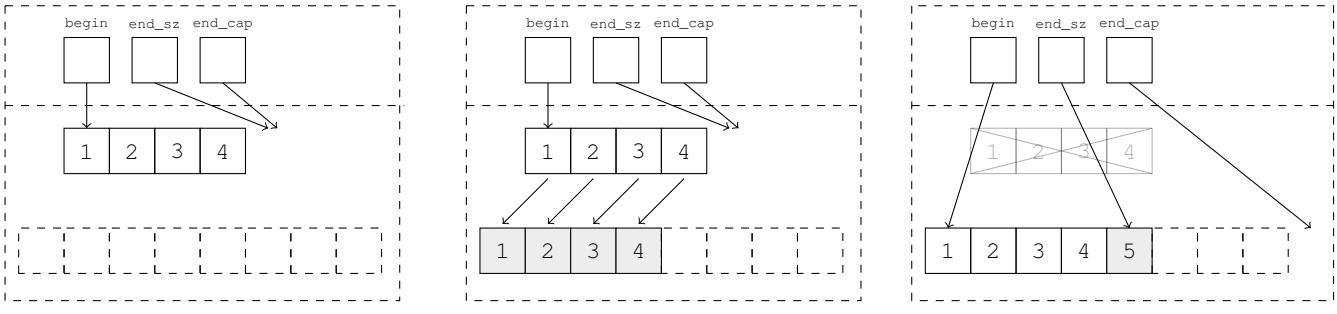
Chapter 12
Amortization and Amortized Analysis

12.1 Amortized Complexity

Suppose you are renting out an apartment with a rent of $1,500 a month, paid on the first of every month. On any given day, what is the *worst-case* cost that you would need to pay to be able to live in your apartment?

The answer to this question is $1,500, since that is the amount you would need to pay on the first of every month. However, it would be quite silly to claim that the daily cost of living in your apartment is $1,500, since that payment is not done in isolation. By paying that $1,500 up front, not only do you earn the privilege to stay at your apartment for that day, but also for the remaining days in the month! It would be more meaningful to distribute the $1,500 payment over all the days you are paying for to describe the daily cost of living in your apartment, which comes out to $50 per day (assuming 30 days in the month).

Why is this analogy relevant? Consider `std::vector::push_back()`, which we claimed had a worst-case time complexity of $\Theta(n)$. This is because a single push may require you to reallocate the underlying array with double the capacity and copy all existing items over if it goes beyond the vector's existing capacity.



However, even though a single push could take $\Theta(n)$ time in the worst case, this worst-case push has the side effect of ensuring that the next n pushes can be done in constant time (due to the reallocation). Because of this, the worst-case time complexity of $\Theta(n)$ does not give us the most accurate reflection of the operation's performance. Similar to paying your rent upfront to avoid paying on the remaining days of the month, `std::vector::push_back()` pays a $\Theta(n)$ cost up front during reallocation to allow subsequent pushes to be done in $\Theta(1)$ time.

This is an issue that comes up in certain data structures, where an operation may be cheap in some scenarios, but expensive in others. When we measure the cost of an operation, we often use its worst-case time complexity, since it gives us a reasonable upper bound for the resources that are needed for that operation. However, `std::vector::push_back()` is a good example of a situation where the worst-case time complexity can be too pessimistic. For instance, what is the worst-case time complexity of pushing n elements to the back of a vector rather than just one? If we strictly consider the worst-case scenario, we could assume that the worst-case time complexity of pushing n elements is $\Theta(n^2)$ if all n pushes take $\Theta(n)$ time. However, this produces an upper bound that is way too high, since it is impossible for all n pushes to take $\Theta(n)$ time. How can we ensure that our complexity bound is as tight as possible for a sequence of operations, if a worst-case call to a single operation happens infrequently?

To answer this question, we will use a technique known as **amortized analysis**, which can be used to determine a tighter complexity bound for a *sequence* of operations. This bound can be used to derive an operation's **amortized complexity**. The amortized complexity of an operation gives us a convenient way to estimate an operation's overall performance in the long run without having to worry about the details behind individual calls (i.e., which calls exhibited worst-case behavior and which ones did not). To calculate an operation's amortized complexity, we first consider a sequence of operations all at once, and then we distribute the maximum work required to complete this sequence of operations over *all* the operations that are performed. This result gives us a better idea of how much an operation truly costs within a sequence of operations, even if individual calls may be expensive on their own. Using the apartment example, for instance, we could say that the "amortized cost" of living in the apartment is \$50 per day (\$1,500 distributed over 30 days), since the total cost during a 30-day period cannot exceed \$1,500 — treating the daily cost as \$50 (rather than \$1,500) gives us a more reasonable estimate for how much it costs to live in the apartment on each day.

12.2 Aggregate Analysis

There are several ways we can compute the amortized complexity of an operation. One method to determine the amortized complexity of an operation is **aggregate analysis**. In an aggregate analysis, we first prove that a sequence of n operations requires at most $T(n)$ work in the worst case. Once we prove this, then the *amortized cost* of each operation in the sequence is $T(n)/n$.

As an example, let's use aggregate analysis to determine the amortized complexity of `std::vector::push_back()`. Consider a `std::vector<>`, which keeps track of its data using an underlying dynamic array. Whenever the underlying array is completely filled, adding an additional element forces the vector to allocate a larger array that is double the capacity of the original array. The original elements are then copied over from the old array to the new array.

How much work is required for a single call to `.push_back()`? This depends on whether reallocation is involved. If there is no reallocation involved, we can just append the element to the back of the vector in constant time. However, if reallocation is involved, we will have to copy the existing elements in the vector to a new location before appending a new element at the end, which takes linear time. The following table shows the work involved with different calls to `.push_back()` on a vector:

Item Number	1	**2**	**3**	4	**5**	6	7	8	**9**	10	11	12	13	14	15	16	**17**
Cost	1	**1+1**	**1+2**	1	**1+4**	1	1	1	**1+8**	1	1	1	1	1	1	1	**1+16**

Since the table doubles upon reallocation, we would need to copy 1, 2, 4, 8, and 16 elements on the 2nd, 3rd, 5th, 9th, and 17th pushes, respectively. This is why pushing the 2nd element requires $1 + 1 = 2$ units of work: 1 for appending the new element, and 1 for copying the old value over. We can express the total work required to push n elements as

$$T(n) = \underbrace{(1+1+1+1+\ldots)}_{\text{cost of appending new elements}} + \underbrace{(1+2+4+8+\ldots+n')}_{\text{cost of copying during reallocation}}$$

where $1+1+\ldots$ is done n times, and n' represents the largest power of two that is smaller than n. Since 1 is added n times, the total cost of appending new elements is n. To determine the cost of copying during reallocation, we can use the following identity

$$\sum_{k=0}^{n-1}(ar^k) = a\left(\frac{1-r^n}{1-r}\right)$$

where a is the first term, r is the common ratio, and n is the number of terms. We can express $1+2+4+8+\ldots+n'$ as

$$\sum_{k=0}^{(\log_2 n+1)-1} 2^k = \frac{1-2^{\log_2 n+1}}{1-2} = \frac{1-2^{\log_2 n}\times 2}{1-2} = \frac{1-2n}{1-2} = \frac{2n-1}{2-1} = 2n-1$$

Putting it all together, the total work required to push n elements in the worst case is bounded by

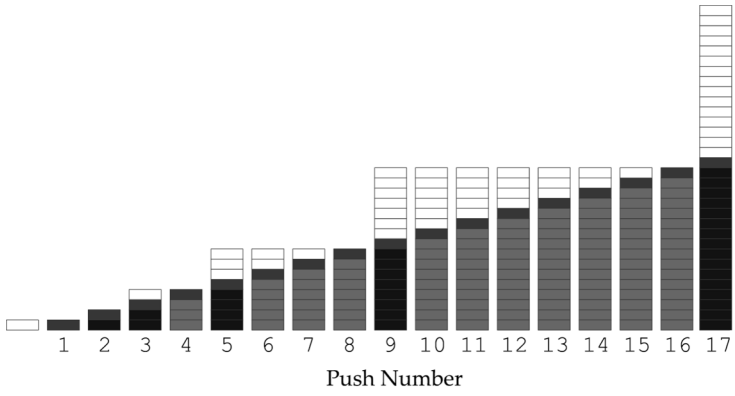
$$T(n) = n+2n-1 = \Theta(n)$$

Using aggregate analysis, we can calculate the amortized complexity as

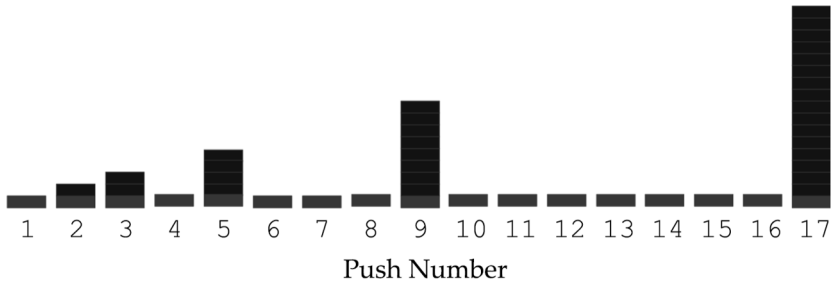
$$\text{Amortized Complexity} = \frac{T(n)}{n} = \frac{\Theta(n)}{n} = \Theta(1)$$

This implies two things. First, if capacity is doubled upon reallocation, the time complexity of calling `.push_back()` n times is $n \times \Theta(1) = \Theta(n)$, regardless of how many pushes are made. Second, since the time complexity of n calls to `.push_back()` is bounded by $\Theta(n)$, each call to `.push_back()` within a *sequence* of n pushes can be treated as if it were a $\Theta(1)$ time operation.

We can also see this visually. Consider the following figure, which shows the amount of work required for 17 calls to `.push_back()` on a vector. The black rectangles (pushes 2, 3, 5, 9, and 17) indicate elements that were copied due to reallocation. The dark gray rectangles at the top of each column represent the new elements added with each call. The white rectangles indicate unused capacity, and the light gray rectangles represent positions in the vector that are already occupied.



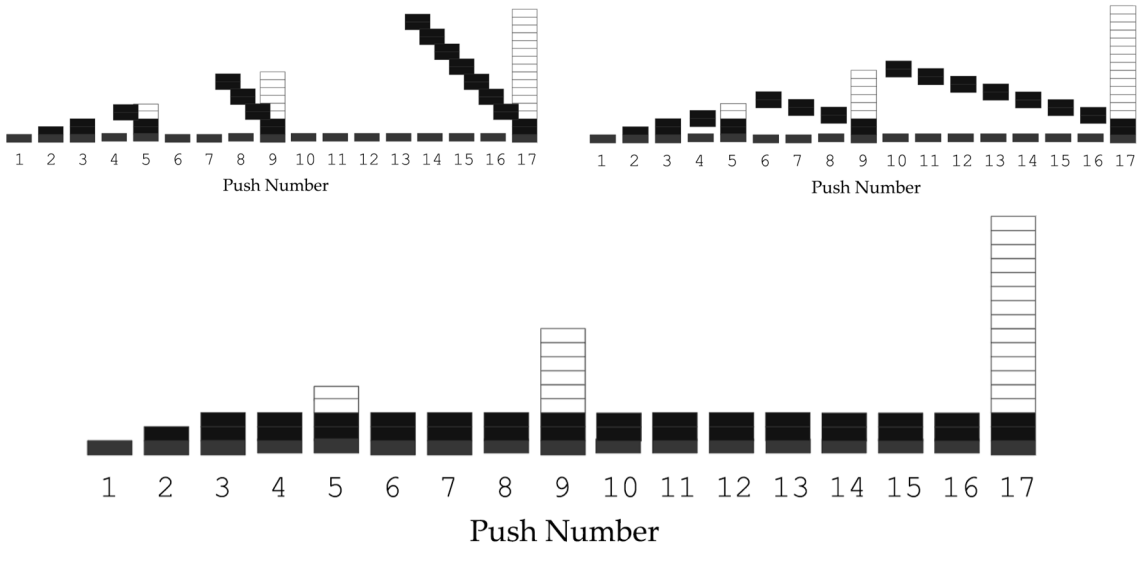
Each individual call to `.push_back()` involves a constant amount of work for the new element that was added in, but calls 2, 3, 5, 9, and 17 have additional work associated with reallocation. During reallocation, a new array with double the capacity is allocated, and the data in the old array is copied over one-by-one. If we modify the figure to only include the elements that contribute to the work of each `.push_back()` call, we would get the following:



The number of rectangles in the above figure represents the total work needed to push 17 elements into a vector, assuming no other calls are made. The height of each column indicates the amount of work required for that specific `.push_back()` call. The 9$^{\text{th}}$ push, for example, is much costlier than the 8$^{\text{th}}$ push, as adding the 9$^{\text{th}}$ element would require the previous 8 to be copied due to reallocation.

We can see that the worst case of `.push_back()` happens when elements 2, 3, 5, 9, and 17 are added, since reallocation is involved. The amount of work required for these pushes is on the order of the number of elements that need to be copied over, or $\Theta(n)$. On the other pushes, the amount of work required is constant.

However, there is a pattern when it comes to pushes. Even though copying elements during a reallocation takes extra work, this additional copying reduces the work required for subsequent pushes. For instance, the reallocation done on the 9$^{\text{th}}$ push ensures that pushes 10-16 can be done in constant time. If we take the expensive copying work and distribute it evenly across all the pushes, we get a different picture:



Note that the *total* work done is still the same! However, by distributing the work across all pushes, each push is now associated with a constant amount of work. This obviously is not how `.push_back()` works behind the scenes, since some calls may require more work than others. However, from a complexity analysis standpoint, we can simply treat `.push_back()` as an operation that always takes constant time, and we would still obtain the same result for a sequence of pushes. This is what we mean when we say that `.push_back()` has an amortized complexity of $\Theta(1)$: given a sequence of calls to `.push_back()`, the average cost per operation in the sequence is bounded above by $\Theta(1)$.

It should be mentioned that `.push_back()` would *not* have an amortized time complexity of $\Theta(1)$ if capacity were increased by a *fixed constant* during reallocation instead of doubled. To prove this, suppose we increased the capacity of a vector by an arbitrary fixed constant k during reallocation. In this scenario, we would have to copy over k elements during the first reallocation, $2k$ elements during the second reallocation, $3k$ elements during the third reallocation, and so on. The total work required to push n elements can therefore be expressed as

$$T(n) = \underbrace{(1+1+1+1+\ldots)}_{\text{cost of appending new elements}} + \underbrace{(k+2k+3k+4k+\ldots+\left\lfloor \frac{n}{k} \right\rfloor k)}_{\text{cost of copying during reallocation}}$$

Pulling out the k, we get

$$T(n) = \underbrace{(1+1+1+1+\ldots)}_{\text{cost of appending new elements}} + k\underbrace{\left(1+2+3+4+\ldots+\left\lfloor \frac{n}{k} \right\rfloor\right)}_{\text{cost of copying during reallocation}}$$

The sum of an arithmetic sequence can be solved using the following equation, where n is the number of terms, a_1 is the value of the first term, and a_n is the value of the last term.

$$S_n = \frac{n(a_1+a_n)}{2}$$

In this case, the sequence $1+2+3+\ldots+\left\lfloor \frac{n}{k} \right\rfloor$ can be rewritten as

$$1+2+3+\ldots+\left\lfloor \frac{n}{k} \right\rfloor = \frac{\left\lfloor \frac{n}{k} \right\rfloor\left(1+\left\lfloor \frac{n}{k} \right\rfloor\right)}{2} = \Theta(n^2)$$

Putting this back into the original expression for total work, we get

$$T(n) = (1+1+1+1+\ldots)+\Theta(n^2) = n+\Theta(n^2) = \Theta(n^2)$$

Using aggregate analysis, the amortized complexity of `.push_back()` on a vector that increases capacity by a fixed constant during reallocation is thus

$$\text{Amortized Complexity} = \frac{T(n)}{n} = \frac{\Theta(n^2)}{n} = \Theta(n)$$

12.3 The Accounting Method (*)

The **accounting method** is another technique that can be used to determine the amortized complexity of an operation. In the accounting method, we assign a monetary cost to each type of operation within a sequence of operations; this cost represents the operation's *amortized cost*. Unlike aggregate analysis, each type of operation can have its own amortized cost. If the amortized cost of an operation exceeds its actual cost, the difference is stored as *credit* that can be used for later operations. The goal of the accounting method is to assign an amortized cost for each operation so that the entire sequence of operations can be completed without ever running out of credit.

To illustrate this process, let's go back to the vector `.push_back()` example. Let's denote the *actual* cost of push i as c_i and the amortized cost of push as $\hat{c}$. For instance, to push 10 elements into a vector, the actual cost incurred to complete these pushes would be $c_1+c_2+\ldots+c_{10}$. However, some of these pushes may be costlier than others. Our goal is to find a tight bound on $\hat{c}$ such that

$$\sum_{i=1}^{n} c_i \leq n\hat{c}$$

holds for any sequence of n operations. This inequality just means that the total credit (i.e., amortized cost) you accumulate from a sequence of operations ($n\hat{c}$) must be greater than or equal to the actual cost incurred to perform these operations (the summation term). If this inequality holds for a sequence of n operations, the value of $\hat{c}$ would be the amortized cost of the operation.

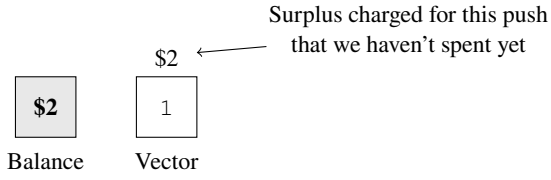
Remark: The above inequality was used under the assumption that all n operations are the same and exhibit the same amortized cost. However, the accounting method can also be used to assign *different* amortized costs to different types of operations within a sequence of operations. If this were the case, then the following expression should be used instead, where $\hat{c}_i$ is the amortized cost of the i^{th} operation within a sequence of n operations.

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

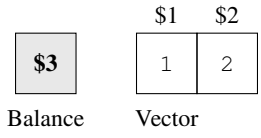
Here, we would want to find an amortized cost $\hat{c}$ for each type of operation within the sequence such that the summation of these amortized costs does not exceed the actual cost required to complete all n operations.

The idea behind finding $\hat{c}$ is to overcharge for the cheaper operations so that enough credit can be built up for more expensive operations later. Consider the vector `.push_back()` example, where we consider the cost of a single push (with no reallocation) as \$1. In this example, the 8^{th} push does not require any copying, so the cost to push back the 8^{th} element is just \$1. However, the 9^{th} push triggers a reallocation, which costs \$9 in total (\$1 for inserting the new element, \$8 for copying over the existing 8 elements). Note that we cannot charge \$1 for each push, since we would quickly run out of credit for the expensive calls.

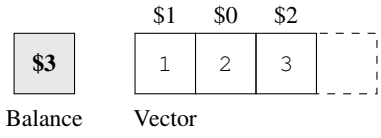
However, if we instead charge \$3 for each push, our credit balance would never fall below zero. We start off with \$0 and an empty vector. During the first push request, we charge \$3 (the *amortized* cost) and use up \$1 (the *actual* cost) to conduct the push. The amount of excess credit we obtain from this push is the difference between amortized and actual cost, or \$3 - \$1 = \$2.



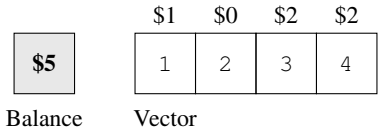
The second push causes a reallocation, doubling the vector's capacity to 2. The cost for the second reallocation is \$2: \$1 for appending the new element, and \$1 for copying the existing element over. We charged \$3 for this operation, but spent \$2, so the net credit from this second push is \$1. Our credit balance is now \$2 + \$1 = \$3. Note that we will always pay for copying costs using the surplus earned from previous pushes.



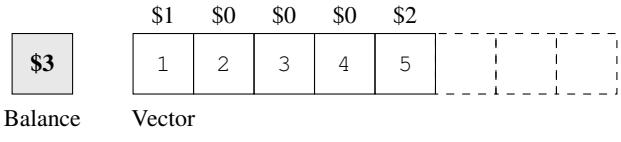
The third push causes another reallocation, doubling the vector's capacity to 4. The cost of this push is \$3, since 1 new element was added and 2 old elements were copied. Since we charged \$3 for this push, we are able to handle the cost of this reallocation.



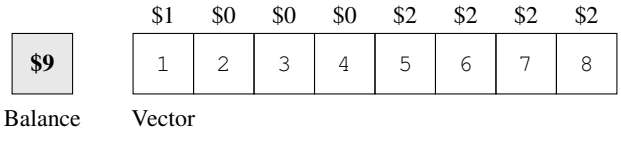
The fourth push costs \$1, since no reallocation is done. We get \$3 but only spend \$1, so we add \$2 to our surplus.



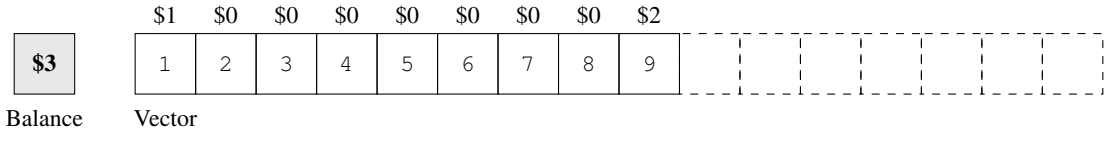
The fifth push causes a reallocation, doubling the vector's capacity to 8. The cost of this reallocation is \$5: \$1 for adding the new element, and \$4 for copying the existing element over. The \$4 cost of the copying will be paid for by the existing credit balance, while the \$1 cost of adding the new element will be paid from the \$3 we earned on the push.



The sixth, seventh, and eighth pushes can be done without copying, since we have excess capacity:



The ninth push requires a reallocation, so the total cost is \$9: \$1 for adding the new element, and \$8 for copying the old elements over. We have enough credit in our balance to pay for the reallocation (using the surplus gained from the fifth to eighth pushes).



If we continue this process, our balance will never drop below zero (after each reallocation, the balance always resets to \$3). The \$3 charged for each element is spent as follows: \$1 is spent on appending the element when it is newly inserted, \$1 is spent on moving this element during its first reallocation, and \$1 is spent on moving a previous element during reallocation. Thus, if we charge \$3 for each push, we will be able to sustain the cost of reallocation in perpetuity. Since we now know that every push has a constant cost of \$3, we can define the amortized complexity of `.push_back()` as $\Theta(1)$, since 3 is a constant.

The accounting method is best suited for proving a $\Theta(1)$ bound on an operation's amortized time complexity. When analyzing an algorithm using the accounting method, we typically set elementary operations (such as adding an element to a container) to a cost of \$1. Then, we try to find a price we can charge for each operation that will allow us to keep our balance non-negative. If we can assign an operation a constant, fixed price (such as \$3) and prove that our balance will always stay non-negative for any n calls to that operation, then that operation must have an amortized complexity of $\Theta(1)$.

12.4 The Potential Method (*)

The **potential method** is another method that can be used to measure the amortized complexity of an operation. Similar to the accounting method, the potential method keeps track of prepaid costs that can be saved up and used for more expensive operations down the line. However, instead of measuring a credit balance, this method measures the "potential energy" of a data structure using something known as a **potential function**. The potential energy of a data structure represents work that has already been "paid for" in the amortized analysis but not yet used; surplus energy gained from cheap operations can be used to complete more expensive operations later on. By measuring potential energy using a function rather than a numeric credit balance, the potential energy of a data structure can be easily derived at any point in time, after any sequence of operations on the data structure.

Suppose we wanted to complete a sequence of n operations on a data structure, which starts off at an initial state we will denote as D_0. Using this notation, we will denote D_1 as the state of the data structure after the first operation, D_2 as the state of the data structure after the second operation, and (in general) D_n as the state of the data structure after the n^{th} operation.

The goal of the potential method is to come up with a **potential function** Φ that maps each state D_i to some real number $\Phi(D_i)$ that represents the potential energy of the data structure at that state. We want to define Φ such that

- $\Phi(D_0) = 0$, where D_0 is the initial state of the data structure (i.e., potential energy starts at 0).
- $\Phi(D_i) \geq 0$ for all states D_i, where $1 \leq i \leq n$ (i.e., the potential energy at any state cannot be negative).

Once we define a potential function, we can measure the amortized cost of the i^{th} operation as

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

Any potential function that satisfies the above two conditions can be used, but it is best to define Φ so that the amortized cost $\hat{c}_i$ of each operation is as small as possible. To do so, we ideally want the difference in potential energy $\Phi(D_i) - \Phi(D_{i-1})$ to be positive if operation i has a relatively low cost and negative if operation i has a relatively high cost (this is because we want potential energy to build up during cheap calls so that we can use it for expensive calls later on). This strategy allows us to accumulate just enough potential energy with the low-cost operations to pay for future high-cost operations without the total potential energy of the data structure ever falling below 0.

Why does the potential method work? Suppose we have a sequence of n operations on a data structure, with *actual* costs c_1, c_2, …, c_n. Some of these costs are expensive, some of them are cheap. However, we can safely express the time complexities of these operations using their amortized costs instead of their actual costs because the sum of all amortized costs ($\hat{c}_1 + \hat{c}_2 + \ldots + \hat{c}_n$) is guaranteed to be no better than the actual cost required to execute all n operations ($c_1 + c_2 + \ldots + c_n$), as proven below.

$$\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i &= \hat{c}_1 + \hat{c}_2 + \ldots + \hat{c}_n \\
&= c_1 + \Phi(D_1) - \Phi(D_0) + c_2 + \Phi(D_2) - \Phi(D_1) + \ldots + c_n + \Phi(D_n) - \Phi(D_{n-1}) \\
&= c_1 + c_2 + \ldots + c_n + \Phi(D_n) = \sum_{i=1}^{n} c_i + \Phi(D_n) \;\geq\; \sum_{i=1}^{n} c_i \text{ (since } \Phi(D_n) \text{ must always be non-negative)}
\end{aligned}$$

Thus, the total amortized cost obtained by the potential method is an upper bound on the actual cost required to complete a sequence of operations, regardless of how expensive or cheap each individual operation is. Because of this, it is valid to express the cost of each operation in the sequence using its amortized cost instead of its actual cost.

Let's consider the `.push_back()` example from before, where capacity is doubled during reallocation. To perform amortized analysis using the potential method, we can use the following potential function

$$\Phi(D_i) = 2n_i - k_i$$

where D_i represents the state of the vector after the i^{th} call to `.push_back()`, n_i represents the size of the vector after the i^{th} call, and k_i represents the capacity of the vector's underlying data array after the i^{th} call. This is a valid potential function, as Φ will always be non-negative because the underlying array is guaranteed to be at least half-full after reallocation (i.e., $2n_i \geq k_i$).

Now, let's measure the amortized cost of each `.push_back()` call. Note that there are two scenarios that can happen: the `.push_back()` call either triggers a reallocation, or it does not. If the i^{th} push does *not* trigger a reallocation, the actual cost of the push c_i is 1. Thus, the amortized cost of a push with no reallocation is:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2n_i - k_i) - (2(n_i - 1) - k_i) \\ &= 1 + 2n_i - k_i - 2n_i + 2 + k_i \\ &= 1 + 2 = 3\end{aligned}$$

If there is a reallocation, the actual cost of the i^{th} push is n_i, since the first $(n_i - 1)$ elements have to be copied over, in addition to the insertion cost of 1 for the new element that was added. Furthermore, if we know that the i^{th} push triggered a reallocation, the capacity of the vector at D_{i-1} must have been $(n_i - 1)$, and the capacity of the vector at D_i must be $2(n_i - 1)$ after doubling during reallocation. Plugging this into the formula for amortized cost, we get

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= n_i + (2n_i - 2(n_i - 1)) - (2(n_i - 1) - (n_i - 1)) \\ &= n_i + 2n_i - 2n_i + 2 - 2n_i + 2 + n_i - 1 \\ &= 2 + 2 - 1 = 3\end{aligned}$$

In both cases, the amortized cost of `.push_back()` has a constant value of 3. Thus, the amortized complexity of `.push_back()` using the potential method also ends up being $\Theta(1)$, since 3 is a constant.

Remark: For the potential function to work, $\Phi(D_0)$ does not actually need to be equal to 0. This is just a convention that makes it easier to come up with a potential function.

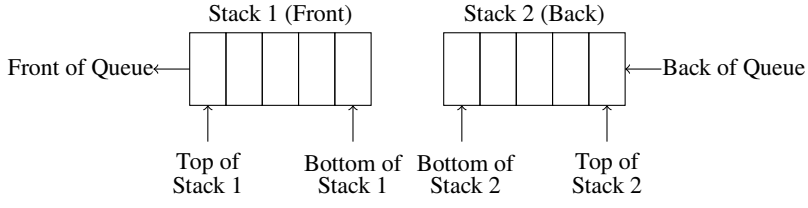
In fact, any function Φ for which $\Phi(D_i) \geq \Phi(D_0)$ for all i can be used as a valid potential function. This is because we can construct another potential function $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$ that satisfies $\Phi'(D_0) = 0$ and $\Phi'(D_i) \geq 0$, and the amortized costs of Φ and Φ' would be exactly the same. This is shown below:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi'(D_i) - \Phi'(D_{i-1}) \\ &= c_i + (\Phi(D_i) - \Phi(D_0)) - (\Phi(D_{i-1}) - \Phi(D_0)) \\ &= c_i + \Phi(D_i) - \Phi(D_{i-1})\end{aligned}$$

In other words, it is possible to convert any potential function Φ that satisfies $\Phi(D_i) \geq \Phi(D_0)$ into a potential function Φ' that satisfies $\Phi'(D_0) = 0$ and $\Phi'(D_i) \geq 0$ without changing the result of the amortized analysis. Because of this, Φ can still be a valid potential function as long as $\Phi(D_i) \geq \Phi(D_0)$ is satisfied for all i.

12.5 Amortized Analysis: Implementing a Queue with Two Stacks

In this section, we will conduct an amortized analysis on the implementation of a queue with two stacks (as discussed in chapter 9). In this algorithm, we treat one stack as the "front" of the queue and the other stack as the "back" of the queue.



Consider the following two functions in this implementation: `.push()`, which pushes an element to the back stack, and `.pop()`, which removes an element from the front stack. If the front stack is empty when `.pop()` is called, all of the elements in the back stack are transferred to the front stack.

```
1  void push(T x) {
2    stackBack.push(x);
3  } // push()
4
5  void pop() {
6    if (stackFront.empty()) {
7      while (!stackBack.empty()) {
8        stackFront.push(stackBack.top());
9        stackBack.pop();
10     } // while
11   } // if
12   stackFront.pop();
13 } // pop()
```

The worst-case time complexity of removing an element from this queue is $\Theta(n)$. This happens when we try to remove an element when the front stack is empty, which forces us to copy the entirety of the back stack into the front stack. However, this situation occurs so rarely that it would be pessimistic to expect a sequence of n pops to take $\Theta(n^2)$ time, since it is impossible for every pop to take $\Theta(n)$ time. Is there a way we can assign the `.pop()` operation a tighter bound on runtime? It turns out that we can, using amortized analysis.

※ 12.5.1 *Aggregate Analysis*

Let's first consider the aggregate analysis approach. If we look at the lifetime of any element that enters and leaves the queue, the total cost associated with that element is at most 4 (1 to push into the back stack, 1 to pop out of the back stack, 1 to push into the front stack, 1 to pop out of the front stack). Thus, for any sequence of n operations on this data structure, the total cost incurred cannot be greater than $4n$. As a result, we can divide this by n to obtain an amortized complexity of $\Theta(1)$.

$$\text{Amortized Complexity} = \frac{T(n)}{n} = \frac{4n}{n} = \Theta(1)$$

※ 12.5.2 *The Accounting Method* (✻)

We can also use the accounting method to prove that the amortized complexities of `.push()` and `.pop()` are both $\Theta(1)$. If we consider the actual cost of pushing or popping an element from the underlying stacks as \$1, we can charge an amortized cost of \$3 per push and \$1 per pop to ensure that our credit balance is always non-negative.

How does this work? When we first push an element into the back stack, we charge an amortized cost of \$3 for an operation that actually costs \$1. This allows us to accumulate \$2 worth of credit for each element that we initially push in. However, if we have to move an element from the back stack to the front stack, we will have to incur an additional \$2 cost (\$1 to pop out of the back stack, and \$1 to push into the front stack). This is why we overcharged \$2 for every push — by doing so, we are guaranteed to have enough credit to move each element from one stack to the other, since this process was paid for in advance when the element was first pushed into the back stack. Lastly, the \$1 cost to pop an element from the front stack is handled by the \$1 amortized cost of `.pop()`. Since `.push()` has an amortized cost of \$3 and `.pop()` has an amortized cost of \$1, we can conclude via the accounting method that both operations have an amortized complexity of $\Theta(1)$.

※ 12.5.3 *The Potential Method* (✻)

With the potential method, we can use the potential function $\Phi(D_i) = 2n_i$, where n_i is the number of elements in the back stack. This is a valid potential function because $\Phi(D_0) = 0$ and $\Phi(D_i)$ is always non-negative for any D_i. The amortized cost for `.push()` is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 2n_i - (2(n_i - 1)) \\ &= 1 + 2n_i - 2n_i + 2 \\ &= 1 + 2 = 3\end{aligned}$$

There are two cases that can happen for `.pop()`: the front stack can either be filled or empty. If the front stack is filled, the cost of the pop is 1, and the number of elements in the back stack remains unchanged. Here, the amortized cost of `.pop()` is

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

On the other hand, if the front stack is empty, all the elements in the back stack must be copied to the front stack. Since there are n_i elements in the back stack, the total cost of pop is $2n_i + 1$ (the $2n_i$ comes from the cost to pop n_i elements from the back stack and then push them into the front stack, and the additional 1 comes from popping off the top element after the copying is done). After all the elements are moved from the back stack to the front stack, the potential energy is 0, since there are no more elements remaining in the back stack. Thus, the amortized cost of `.pop()` in this scenario is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (2n_i + 1) + 0 - 2n_i \\ &= 1\end{aligned}$$

In both cases, the amortized cost of `.push()` is 3, and the amortized cost of `.pop()` is 1. Since both are constants, the amortized complexities of `.push()` and `.pop()` are both $\Theta(1)$.

12.6 Amortized Analysis: Binary Counter (⁎)

In this section, we will look at a *binary k-bit counter*, implemented using a k-element binary array `A` that stores a base-2 number represented using 0s and 1s. The binary counter is initialized to 0 and supports an `.increment()` function, which adds 1 to the binary number in the array. For instance, the following are the contents of the array after eight calls to `.increment()`:[1]

A[0]	A[1]	...	A[k-4]	A[k-3]	A[k-2]	A[k-1]	cost
0	0	...	0	0	0	0	⤵ 1
0	0	...	0	0	0	1	⤵ 2
0	0	...	0	0	1	0	⤵ 1
0	0	...	0	0	1	1	⤵ 3
0	0	...	0	1	0	0	⤵ 1
0	0	...	0	1	0	1	⤵ 2
0	0	...	0	1	1	0	⤵ 1
0	0	...	0	1	1	1	⤵ 4
0	0	...	1	0	0	0	

Every time we call `.increment()` on this binary counter, one unit of work is required to flip a single bit from 0 to 1 (or vice versa). Thus, the worst-case time complexity of a single `.increment()` call is $\Theta(k)$, which happens if all k bits need to be flipped (this occurs when we increment from $2^{k-1}-1$ to 2^{k-1}). However, what if we called `.increment()` n times, a worst-case time complexity class of $\Theta(nk)$ is *not* a tight bound on runtime. For the worst case to be $\Theta(nk)$, all n calls to `.increment()` must take $\Theta(k)$ time, which is impossible. To establish a tighter bound on runtime, we can use amortized analysis.

※ 12.6.1 *Aggregate Analysis* (⁎)

First, let's determine the amortized complexity of `.increment()` using aggregate analysis. If we call `.increment()` n times, what is the maximum amount of work required to perform these calls? To solve this, notice that not every bit is flipped with each `.increment()` call. The last bit in the array (`A[k-1]`) is flipped during every increment, but the second to last bit (`A[k-2]`) is only flipped every other time. Similarly, the third to last bit (`A[k-3]`) is flipped once every 4 increments, the fourth to last bit (`A[k-4]`) is flipped once every 8 increments, and so on. In general, the n^{th} to last bit in the array is only flipped once every 2^{n-1} increments.

As a result, if we call `.increment()` n times, the last bit is flipped n times, the second to last bit is flipped $\lfloor \frac{n}{2} \rfloor$ times, the third to last bit is flipped $\lfloor \frac{n}{4} \rfloor$ times, the fourth to last bit is flipped $\lfloor \frac{n}{8} \rfloor$ times, and so on. Thus, the total number of flips involved with n increment calls is

$$T(n) = n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \left\lfloor \frac{n}{8} \right\rfloor + \ldots + \left\lfloor \frac{n}{2^{k-1}} \right\rfloor$$

Putting this as a summation and applying the sum of a geometric series, we get

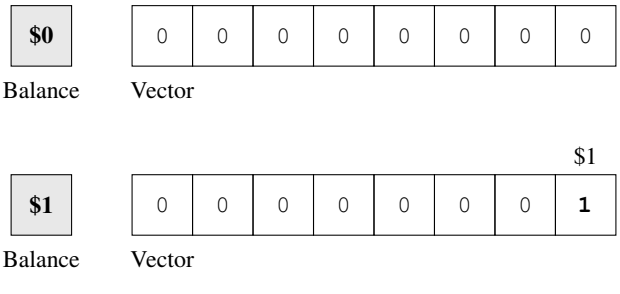
$$T(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor = n \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i = 2n\left(1 - \frac{1}{2^k}\right) \; < \; 2n$$

We can therefore conclude that a sequence of n incrementations cannot involve more than $2n$ bit flips, which is $\Theta(n)$. Using aggregate analysis, each `.increment()` call ends up with an amortized cost of

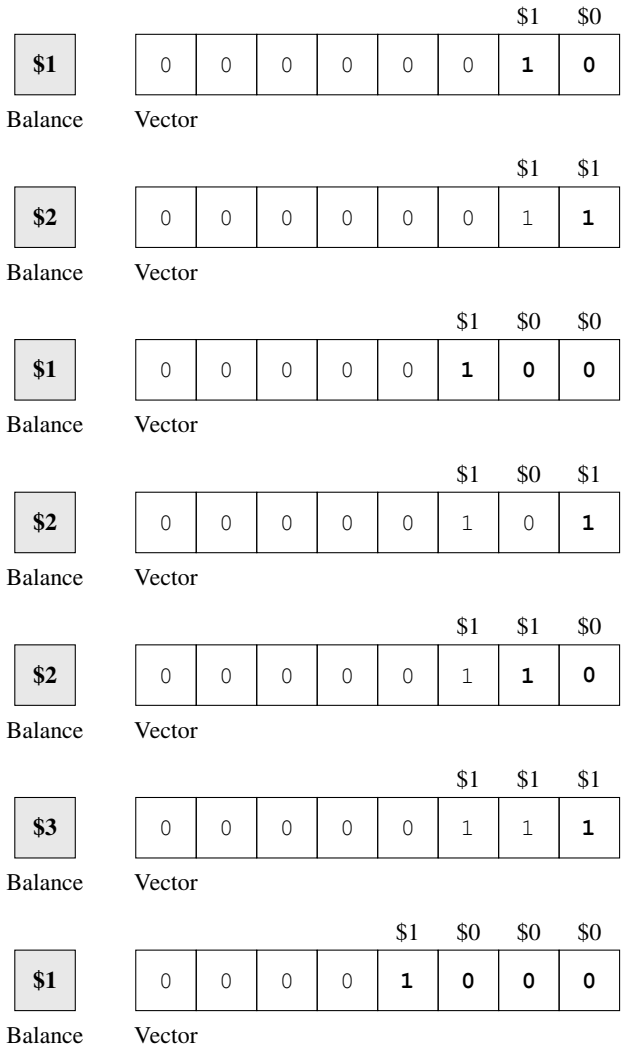
$$\text{Amortized Complexity} = \frac{T(n)}{n} = \frac{\Theta(n)}{n} = \Theta(1)$$

※ 12.6.2 *The Accounting Method* (⁎)

The accounting method and the potential method lead us to the same conclusion. Using the accounting method, we can assign an amortized cost of \$2 to every call to `.increment()`. By doing so, \$1 can be used to flip a bit from 0 to 1, and the remaining \$1 can be used to flip the bit back from 1 to 0. We can satisfy all flips this way without ever running out of credit, as shown:



[1] If you are not familiar with binary and have no idea what the table is showing, 0 in binary is 0, 1 in binary is 1, 2 in binary is 10, 3 in binary is 11, 4 in binary is 100, 5 in binary is 101, 6 in binary is 110, 7 in binary is 111, and 8 in binary is 1000. This is how the bits are flipped in the table; notice that the bits in the array are flipped from ...0000 to ...0001 to ...0010 to ...0011 to ...0100 to ...0101, and so on.



Here, every time a bit is flipped from 0 to 1, we end up charging \$2 and then using \$1 to perform the flip (with \$1 left over). This is why each bit gets assigned a \$1 credit whenever it is flipped from 0 to 1. Then, when the bit needs to be flipped back to zero, we use this extra \$1 credit to perform this flip, which is why a bit's credit drops back to \$0 when it is flipped from 1 to 0. By charging a constant \$2 per `.increment()` call, we can continue flipping bits without ever ending with a negative balance; this proves that the amortized cost of `.increment()` is $\Theta(1)$.

※ 12.6.3 *The Potential Method* (✻)

The potential method can also be used to prove that `.increment()` runs in amortized $\Theta(1)$ time. To show this, we will use the potential function $\Phi(D_i) = n_i$, where n_i represents the number of 1s in the binary number after the i^{th} increment. If we were to calculate the first few values of $\hat{c}_i$ using this potential function, we would see a pattern:

$$\begin{aligned}
\hat{c}_1 &= c_1 + \Phi(D_1) - \Phi(D_0) = c_1 + n_1 - n_0 = 1 + 1 - 0 = 2 \\
\hat{c}_2 &= c_2 + \Phi(D_2) - \Phi(D_1) = c_2 + n_2 - n_1 = 2 + 1 - 1 = 2 \\
\hat{c}_3 &= c_3 + \Phi(D_3) - \Phi(D_2) = c_3 + n_3 - n_2 = 1 + 2 - 1 = 2 \\
\hat{c}_4 &= c_4 + \Phi(D_4) - \Phi(D_3) = c_4 + n_4 - n_3 = 3 + 1 - 2 = 2 \\
\hat{c}_5 &= c_5 + \Phi(D_5) - \Phi(D_4) = c_5 + n_5 - n_4 = 1 + 2 - 1 = 2 \\
\hat{c}_6 &= c_6 + \Phi(D_6) - \Phi(D_5) = c_6 + n_6 - n_5 = 2 + 2 - 2 = 2 \\
\hat{c}_7 &= c_7 + \Phi(D_7) - \Phi(D_6) = c_7 + n_7 - n_6 = 1 + 3 - 2 = 2 \\
\hat{c}_8 &= c_8 + \Phi(D_8) - \Phi(D_7) = c_8 + n_8 - n_7 = 4 + 1 - 3 = 2 \\
\hat{c}_9 &= c_9 + \Phi(D_9) - \Phi(D_8) = c_9 + n_9 - n_8 = 1 + 2 - 1 = 2
\end{aligned}$$

In fact, we can prove that $\hat{c}_i$ will always equal 2 for all values of i. To do so, we will first define m_i as the number of bits that are flipped from 1 back to 0 on the i^{th} increment. Since exactly one bit is always flipped from 0 to 1 with each call to `.increment()`, the actual cost of the i^{th} increment must be $m_i + 1$ (m_i bits flipped from 1 to 0 by definition, and one bit flipped from 0 to 1, for a total of $m_i + 1$ bits flipped). Furthermore, the change in potential, or $\Phi(D_i) - \Phi(D_{i-1})$, must be equal to the difference between the number of bits flipped to 1 and the number of bits flipped to 0, or $1 - m_i$ (since changes in potential are determined by changes in the number of 1s in the binary number). Thus, the amortized cost of any call to `.increment()` is

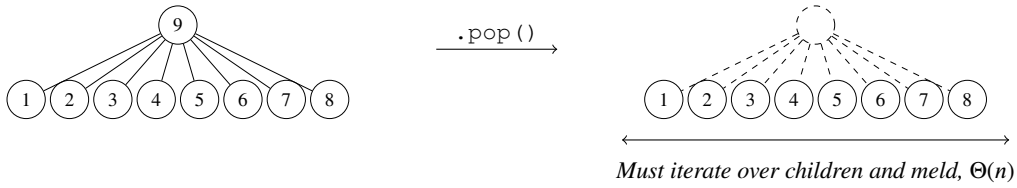
$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= (m_i + 1) + (1 - m_i) \\
&= 2
\end{aligned}$$

Since $\hat{c}_i$ is always equal to a constant 2, the amortized complexity of `.increment()` is $\Theta(1)$.

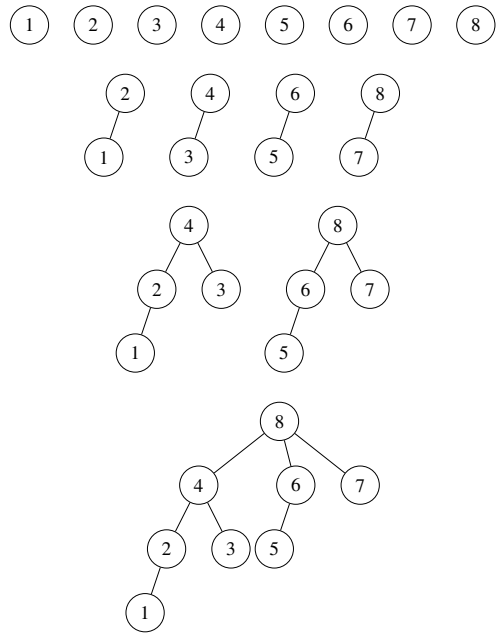
12.7 Amortized Analysis: Pairing Heap (⁎)

When we discussed the time complexities of pairing heaps back in chapter 10, we claimed that the time complexity of popping the highest priority element was amortized $O(\log(n))$. Now that we have gone over the amortization process, we can justify why this is the case.

First, let us consider the worst-case time complexity of a single pop using the pairing heap implementation discussed in chapter 10. In the worst case, every other node is a direct descendant of the root; this would require us to iterate over and meld $\Theta(n)$ subheaps to rebuild the heap after popping off the root. The worst-case time complexity of a single pop is therefore $\Theta(n)$.



However, if we use a two-pass or multipass approach, notice what happens to the remelded tree after a worst-case pop:



Even though our initial pop took $\Theta(n)$ time, the remaining nodes of the pairing heap ended up arranging themselves into a structure that allows future pops to be done in $O(\log(n))$ time. In fact, if we repeatedly insert n elements into an initially empty pairing heap, and then successively pop out all n elements until the pairing heap is empty again, the total time required to clear the heap is bounded by $O(n\log(n))$ due to the reorganization caused by melding the children using a two-pass or multipass approach. Using aggregate analysis, we can average out this $O(n\log(n))$ work over all n pops to get an amortized cost of $O(\log(n))$ for each individual pop operation:

$$\text{Amortized Complexity} = \frac{T(n)}{n} = \frac{O(n\log(n))}{n} = O(\log(n))$$

12.8 Amortized vs. Average-Case Analysis

It is very easy to mix up the concept of amortized complexity with average-case complexity, but they are *not* the same! The amortized complexity of an operation deals with its cost when *evaluated over a sequence of operations*. On the other hand, the average-case complexity of an operation represents the expected cost of a *single call* to that operation over all possible inputs.

Because an average-case analysis deals with expectation over all possible inputs, it relies on probabilistic assumptions about the data structures and inputs involved in a given algorithm or operation. To compute the average-case time complexity of an operation, you will have to look at all the possible ways an operation can be run, and then compute its expected performance over all these possible inputs.

In contrast, an amortized analysis does not need to deal with these factors. Instead of considering all the possible inputs an operation may be run on, amortized complexity simply establishes a bound on a single operation's average cost contribution when it is considered within a sequence of operations. You can think of amortized analysis as a worst-case analysis for a *sequence* of operations rather than a *single* operation: after a worst-case bound on total work is calculated for a sequence of operations, this work is averaged across all the operations performed to obtain each operation's amortized cost.

Since an amortized analysis measures the average cost of an operation over a sequence of operations, a single call to an operation may be more expensive than its amortized cost. However, if we were to run that operation enough times and average the total cost over the entire sequence of calls, we end up with that operation's amortized cost. For example, if an operation had a worst-case time complexity of $\Theta(n)$ and an amortized complexity of $\Theta(1)$, running the operation *once* may result in a $\Theta(n)$ performance, but running the operation multiple times ensures that the average performance of each operation in the sequence comes out to $\Theta(1)$.