherwise consider too resource-consuming.

6. Move-to-front

Consider the following situation. You have a linked list of s items, indexed from 1 to s, with a pointer to the head of the list (item 1). To access item i in the list requires walking through the list, following the pointers, until you get to item i. The actual cost is i (1 unit per leg of the walk). Below, you have to follow 2 pointers to access item 2 in the list.



An arbitrary sequence of n accesses is to be done, where any item i can be accessed more than once.

Any two adjacent elements can be exchanged by re-linking a couple of pointers, so call that a cost of 1. It's cheap to exchange adjacent elements. If you have knowledge of the order of access in the sequence, you could possibly combine exchanges and accesses to reduce the overall cost.

Example A:

The original list is

a c b d e

The sequence of access requests is b, c, d, c, b. Using the original list, the total cost of these accesses will be 3 + 2 + 4 + 2 + 3 = 14. Using some judicious exchanges, the total cost can be reduced:

c a b d e (swap c and a)
c b a d e (swap a and b)

then access the items in the desired sequence. The total cost is 2 (swaps) + 2 + 1 + 4 + 1 + 2 = 12.
end of Example

But in general you don't have advance knowledge of the access sequence, which is supposed to be arbitrary.

The **Move-To-Front (MTF) algorithm** assumes no advance knowledge of the access sequence. Its approach is similar to the idea of caching to minimize disk access – once you ask for something from the disk, you are likely to want to use it again, so store it [for a while] in local cache memory. MTF moves (via a series of exchanges) the most recently accessed item to the front of the list on the thought that it is somewhat likely to be accessed again. The total cost to access item i and move it to the front is i (access) + i – 1 (swaps) = 2i – 1.

Example B:

Using the same original list of a c b d e and the same access sequence of b, c, d, c, b, the MTF algorithm would do

Access b (3)
Swap a b c d e
Swap b a c d e
Access c (3)
Swap b c a d e
Swap c b a d e
Access d (4)
Swap c b d a e
Swap c d b a e
Swap d c b a e
Access c (2)
Swap c d b a e
Access b (3)
Swap c b d a e
Swap b c d a e

for a total of 25. So for this particular case, MTF was not an improvement over the clever moves made in Example A due to the foreknowledge of the access sequence.
end of Example

Let OPT be an optimum algorithm that always minimizes the total cost of swaps and accesses. This is purely hypothetical because the access sequence is arbitrary and can't be known in advance, so such an algorithm doesn't really exist. We can use amortization to prove that MTF is no more than 4 times the work of the hypothetical OPT. This is called **competitive analysis**. Instead of getting absolute information about the work done by the algorithm, we get relative information; the algorithm does no worse than a constant factor times the hypothetical optimum solution.

We will use a potential argument, so we have to define a potential function Φ. Both OPT and MTF start with the same original list. Again using D_i as the data structure after i access (or access and swap) operations have been done, we'll define $\Phi(D_i)$ to be 2*v where v is the number of *inversions* between the OPT list and the MTF list. An inversion occurs when two list items x and y are in different relative orders between the OPT list and the MTF list, for example, in the

OPT list, b precedes w but in the MTF list, w precedes b. By this definition, $\Phi(D_0) = 0$ because the two lists are identical before any operations have been performed. This definition satisfies the two conditions mentioned earlier, namely, $\Phi(D_0) = 0$, and $\Phi(D_i) \geq 0$ for all $i \geq 1$ (the number of inversions can't be negative). Therefore the amortized cost of MTF will be an upper bound on the actual cost of MTF.

Suppose item x is to be accessed, and that item x is the jth item in the MTF list and the kth item in the OPT list. The actual cost to access item x and swap it to the front in the MTF list is $2j - 1$, while the cost to access item x in the OPT list is k. What is the change in potential as a result of these actions? Any elements that appeared after item x in the MTF list before these actions remain in the same relative order with respect to each other and with respect to x as before, and so contribute nothing to a change in the potential. The $j - 1$ items before x in the MTF list will all remain in the same relative order to each other but will now be behind x. So the only change in potential comes from considering the $j - 1$ pairs {x, w} where w preceded x before the swaps were made. In the little example below, a, b, c, d, f, g are all in the same relative order as before x was moved, but the relative positions of the two components in the ordered pairs (a, x), (b, x), (c, x), (d, x) have all changed.

a b c d x f g
a b c x d f g
a b x c d f g
a x b c d f g
x a b c d f g

Each of these $j - 1$ pair re-orderings will either create a new inversion or remove an existing inversion. Let y be an element that preceded x in the MTF list before the swaps. After the swaps, x precedes y in the MTF list. This will be a new inversion if y precedes x in the OPT list. But if x precedes y in the OPT list, that inversion has now been removed. There are only $k - 1$ elements that precede x in the OPT list, so there are at most $\min\{j - 1, k - 1\}$ new inversions created. The rest of the $j - 1$ pair re-orderings, at least $(j - 1) - \min\{j - 1, k - 1)\}$, result in inversion removals.

After these actions, the number of inversions (I_{after}) is the number of inversions before the actions (I_{before}) plus the number of the new inversions created minus the number of old inversions removed, so

$$I_{after} \leq I_{before} + \min\{j - 1, k - 1\} - ((j - 1) - \min\{j - 1, k - 1)\})$$

The $\leq$ is because the first min term could be smaller and the term subtracted could be bigger, either of which makes the expression smaller.

Therefore the *change* in potential, $I_{after} - I_{before}$, is at most

$$2*[\min\{j - 1, k - 1\} - ((j - 1) - \min\{j - 1, k - 1)\})] = 4*\min\{j - 1, k - 1\} - 2(j - 1)$$

(Remember that potential is defined to be 2* the number of inversions, that's where the 2 comes from in the above expression. If you don't multiply by 2, then j does not cancel out in the following computation of MTF.)

From the formula

$$m_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

the amortized cost of these operations in MTF is at most

$$(2j - 1) \text{ [actual cost]} + 4^* \min\{j - 1, k - 1\} - 2(j - 1) \text{ [max potential change]}$$

$$= 4^* \min\{j - 1, k - 1\} + 1 \leq 4^*(k - 1) + 1 = 4^*k - 4 + 1 \leq 4^*k$$

which is 4 times the cost to access item x in the OPT list.

But we can't rule out the possibility that OPT, in its omniscient wisdom, also swaps some elements. The swap will either create or destroy a single inversion, which will produce a change in potential of at most 2*1 = 2. A swap in the OPT list costs 1 unit of work for the OPT algorithm, and of course no work for the MTF algorithm. The amortized cost to the MTF algorithm is at most 0 + 2 = 2, which is ≤ 4 times the cost (which is 1) to swap in the OPT list.

Therefore using any sequence of access requests, the amortized cost of the MTF algorithm, which is an upper bound on the actual cost, is ≤ 4 times the actual cost of the hypothetical OPT algorithm.