There are three runtimes we've looked at so far. If a program $P$ runs on inputs $w \in \Sigma^*$ in time $T(w)$, then these runtimes are:

- **Worst case**: $P$ runs in worst-case time $O(f(n))$ if
  $\exists c > 0 \ \forall n \in \mathbb{N} \ \forall w \in \Sigma^*$ with $|w| = n$, $T(w) \leq cf(n)$.
  Here we look for some offensive $w$ for each $n$ that provides the upper bound.

- **Best case**: $P$ runs in best-case time $O(f(n))$ if
  $\exists c > 0 \ \forall n \in \mathbb{N} \ \exists w \in \Sigma^*$ with $|w| = n$, $T(w) \leq cf(n)$.
  Here we look for some good actor $w$ for each $n$ that is as efficient as possible.

- **Average case**: $P$ runs in average-case time $O(f(n))$ if
  $\exists c > 0 \ \forall n \in \mathbb{N}$, $E(T(w)) \leq cf(n)$.
  Here we deal with the typical input $w$ (even if such an $w$ does not exist).

Similar definitions follow for $o$, $\Omega$, $\omega$, and $\Theta$ as well.

Each of these runtimes speak to a single execution of $P$.

What if we want to know about its average runtime over a sequence of executions? This is amortized analysis.

- **Amortized case**: $P$ runs in worst-case amortized time $O(f(n))$ if
  $\exists c > 0 \ \forall n \in \mathbb{N} \ \forall w_1, w_2, \ldots, w_n \in \Sigma^*$ for which $w_1, \ldots, w_n$ is a valid sequence of inputs (often all empty strings or equivalent),
  $\frac{1}{n} \sum_{i=1}^{n} T(w_i) \leq cf(n)$.

It is possible to pull out best-case and average-case amortized definitions as well, but let's focus on the worst-case.
This definition is not easy to work with in many cases. It's also entirely useless when the inputs have unbounded size. For example, we can always ask to sort a larger array.

Usually, we perform amortized analysis only on data structures for this reason, which don't particularly care about input size (AddAll, for example, we would treat as a sequence of single Adds).

Let's try brute forcing this definition on a dynamically expanding array (this is called aggregate analysis).

```
Array
  Item[] L
  int Count

Array()
  Let L be an Item array of length 1
  Count = 0

Add(Array A, Item w)
  If |A.L| = A.Count
    Copy A.L into L'
    A.L = L'
  A.L[A.Count] = w
  A.Count += 1
```

What happens when we perform n Adds? For the sake of clarity, we will assume the Array is initially empty.

The total runtime of n Adds is (ignoring some overhead constants)

\[
1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 + 9 + \ldots
\]

\[
k=1 k=2 k=3 k=4 k=5 k=6 k=7 k=8 k=9
\]

\[
= n + 0 + 1 + 2 + 0 + 4 + 0 + 0 + 0 + 8 + \ldots
\]

\[
\log(n-1) \quad \log(n-1) + 1
\]

\[
= n + \sum_{i=0}^{\log(n-1)} 2^i = n + 2^{\log(n-1)} - 1 = n + 2n - 1 = 3n - 1.
\]

The average runtime of each of the n Adds then is

\[
\frac{3n - 1}{n} \in O(1),
\]

which is the amortized runtime of Add.
What if we added a Remove function?

Remove (Array A) =
If A[Count] ≤ \frac{1}{2}|A| + |L|
Let L' be an Item array with |L'| = \frac{1}{2}|A| + |L|
Copy A[1...|L|] into L'
A[1...|L|] = L'

Consider a sequence of operations where we add $2^m + 1$ items
($m > 0$) and then repeatedly Remove/Add $2^{m+1}$ times, here $n = 2^{m+1} + 1$, which is fine, because we're dealing with upper bounds for the WC. The total runtime of this sequence of operations (only for Add) is

$$3(2^{m+1}) - 1 + (O + 2^m + 1) + (O + 2^m + 1) + \ldots$$

$$= 3(2^{m+1}) - 1 + 2^m (2^m + 1)$$
$$= 3 \cdot 2^m + 2^m + 2^m + 2^m + 2$$
$$= 2^{2m} + 4 \cdot 2^m + 2$$

But we performed $n = 2^{m+1} + 1$ Add operations, so the amortized time of Add is

$$\frac{2^{2m} + 4 \cdot 2^m + 2}{2^{m+1}} = 2^{m-1} + 2 + \frac{1}{2^m} \in O(2^m).$$

This seems terrible, but remember that $n = 2^{m+1} + 1$, so

$O(2^m) = O(n)$. Then this Add with Remove available is worst-case amortized time $O(n)$.

This is terrible, but we can improve it! The problem was the boundary condition, so we need to massage it.
Add (Array A, Item w)
If |A.L| = A.Count
   Double the length of A.L
   A.L[A.Count] = w
   A.Count += 1
Remove (Array A)
If |A.L| ≥ 2
   Decrease the length of A.L by 2/3

The key change here is that when we contract our array, we still have 1/3 of the array to add things into. This prevents us from repeatedly expanding/contracting the array. We can still do that, of course, but it requires a linear number of operations for each expansion and contraction.

Now we could perform aggregate analysis on these Add/Remove functions much like we could shoot ourselves in the foot. Let's instead turn our attention to the accounting method. Here we assign extra amortized time to operations that may be distinct from its actual runtime. In this way items store a "credit" of time that we can expend later as free actions.

Consider the following implementation of a stack.

Stack
   Node top
Stack()
   top = null

Node
   Item w
   Node prev
   Node (Item w0, Node p)
   w = wo
   prev = p

Push (stack S, Item w)
   S.top = Node (S.top, w)
Pop (Stack S)
   If S.top is null
      Explode
      Node ret = S.top
      S.top = S.top.prev
      Return ret, w
Clearly both push and pop are $O(1)$ operations. They do nothing interesting. But what if we add a `PopK` operation?

\[ \text{PopK} \quad (\text{stack } S, \text{ int } K) \]

\[ \text{If } |S| < K, \quad \text{Pop everything} \]

\[ \text{Else} \]

\[ \text{Pop } K \text{ items} \]

The worst-case runtime of `PopK` is $O(\min(K, |S|))$ clearly, but it's amortized worst-case time is not. To see why, observe that to `pop`, we must first `push`. If we invest a credit of runtime while `pushing`, we can get the math to work out.

Define the cost of the $i^{th}$ operation to be $c_i$.
Define the amortized cost of the $i^{th}$ operation to be $\hat{c}_i$.
Then for a series of $n$ operations, so long as

\[ \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i, \]

we have a valid assignment of amortized times. In other words, our amortized total time is at least the real total time.

Define

\[ \hat{c}_{\text{push}} = c_{\text{push}} + c_{\text{pop}} + 1 \]
\[ \text{for push, } c_{\text{pop}} \text{ for pop later and } 1 \text{ for a loop iteration overhead in pop.K} \]

\[ \hat{c}_{\text{pop}} = \hat{c}_{\text{pop-K}} = 1 \]
\[ \text{for function overhead/return} \]

If we can show the inequality holds, then we have that each operation has amortized constant time.
As noted before, we can only pop if we first push. If a pop appears before a corresponding push in a sequence of operations, it only does 1 work. (or 1 1 work in the case of pop-k for the loop overhead). Thus we have

\[
\sum_{i=1}^{n} C_i = \#(\text{push}) \hat{C}_{\text{push}} + \#(\text{pop}) \hat{C}_{\text{pop}} + \#(\text{pop-k}) \hat{C}_{\text{pop-k}}
\]

\[
= \#(\text{push}) (C_{\text{push}} + C_{\text{pop}} + 1) + \#(\text{pop}) + \#(\text{pop-k})
\]

\[
= \#(\text{push}) C_{\text{push}} + (C_{\text{pop}} \#(\text{push}) + \#(\text{pop}) + \#(\text{pop-k}))
\]

\[
\geq \#(\text{push}) C_{\text{push}} + (C_{\text{pop}} \#(\text{actual pops}) + \#(\text{pop}) + \#(\text{pop-k}))
\]

\[
\geq \sum_{i=1}^{n} C_i
\]

This math got kind of messy. Usually we just hand wave the details because it’s clear what’s going on, but there’s a way to clean this up nicely. We observed that you must push before you pop, and there are often analogous operations on other data structures. For example, you must add before removing.

We can define a potential function \( \Phi \) on a data structure to take advantage of this restriction to define amortized time:

\[
\hat{C}_i = C_i + c(\Phi(D_i) - \Phi(D_{i-1}))
\]

where \( D_i \) is the data structure’s state (or just the general state of the system) after the \( i \text{th} \) operation. \( c \) here is a constant to account for overhead time in the actual operations, but since constants are irrelevant to asymptotics, we usually take \( c = 1 \) and ignore it.
With this definition, we have
\[
\sum_{i=1}^{\ell} C_i = \sum_{i=1}^{\ell} C_i + \sum_{i=1}^{\ell} \bar{\pi}(D_i) - \bar{\pi}(D_{i+1}) = \left[ \sum_{i=1}^{\ell} C_i + \sum_{i=1}^{\ell} \bar{\pi}(D_i) - \bar{\pi}(D_{i+1}) \right] \geq \sum_{i=1}^{\ell} C_i
\]

holds when \( 0 \leq \bar{\pi}(D_0) \leq \bar{\pi}(D_n) \) for all \( n \). Generally, we want to pick \( \bar{\pi} \) such that \( \bar{\pi}(D_0) = 0 \) and \( \bar{\pi} \geq 0 \).

When we do the math like this, we have the potential method, a special case of the accounting method.

Now let's apply it to our stack data structure. Pick \( \bar{\pi}(s) = |s| \),

where \( s \) is the stack. Clearly \( \bar{\pi}(s) \geq 0 \), and \( s_0 \) is the empty stack, giving us \( \bar{\pi}(s_0) = 0 \), hence \( \bar{\pi}(s_0) \leq \bar{\pi}(s_n) \) for all \( n \).

Then we get
\[
\begin{align*}
\bar{C}_{\text{push}} &= 1 + (|s| + 1) - |s| = 1 + 1 = 2 \\
\bar{C}_{\text{pop}} &= 1 + (|s| - 1) - |s| = 1 - 1 = 0 \\
\bar{C}_{\text{pop\-\-}} &= \min(|s|k) + (|s| - \min(|s|, k)) - |s| = 0.
\end{align*}
\]

Thus each operation has constant amortized time!

Let's do another example: bit addition.
Consider a binary counter
\[ b = b_n b_{n-1} \cdots b_1, \]
where the value of \( b \) is
\[ b = \sum_{i=1}^{n} b_i 2^{i-1}. \]

We have the operation Increment which adds 1 to \( b \).

\[ \text{Increment}(b) \quad // b = b_n b_{n-1} \cdots b_1 \]
\[ \text{For } i = 1 \text{ to } n \]
\[ \quad \text{If } b_i = 0 \]
\[ \quad \quad b_i = 1 \]
\[ \quad \quad \text{Return} \]
\[ \quad b_i = 0 \]
\[ b_{n+1} = 1 \]
\[ \text{Return} \]

The BC time is \( \Theta(1) \) and the WC time is \( \Theta(n) \).
The AC time is the number of initial consecutive 1s for \( b_1, b_2, b_3, \ldots \).
The arbitrary input is basically flipping \( n \) coins, so the expected number of initial consecutive 1s is
\[ \frac{1}{2} \cdot 0 + \frac{1}{4} \cdot 1 + \frac{1}{8} \cdot 2 + \cdots + \frac{1}{2^{n+1}} \cdot n \]
\[ = \sum_{i=0}^{n} 2^{-i} \]
\[ = 2^{-n} (2^{n+1} - n - 2), \]
The AC time is thus \( \Theta(1) \). This suggests that the amortized time of Increment should be \( \Theta(1) \), but we should prove this formally.

For each of the amortized-time analysis methods, do the math. Assume that \( b=0 \) initially.
**Aggregate Analysis**

The total runtime of \( n \) Increments (assume \( n \) is a power of 2, which produces the largest total time for a given \( n \)) is (ignoring constant overheads)

\[
\sum_{i=1}^{\log_2 n} \text{# of times we flip bit } i
\]

\[
= \sum_{i=1}^{\log_2 n} (2^{i-1} n) + 1
\]

\[
= 2n \sum_{i=1}^{\log_2 n} (2^{-i}) + 1
\]

\[
\leq 4n + 1
\]

Thus the average runtime of Increment is bounded above by

\[
\frac{4n+1}{n} \in \Theta(1).
\]

**Accounting Method**

Here we will do 1 extra virtual work when we flip a bit from 0 to 1 which we consume when we flip a bit from 1 to 0. This gives us

\[
\hat{c}_{\text{Increment}} = 2 \#(\text{bits we flip to 1}) - \#(\text{bits we flip to 0})
\]

Since to flip a bit from 1 to 0, we must first flip it to 1, we have

\[
\sum_{i=1}^{n} \hat{c}_{\text{Increment}} \geq \sum_{i=1}^{n} c_{\text{Increment}}.
\]

Notice, however, that each Increment flips exactly 1 bit from 0 to 1 (or adds a new bit with value 1, which is identical). This means

\[
\hat{c}_{\text{Increment}} = 2 - \#(\text{bits we flip to 0}) \leq 2,
\]

hence the amortized time of Increment is \( \Theta(1) \).
Potential Method

Define the potential function

$$\Phi(b) = \sum_{i=1}^{n} b_i = \# \text{ of } 1's \text{ in } b.$$  

This gives us $$\Phi(0) = 0$$ and $$\Phi(b) \geq \Phi(0)$$ for all $$b$$.

Thus the amortized time of Increment is

$$\hat{C}_{\text{Increment}} = C_{\text{Increment}} + \Phi(b+1) - \Phi(b)$$

$$= 1 + \#(\text{bits we flip to 0}) + \sum_{i=1}^{n} (b+1)_i - \sum_{i=1}^{n} b_i.$$  

But

$$\sum_{i=1}^{n} (b+1)_i - \sum_{i=1}^{n} b_i = \#(\text{bits we flip to 0}) + 1.$$  

Thus

$$\hat{C}_{\text{Increment}} = 1 + \#(\text{bits we flip to 0}) - \#(\text{bits we flip to 0}) + 1$$

$$= 2,$$

hence Increment’s amortized time is $$\Theta(1).$$