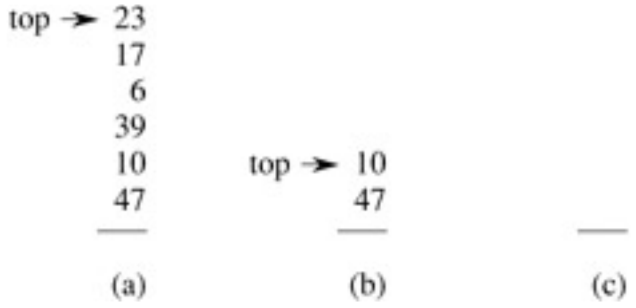
Amortized analysis (Cormen et. al., Chapter 17)

Amortized analysis analyzes algorithms that perform a sequence of similar operations. Instead of bounding the cost of the sequence of operations by bounding the cost of each operation separately, an amortized analysis can be used to provide a bound on the actual cost of the entire sequence. The goal of amortized analysis is to provide a tighter estimate the cost of a sequence of operation than that estimated by multiplying the worst-case cost of each operation. Note that amortized analysis is not an average-case analysis in that probability is not involved; an amortized analysis guarantees the worst-case of an entire sequence of operations by ensuring each operation in the worst case, which can be converted into the average performance of each operation in the sequence (i.e., **the amortized cost**). Amortized analysis is not just an analysis tool, however; it is also an way of thinking about the design of algorithms, since the design of an algorithm and the analysis of its running time are often closely intertwined.

Example 1. Stack operations.

Consider the cost of the operations of PUSH(**S**, **x**), POP(**S**) and MultiPop(), where
PUSH(**S**, **x**): pushes object **x** onto stack **S**.
POP(**S**): pops the top of stack **S** and returns the popped object
MULTIPOP(**S** , **k**): removes the k top objects of stack S, or pops the entire stack if it contains fewer than k objects.

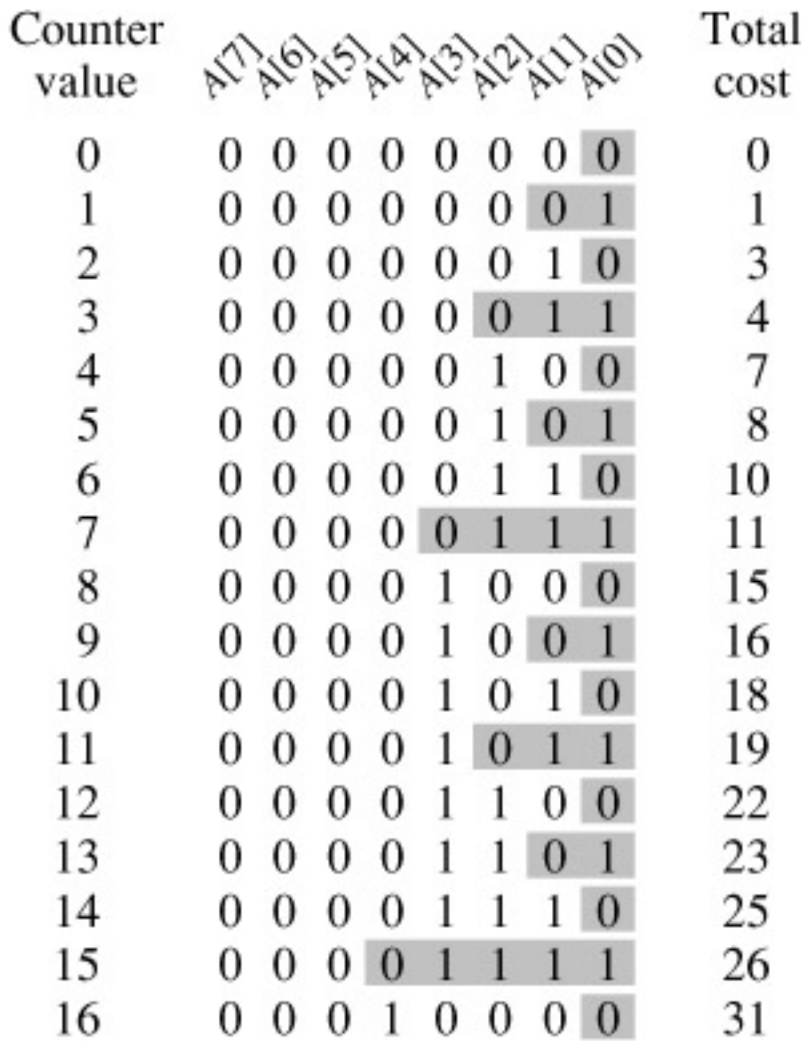


```
MULTIPOP(S , k)
1   while not STACK-EMPTY(S ) and $k \neq 0$
2       POP(S )
3       $k \leftarrow k - 1$
```

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 amortized analysis, we can obtain a better upper bound that considers the entire sequence of n operations. 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 average cost of an operation is **O**(**n**)/**n** = **O**(1). In aggregate analysis, we assign the *amortized cost* of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of **O**(1).

Example 2. Incrementing a binary counter



Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

We use an array **A** of bits, where |**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, ..., **k** - 1.

```
INCREMENT(A)
    i ← 0
    while i < |A| and A [i] = 1
        A [i ] ← 0
        i ← i + 1
    if i < |A|
        A[i ] ← 1
```

A single execution of INCREMENT takes time **Θ(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 $\lceil n/2 \rceil$ times. Similarly, bit A[2] flips only every fourth time, or $\lceil n/4 \rceil$ times in a sequence of n INCREMENT's. In general, for i = 0, 1, ..., $\lceil \lg n \rceil$, bit A[i] flips $\lceil n/2_i \rceil$ times in a sequence of n INCREMENT operations on an initially zero counter. For i > $\lceil \lg n \rceil$, bit A[i] never flips at all. The total number of flips in the sequence is thus

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$
$$= 2n ,$$

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

Example 3. Accounting (the banker's) method for stack operations

The actual costs of the operations were PUSH 1, POP 1, MULTIPOP(S, k), min(**k**, |**s**|). Let us assign each of these operations the following amortized costs (*charges*): PUSH 2, POP 0, MULTIPOP 0. Note that the charge (amortized cost) of MULTIPOP is a constant (0), whereas the actual cost is variable. When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as *credit*, which can be used later on to help pay for operations whose amortized cost is less than their actual cost. We need to prove the charges (amortized cost) should be assigned in a way that the total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence. When the PUSH is operated, the actual cost is 1, but a charge of 2 is applied, which gives a credit of 1. As such, each POP operation do not need to be charged as one can take the credit left by POP (note that, as the stack is empty initially, there should be at least one credit on the table because there should be at least one element in the stack for which the credit of the PUSH operation has not been used). The similar argument applies to MULTIPOP for which no charge needs to be made. Therefore, the assigned amortized cost gives the upper bound on a total actual cost of a sequence of n operations. The accounting method is different from aggregate analysis, in which all operations have the same amortized cost

Example 4. Accounting method for incrementing a binary counter

Let us charge an amortized cost of 2 to set a bit from 0 to 1 and 0 to reset a bit from 1 to 0. When a bit is set to 1, there is always a credit of 1 (out of the charge of 2) left, which will be used later when the bit is reset to 0. As the initial counter is 0, at any point in time, every 1 in the counter has a credit of 1 on it, and thus we needn't charge anything to reset a bit to 0; we just pay by using the credit. With this assignment, every INCREMENT operation's amortized cost is at most 2 because it has at most operation (outside the while loop) that set one bit from 0 to 1. Thus, for n INCREMENT operations, the total amortized cost is O(n), which bounds the total actual cost.

The potential (Physicist') method

We start with an initial data structure $\mathbf{D_0}$ on which **n** operations are performed. For each **i** = 1, 2, ..., **n**, we let $\mathbf{c_i}$ be the actual cost of the **i**th operation and $\mathbf{D_i}$ be the data structure that results after applying the **i**th operation to data structure $\mathbf{D_{i-1}}$. A potential function Φ maps each data structure $\mathbf{D_i}$ to a real number $\Phi(\mathbf{D_i})$, which is the potential associated with data structure $\mathbf{D_i}$. The amortized cost of the **i**th operation with respect to potential function Φ is defined by

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

The total amortized cost of the n operations is

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

If we can define a potential function Φ so that $\Phi(\mathbf{D_n}) \geq \Phi(\mathbf{D_0})$, then the total amortized cost 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 $\Phi(\mathbf{D_i}) \geq \Phi(\mathbf{D_0})$ for all **i**, then we guarantee, as in the accounting method, that we pay in advance. It is convenient for us to define $\Phi(\mathbf{D_0})$ to be 0 and then to show that $\Phi(\mathbf{D_i}) \geq 0$ for all **i**. Intuitively, if the potential difference $\mathbf{\Phi(D_i) - \Phi(D_{i-1})}$ of the **i**th operation is positive, then the amortized cost 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.

Example 5. The potential (Physicist') method for stack operations

We define the potential function Φ on a stack to be the number of objects in the stack. For the empty stack $\mathbf{D_0}$ with which we start, we have $\Phi(\mathbf{D_0}) = 0$. Since the number of objects in the stack is never negative, the stack $\mathbf{D_i}$ that results after the **i**th operation has nonnegative potential, and thus $\mathbf{\Phi(D_n) \geq \Phi(D_0)}$. The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

If the **i**th operation on a stack containing **s** objects is a PUSH operation, then the potential difference is $\mathbf{\Phi(D_i) - \Phi(D_{i-1})} = (\mathbf{s} + 1) - \mathbf{s} = 1$. Thus, the amortized cost of this PUSH operation is 2. 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 $\Phi(D_i) - \Phi(D_{i-1}) = -k'$. And thus, the amortized cost of the MULTIPOP operation = $c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$. 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**).

Example 6. Potential method for 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. 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 1. If $b_i = 0$, then the **i**th operation resets all **k** bits, and so $b_{i-1} = t_i = k$. If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$. In either case, $b_i \leq b_{i-1} - t_i + 1$, 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 = $c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$. 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**).

Example 7. Dynamic table

Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high. The solution to this trade-off problem is to use Dynamic table (arrays). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full: 1) Allocate memory for a larger table of size, typically twice the old table; 2) Copy the contents of old table to new table; 3) Free the old table. If the table has space available, we simply insert new item in available space.

What is the time complexity of n insertions using the above scheme? If we use simple analysis, the worst case cost of an insertion is O(n). Therefore, worst case cost of n inserts is n * O(n) which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don’t take Θ(n) time.

Item No.	1	2	3	4	5	6	7	8	9	10	......
Table Size	1	2	4	4	8	8	8	8	16	16	......
Cost	1	2	3	1	5	1	1	1	9	1	......

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1 \ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1 \ldots)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + \ldots)}^{\lfloor \log_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$