

AMORTIZED ANALYSIS

In an **amortized analysis**, the time required to perform a sequence of data-structure operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive. 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.

The aggregate method

In the **aggregate method** of amortized analysis, we show that for all n , a sequence of n operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or **amortized cost**, per operation is therefore $T(n) / n$. Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Stack operations

In our first example of the aggregate method, we analyze stacks that have been augmented with a new operation. The two fundamental stack operations, each of which takes $O(1)$ time:

PUSH(S, x) pushes object x onto stack S .

POP(S) pops the top of stack S and returns the popped object.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n **PUSH** and **POP** operations is therefore n , and the actual running time for n operations is therefore $\Theta(n)$.

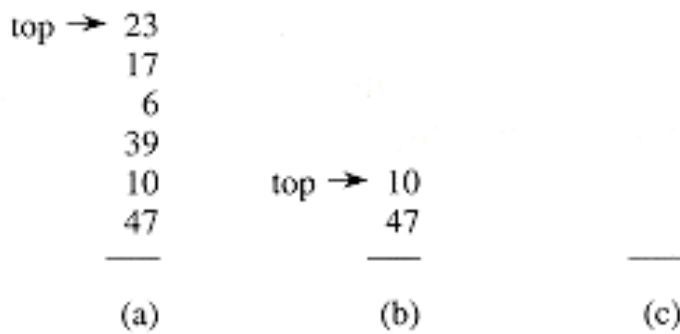
The situation becomes more interesting if we add the stack operation **MULTIPOP**(S, k), which removes the k top objects of stack S , or pops the entire stack if it contains less than k objects. In the following pseudocode, the operation **STACK-EMPTY** returns **TRUE** if there are no objects currently on the stack, and **FALSE** otherwise.

MULTIPOP(S, k)

```
1 while not STACK-EMPTY( $S$ ) and  $k \neq 0$ 
2   do POP( $S$ )
3    $k \leftarrow k - 1$ 
```

What is the running time of **MULTIPOP(S, k) on a stack of s objects?**

The actual running time is linear in the number of **POP** operations actually executed, and thus it suffices to analyze **MULTIPOP** in terms of the abstract costs of 1 each for **PUSH** and **POP**. The number of iterations of the **while** loop is the number $\min(s, k)$ of objects popped off the stack. For each iteration of the loop, one call is made to **POP** in line 2. Thus, the total cost of **MULTIPOP** is $\min(s, k)$, and the actual running time is a linear function of this cost.



The action of MULTIPOP on a stack S , shown initially in (a). The top 4 objects are popped by $\text{MULTIPOP}(S, 4)$, whose result is shown in (b). The next operation is $\text{MULTIPOP}(S, 7)$, which empties the stack—shown in (c)—since there were fewer than 7 objects remaining.

Let us analyze a sequence of n PUSH , POP , and MULTIPOP operations on an initially empty stack. The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n . The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each. Although this analysis is correct, the $O(n^2)$ result, obtained by considering the worst-case cost of each operation individually, is not tight.

Using the aggregate method of amortized analysis, we can obtain a better upper bound that considers the entire sequence of n operations. In fact, although a single MULTIPOP operation can be expensive, any sequence of n PUSH , POP , and MULTIPOP operations on an initially empty stack can cost at most $O(n)$.

Why? Each object can be popped at most once for each time it is pushed. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP , is at most the number of PUSH operations, which is at most n . For any value of n , any

sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time. The amortized cost of an operation is the average: $O(n)/n = O(1)$.

We emphasize again that although we have just shown that the average cost, and hence running time, of a stack operation is $O(1)$, no probabilistic reasoning was involved. We actually showed a *worst-case* bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded the average cost per operation, or the amortized cost.

Incrementing a binary counter

As another example of the aggregate method, consider the problem of implementing a k -bit binary counter that counts upward from 0. We use an array $A[0 \dots k-1]$ of bits, where $\text{length}[A] = k$, as the counter. A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \dots, k-1$. To add 1 (modulo 2^k) to the value in the counter, we use the following procedure.

| Counter value | [7] | [6] | [5] | [4] | [3] | [2] | [1] | [0] | Incre cost | Total cost |
|------------------|-----|-----|-----|-----|-----|-----|-----|-----|---------------|---------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 3 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 4 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 3 | 7 |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 8 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 2 | 10 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 11 |
| 8 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 4 | 15 |
| 9 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 16 |
| 10 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 2 | 18 |
| 11 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 19 |

An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is never more than twice the total number of INCREMENT operations.

INCREMENT(*A*)

1 $i \leftarrow 0$

2 **while** $i < \text{length}[A]$ and $A[i] = 1$

3 **do** $A[i] \leftarrow 0$

4 $i \leftarrow i + 1$

5 **if** $i < \text{length}[A]$

6 **then** $A[i] \leftarrow 1$

A binary counter as it is incremented 16 times, starting with the initial value 0 and ending with the value 16. At the start of each iteration of the **while** loop in lines 2-4, we wish to add a 1 into position i .

If $A[i] = 1$, then adding 1 flips the bit to 0 in position i and yields a carry of 1, to be added into position $i + 1$ on the next iteration of the loop. Otherwise, the loop ends, and then, if $i < k$, we know that $A[i] = 0$, so that adding a 1 into position i , flipping the 0 to a 1, is taken care of in line 6. The cost of each INCREMENT operation is linear in the number of bits flipped.

As with the stack example, a cursory analysis yields a bound that is correct but not tight. A single execution of INCREMENT takes time $\Theta(k)$ in the worst case, in which array A contains all 1's. Thus, a sequence of n INCREMENT operations on an initially zero counter takes time **$O(nk)$** in the worst case.

We can tighten our analysis to yield a worst-case cost of **$O(n)$** for a sequence of **n** INCREMENT'S by observing that not all bits flip each time INCREMENT is called.

$A[0]$ does flip each time INCREMENT is called. The next-highest-order bit, $A[1]$, flips only every other time: a sequence of n INCREMENT operations on an initially zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times. Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n INCREMENT'S.

In general, for $i = 0, 1, \dots, \lfloor \lg n \rfloor$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter. For $i > \lfloor \lg n \rfloor$, bit $A[i]$ never flips at all. The total number of flips in the sequence is thus

$$\sum_{i=0}^{\lg n} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} 1/2^i = 2n$$

. The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$, so the amortized cost of each operation is $O(n)/n = O(1)$.

The accounting method

In the **accounting method** of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount we charge an operation is called its **amortized cost**. When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as **credit**. Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost. Thus, one can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. This is very different from the aggregate method, in which all operations have the same amortized cost.

One must choose the amortized costs of operations carefully. If we want analysis with amortized costs to show that in the worst case the average cost per operation is small, the total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence. Moreover, as in the aggregate method, this

relationship must hold for all sequences of operations. Thus, the total credit associated with the data structure must be nonnegative at all times, since it represents the amount by which the total amortized costs incurred exceed the total actual costs incurred. If the total credit were ever allowed to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred; for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, we must take care that the total credit in the data structure never becomes negative.

Stack operations

To illustrate the accounting method of amortized analysis, let us return to the stack example. Recall that the actual costs of the operations were

PUSH 1 ,

POP 1 ,

MULTIPOP $\min(k,s)$, where k is the argument supplied to **MULTIPOP** and s is the stack size when it is called.

Let us assign the following amortized costs:

PUSH 2 ,

POP 0 ,

MULTIPOP 0 .

Note that the amortized cost of **MULTIPOP** is a constant (0), whereas the actual cost is variable. Here, all three amortized costs are $O(1)$, although in general the amortized costs of the operations under consideration may differ asymptotically.

We shall now show that we can pay for any sequence of stack operations by charging the amortized costs. Suppose we use a dollar bill to represent each unit of cost. We start with an empty stack. Recall the analogy of Section 11.1 between the stack data structure and a stack of plates in a cafeteria. When we push a plate on the stack, we use 1 dollar to pay the actual cost of the push and are left with a credit of 1 dollar (out of the 2 dollars charged), which we put on top of the plate. At any point in time, every plate on the stack has a dollar of credit on it.

The dollar stored on the plate is prepayment for the cost of popping it from the stack. When we execute a POP operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack. To pop a plate, we take the dollar of credit off the plate and use it to pay the actual cost of the operation. Thus, by charging the PUSH operation a little bit more, we needn't charge the POP operation anything.

Moreover, we needn't charge MULTIPOP operations anything either. To pop the first plate, we take the dollar of credit off the plate and use it to pay the actual cost of a POP operation. To pop a second plate, we again have a dollar of credit on the plate to pay for the POP operation, and so on. Thus, we have always charged at least enough up front to pay for MULTIPOP operations. In other words, since each plate on the stack has 1 dollar of credit on it, and the stack always has a nonnegative number of plates, we have ensured that the amount of credit is always nonnegative. Thus, for *any* sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

Incrementing a binary counter

As another illustration of the accounting method, we analyze the INCREMENT operation on a binary counter that starts at zero. As we observed earlier, the running time of this operation is proportional to the number of bits flipped, which we shall use as our cost for this example. Let us once again use a dollar bill to represent each unit of cost (the flipping of a bit in this example).

For the amortized analysis, let us charge an amortized cost of 2 dollars to set a bit to 1. When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay for the actual setting of the bit, and we place the other dollar on the bit as credit. At any point in time, every 1 in the counter has a dollar of credit on it, and thus we needn't charge anything to reset a bit to 0; we just pay for the reset with the dollar bill on the bit.

The amortized cost of INCREMENT can now be determined. The cost of resetting the bits within the **while** loop is paid for by the dollars on the bits that are reset. At most one bit is set, in line 6 of INCREMENT, and therefore the amortized cost of an INCREMENT operation is at most 2 dollars. The number of 1's in the counter is never negative, and thus the amount of credit is always nonnegative. Thus, for n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the **potential method** of amortized analysis represents the prepaid work as "potential energy," or just "potential," that can be released to pay for future operations. The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

Suppose we can define a potential function Φ (read "Phi") on states of a data structure with the following properties:

- $\Phi(h_0) = 0$, where h_0 is the initial state of the data structure.
- $\Phi(h_t) \geq 0$ for all states h_t of the data structure occurring during the course of the computation.

Intuitively, the potential function will keep track of the precharged time at any point in the computation. It measures how much saved-up time is available to pay for expensive operations. It is analogous to the bank balance in the banker's method. But interestingly, it depends only on the current state of the data structure, irrespective of the history of the computation that got it into that state.

We then define the amortized time of an operation as

$$c + \Phi(h') - \Phi(h),$$

where **c** is the actual cost of the operation and **h** and **h'** are the states of the data structure before and after the operation, respectively. Thus the amortized time is the actual time plus the change in potential. Ideally, Φ should be defined so that the amortized time of each operation is small. Thus the change in potential should be positive for low-cost operations and negative for high-cost operations.

Now consider a sequence of n operations taking actual times **c0**, **c1**, **c2**, ..., **cn-1** and producing data structures **h1**, **h2**, ..., **hn** starting from **h0**. The total amortized time is the sum of the individual amortized times:

$$(c_0 + \Phi(h_1) - \Phi(h_0)) + (c_1 + \Phi(h_2) - \Phi(h_1)) + \dots + (c_{n-1} + \Phi(h_n) - \Phi(h_{n-1}))$$

$$= c_0 + c_1 + \dots + c_{n-1} + \Phi(h_n) - \Phi(h_0)$$

$$= c_0 + c_1 + \dots + c_{n-1} + \Phi(h_n).$$

Therefore the amortized time for a sequence of operations overestimates of the actual time by $\Phi(h_n)$, which by assumption is always positive. Thus the total amortized time is always an upper bound on the actual time.

The potential method works as follows. We start with an initial data structure **D₀** on which **n** operations are performed. For each $i = 1, 2, \dots$

. . . , n , we let c_i be the actual cost of the i th operation and D_i be the data structure that results after applying the i th operation to data structure D_{i-1} . 1. A **potential function** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the **potential** associated with data structure D_i . The **amortized cost** \hat{c}_i of the i th operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \quad (18.1)$$

The amortized cost of each operation is therefore its actual cost plus the increase in potential due to the operation. By equation (18.1), the total amortized cost of the n operations is

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned} \quad (18.2)$$

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^n \hat{c}_i$ is an upper bound on the total actual cost. In practice, we do not always know how many operations might be performed. Therefore, if we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i , then we guarantee, as in the accounting method, that we pay in advance. It is often convenient to define $\Phi(D_0)$ to be 0 and then to show that $\Phi(D_i) \geq 0$ for all i .

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the i th operation is positive, then the amortized cost \hat{c}_i represents an overcharge to

the i th operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized cost represents an undercharge to the i th operation, and the actual cost of the operation is paid by the decrease in the potential.

The amortized costs defined by equations (18.1) and (18.2) depend on the choice of the potential function Φ . Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs. There are often trade-offs that can be made in choosing a potential function; the best potential function to use depends on the desired time bounds.

Stack operations

To illustrate the potential method, we return once again to the example of the stack operations PUSH, POP, and MULTIPOP. We define the potential function Φ on a stack to be the number of objects in the stack. For the empty stack D_0 with which we start, we have $\Phi(D_0) = 0$. Since the number of objects in the stack is never negative, the stack D_i that results after the i th operation has nonnegative potential, and thus

$$\begin{aligned}\Phi(D_i) &\geq 0 \\ &= \Phi(D_0).\end{aligned}$$

The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Let us now compute the amortized costs of the various stack operations. If the i th operation on a stack containing s objects is a PUSH operation, then the potential difference is

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

By equation (18.1), the amortized cost of this PUSH operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Suppose that the i th operation on the stack is MULTIPOP(S, k) and that $k' = \min(k, s)$ objects are popped off the stack. The actual cost of the operation is k' , and the potential difference is

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

Thus, the amortized cost of the MULTIPOP operation is

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 . \end{aligned}$$

Similarly, the amortized cost of an ordinary POP operation is 0.

The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$. Since we have already argued that $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

Incrementing a binary counter

We define the potential of the counter after the i th INCREMENT operation to be b_i , the number of 1's in the counter after the i th operation.

Let us compute the amortized cost of an INCREMENT operation. Suppose that the i th INCREMENT operation resets t_i bits. The actual cost of the operation is therefore at most $t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to a 1. The number of 1's in the counter after the i th operation is therefore $b_i \leq b_{i-1} - t_i + 1$, and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i.$$

The amortized cost is therefore

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2. \end{aligned}$$

If the counter starts at zero, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i , the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is $O(n)$.

The potential method gives us an easy way to analyze the counter even when it does not start at zero. There are initially b_0 1's, and after n INCREMENT operations there are b_n 1's, where $0 \leq b_0, b_n \leq k$. We can rewrite equation (18.2) as

$$\sum_{i=1}^n c_i = \sum_{i=1}^n \hat{c}_i - \Phi(D_n) + \Phi(D_0). \quad (18.3)$$

We have $\hat{c}_i \leq 2$ for all $1 \leq i \leq n$. Since $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned}\sum_{i=1}^n c_i &\leq \sum_{i=1}^n 2 - b_n + b_0 \\ &= 2n - b_n + b_0 .\end{aligned}$$

Note in particular that since $b_0 \leq k$, if we execute at least $n = \Omega(k)$ INCREMENT operations, the total actual cost is $O(n)$, no matter what initial value the counter contains.