Data Abstractions
More Recurrences

\[ T(n) \]

\[ T(n/2) \]

\[ T(n/4) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]

\[ T(1) \]
Solving the reverse Recurrence

\[ T(n) = \begin{cases} 
  d_0 & \text{if } n = 0 \\
  c_0 + c_1 n + T(n-1) & \text{otherwise}
\end{cases} \]

\[ T(n) = (c_0 + c_1 n) + T(n-1) \]
\[ = (c_0 + c_1 n) + (c_0 + c_1 (n-1)) + T(n-2) \]
\[ = (c_0 + c_1 n) + (c_0 + c_1 (n-1)) + (c_0 + c_1 (n-2)) + \ldots + (c_0 + c_1 (1)) + d_0 \]
\[ = \sum_{i=0}^{n-1} (c_0 + c_1 (n-i)) + d_0 \]
\[ = \sum_{i=0}^{n-1} c_0 + \sum_{i=0}^{n-1} c_1 (n-i) + d_0 \]
\[ = nc_0 + c_1 \sum_{i=1}^{n} i + d_0 \]
\[ = nc_0 + c_1 \left( \frac{n(n+1)}{2} \right) + d_0 \]
\[ = \mathcal{O}(n^2) \]
A recurrence where we solve some constant piece of the problem (e.g. “-1”, “-2”, etc.) is called a **Linear Recurrence**.

We solve these like we did above by **Unrolling the Recurrence**.

This is a fancy way of saying “plug the definition into itself until a pattern emerges”.

Now, back to mergesort.
Analyzing Merge Sort

Merge Sort

```java
sort(L) {
    if (L.size() < 2) {
        return L;
    }
    else {
        int mid = L.size() / 2;
        return merge(
            sort(L.subList(0, mid)),
            sort(L.subList(mid, L.size()))
        );
    }
}
```

First, we need to find the recurrence:

\[
T(n) = \begin{cases} 
  d_0 & \text{if } n = 0 \\
  d_1 & \text{if } n = 1 \\
  c_0 + c_1 n + 2T(n/2) & \text{otherwise} 
\end{cases}
\]

This recurrence isn’t linear! This is a “divide and conquer” recurrence.
Analyzing Merge Sort

\[ T(n) = \begin{cases} 
  d_0 & \text{if } n = 0 \\
  d_1 & \text{if } n = 1 \\
  c_0 + c_1 n + 2T(n/2) & \text{otherwise}
\end{cases} \]

This time, there are multiple possible approaches:

**Unrolling the Recurrence**

\[
T(n) = (c_2 + c_1 n) + 2(c_2 + c_1 n + 2T(n/4)) \\
= (c_2 + c_1 n) + 2(c_2 + c_1 n + 2(c_2 + c_1 n + 2T(n/8))) \\
= c_2 + 2c_2 + 4c_2 + ... + argh + ...
\]

This works, but I’d rarely recommend it.

**Insight:** We’re branching in this recurrence. So, represent it as a tree!
$$T(n) = \begin{cases} 
  d_0 & \text{if } n = 0 \\
  d_1 & \text{if } n = 1 \\
  c_0 + c_1n + 2T(n/2) & \text{otherwise} 
\end{cases}$$
Merge Sort: Solving the Recurrence

\[ T(n) = \begin{cases} 
  d_0 & \text{if } n = 0 \\
  d_1 & \text{if } n = 1 \\
  c_0 + c_1 n + 2T(n/2) & \text{otherwise} 
\end{cases} \]
$T(n) = \begin{cases} 
  d_0 & \text{if } n = 0 \\
  d_1 & \text{if } n = 1 \\
  c_0 + c_1 n + 2T(n/2) & \text{otherwise} 
\end{cases}$
Merge Sort: Solving the Recurrence

\[ T(n) = \begin{cases} 
    d_0 & \text{if } n = 0 \\
    d_1 & \text{if } n = 1 \\
    c_0 + c_1 n + 2T(n/2) & \text{otherwise}
\end{cases} \]
$$T(n) = \begin{cases} 
  d_0 & \text{if } n = 0 \\
  d_1 & \text{if } n = 1 \\
  c_0 + c_1 n + 2T(n/2) & \text{otherwise}
\end{cases}$$

Since the recursion tree has height $\lg(n)$ and each row does $n$ work, it follows that $T(n) \in \mathcal{O}(n\lg(n))$. 
Find A Big-Oh Bound For The Worst Case Runtime

```python
sum(n) {
    if (n < 2) {
        return n;
    }
    return 2 + sum(n - 2);
}
```

$$T(n) = \begin{cases} 
  d_0 & \text{if } n = 0 \\
  d_0 & \text{if } n = 1 \\
  c_0 + T(n-2) & \text{otherwise}
\end{cases}$$

$$T(n) = c_0 + c_0 + \cdots + c_0 + d_0$$

$$= c_0 \left( \frac{n}{2} \right) + d_0$$

$$= \mathcal{O}(n)$$
Find A Big-Oh Bound For The Worst Case Runtime

```java
binarysearch(L, value) {
    if (L.size() == 0) {
        return false;
    }
    else if (L.size() == 1) {
        return L[0] == value;
    }
    else {
        int mid = L.size() / 2;
        if (L[mid] < value) {
            return binarysearch(L.subList(mid + 1, L.size()), value);
        }
        else {
            return binarysearch(L.subList(0, mid), value);
        }
    }
}
```

\[ T(n) = \begin{cases} 
  d_0 & \text{if } n = 0 \\
  d_1 & \text{if } n = 1 \\
  c_0 + T(n/2) & \text{otherwise} 
\end{cases} \]
$T(n) = \begin{cases} 
    d_0 & \text{if } n = 0 \\
    d_1 & \text{if } n = 1 \\
    c_0 + T(n/2) & \text{otherwise} 
\end{cases}$

So, $T(n) = c_0(\lg(n) - 1) + d_1 = O(\lg n)$. 
Consider a recurrence of the form:

\[ T(n) = \begin{cases} 
  d & \text{if } n = 1 \\
  aT\left(\frac{n}{b}\right) + n^c & \text{otherwise}
\end{cases} \]

Then,

- If \( \log_b(a) < c \), then \( T(n) = \Theta(n^c) \).
- If \( \log_b(a) = c \), then \( T(n) = \Theta(n^c \log(n)) \).
- If \( \log_b(a) > c \), then \( T(n) = \Theta(n^{\log_b(a)}) \).

**Sanity Check:** For Merge Sort, we have \( a = 2, b = 2, c = 1 \). Then, \( \log_2(2) = 1 = 1 \). So, \( T(n) = n \log n \).
Proving the First Case of Master Theorem

\[ T(n) = \begin{cases} 
  d & \text{if } n = 1 \\
  aT\left(\frac{n}{b}\right) + n^c & \text{otherwise}
\end{cases} \]

We assume that \( \log_b(a) < c \). Then, unrolling the recurrence, we get:

\[ T(n) = n^c + aT\left(\frac{n}{b}\right) \]
\[ = n^c + a\left((\frac{n}{b})^c + aT\left(\frac{n}{b^2}\right)\right) \]
\[ = n^c + a(n/b)^c + a^2(n/b^2)^c + \cdots + a^{\log_b(n)}\left(n/b^{\log_b n}\right)^c \]
\[ = \sum_{i=0}^{\log_b(n)} a^i \left(\frac{n^c}{b^{ic}}\right) \]
\[ = n^c \sum_{i=0}^{\log_b(n)} \left(\frac{a}{b^c}\right)^i \]
\[ = n^c \left(\left(\frac{a}{b^c}\right)^{\log_b(n)+1} - 1\right) \approx n^c \left(\left(\frac{a}{b^c}\right)^{\log_b(n)}\right) \approx n^c \]
CSE 332
Data Abstractions
Amortized Analysis

That new guy in class?
Yeah, I thought he was a hipster, too.
Turns out, he’s an accountant.
### Stack ADT

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push(val)</code></td>
<td>Adds <code>val</code> to the stack.</td>
</tr>
<tr>
<td><code>pop()</code></td>
<td>Returns the <strong>most-recent</strong> item not already returned by a <code>pop</code>. (Errors if empty.)</td>
</tr>
<tr>
<td><code>peek()</code></td>
<td>Returns the <strong>most-recent</strong> item not already returned by a <code>pop</code>. (Errors if empty.)</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>Returns true if all inserted elements have been returned by a <code>pop</code>.</td>
</tr>
</tbody>
</table>

Let’s analyze the time complexity for these various methods. (You know how they work, because you just implemented them!)

<table>
<thead>
<tr>
<th>Method</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isEmpty()</code></td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>peek()</code></td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>pop()</code></td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><code>push(val)</code></td>
<td>??</td>
</tr>
</tbody>
</table>

`push` is actually slightly more interesting.
Analyzing push for an ArrayStack

Best Case

Worst Case

Insight: Our analysis seems wrong. Saying linear time feels wrong.
Analyzing push for an ArrayStack

Best Case

There’s more space in the underlying array! Then, it’s $\Omega(1)$.

Worst Case

Insight: Our analysis seems wrong. Saying linear time feels wrong.
Analyzing push for an ArrayStack

Best Case
There’s more space in the underlying array! Then, it’s \( \Omega(1) \).

Worst Case
If there’s no more space, we double the size of the array, and copy all the elements. So, it’s \( O(n) \).

Insight: Our analysis seems wrong. Saying linear time feels wrong.
Analyzing push for an ArrayStack

This is where “amortized analysis” comes in. Sometimes, we have a very rare expensive operation that we can “charge” to other operations.

Intuition: Rent, Tuition

You pay one big sum for a long period of time, but you can afford it because it happens very rarely.

Back to ArrayStack

Say we have a full Stack of size $n$. Then, consider the next $n$ pushes:

- The next push will take $O(n)$ (to resize the array to size $2n$)
- The $n - 1$ operations after that will all be $O(1)$, because we know we have enough space

Considering these operations in aggregate, we have $n$ operations that take $(c_0 + c_1 n) + (n - 1) \times c_2$ time.

So, how long does each operation take:

$$
\frac{(c_0 + c_1 n) + (n - 1) \times c_2}{n} \leq \frac{n \max(c_0, c_2) + c_1 n}{n} = \max(c_0, c_2) + c_1 = O(1)
$$
Analyzing push for an ArrayStack

What happens if we change our resize rule to each of the following:

- $n \rightarrow n + 1$
- $n \rightarrow \frac{3n}{2}$
- $n \rightarrow 5n$

Which is better $2n$, $\frac{3n}{2}$, or $5n$?

Java uses $\frac{3n}{2}$ to minimized wasted space.
Analyzing push for an ArrayStack

What happens if we change our resize rule to each of the following:

- **$n \to n + 1$**
  
  This is really bad! We can only amortize over the single operation which gives us:

  \[
  \frac{n}{1} = O(n)
  \]

- **$n \to \frac{3n}{2}$**

- **$n \to 5n$**

Which is better $2n$, $\frac{3n}{2}$, or $5n$?

Java uses $\frac{3n}{2}$ to minimized wasted space.
Analyzing `push` for an `ArrayStack`

What happens if we change our resize rule to each of the following:

- $n \to n + 1$
  
  This is really bad! We can only amortize over the single operation which gives us:
  
  $$\frac{n}{1} = O(n)$$

- $n \to \frac{3n}{2}$
  
  This still works. Now, we go over the next $\frac{3n}{2} - n$ operations:
  
  $$\frac{n + (n/2 - 1) \times 1}{\frac{n}{2}} = O(1)$$

- $n \to 5n$

Which is better $2n$, $\frac{3n}{2}$, or $5n$?

Java uses $\frac{3n}{2}$ to minimized wasted space.
Analyzing push for an ArrayStack

What happens if we change our resize rule to each of the following:

- $n \to n + 1$
  This is really bad! We can only amortize over the single operation which gives us:
  \[
  \frac{n}{1} = O(n)
  \]

- $n \to \frac{3n}{2}$
  This still works. Now, we go over the next $\frac{3n}{2} - n$ operations:
  \[
  \frac{n + \left(\frac{n}{2} - 1\right) \times 1}{\frac{n}{2}} = O(1)
  \]

- $n \to 5n$
  This is good too:
  \[
  \frac{n + (4n - 1) \times 1}{4n} = O(1)
  \]

Which is better $2n$, $\frac{3n}{2}$, or $5n$?

Java uses $\frac{3n}{2}$ to minimized wasted space.