Amortized Analysis
- average: expected cost probabilistically.
- aggregate: total cost over x operations
- amortized: \( \frac{\text{aggregate}}{\# \text{ ops.}} \)

ex: rolling dice \( \rightarrow \) average case is 3.5

drawing 52 cards from a deck \( \rightarrow \) aggregate value = \( 13 + 12 + \ldots + 2 + 1 \) * 4

amortized value = \( \frac{\text{aggregate value}}{52} \) = 7

Straightforward analysis
for each operation - determine its cost

\( T(n) = \sum_{i=1}^{n} \text{operation } i \)

amortized = \( \frac{T(n)}{n} \)

to generalize - determine pattern of costs for operations.

i.e. all ops have a constant cost except when \( i = 2^k \) \( k = 0, 1, 2 \ldots \)

then the operation costs \( 2i = 2^{k+1} \).

**EX** BINARY COUNTER

<table>
<thead>
<tr>
<th>counter value</th>
<th>binary representation</th>
<th>i\text{th} step cost</th>
<th>total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>$\phi$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>2</td>
<td>18</td>
</tr>
</tbody>
</table>

pattern:
\( \frac{n}{n + \frac{n}{2} + \frac{n}{4} + \ldots + 1}{n(1 + \frac{1}{2} + \frac{1}{4} + \ldots)} \leq 2n \)
thus the aggregate cost is $2n = O(n)$ and the amortized cost is $2 = \Theta(1)$

**The Accounting Method**

We "charge" some amount for each operation - the charge is independent of its cost! (if you want)
we pay the actual cost of the operation.

idea: store credit for the times when an operation is expensive like insurance!
never let your balance drop below zero.
useful when you have varying operations that contribute to each other's running times.

example in book: stack (push pop multipop)
ex: binary counter
charge $2$ for each insert
- at most one 1 is added - costs $1$
- every 1 has a dollar "stored" on it for when it is flipped back to zero.
- all flips from 1 -> 0 use previously stored money.

---

**DYNAMIC TABLES**

From class:

\[ \alpha = \frac{n}{m} \]

must be below a threshold of 4/5

<table>
<thead>
<tr>
<th>want</th>
<th>$m = \Theta(n)$ at all times</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_0 = 5 )</td>
<td>( m_1 = 10 )</td>
</tr>
</tbody>
</table>

\[ \alpha = \frac{4}{5} \]
\[ \alpha = \frac{2}{5} \]
\[ \alpha = \frac{4}{5} \]
\[ \alpha = \frac{2}{5} \]
\[ \alpha = \frac{4}{5} \]

... etc.

Table growth during a series of insert operations:

(shading represents \( \alpha \), not actual locations of values in table)

aggregate: \( i = 2^k \rightarrow \text{cost}(i) = \Theta(2^k) \) else \( \text{cost}(i) = 1 \)
\[ \text{aggregate cost } (i) = \begin{cases} 1 & \text{use } i = 2^k \\ 2^k & \text{else} \end{cases} \Rightarrow T(n) = 2^k + 2^{k-1} + 2^{k-2} + \ldots + n \]
\[ = n + 2^k \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) \leq n + 2^k (2) \]
and \(n = 2^k\) so
\[ = 8n \]

\[ \text{amortized } \frac{2n}{n} = 3 = \Theta(1) \]

**Deletions**

Same idea in reverse \( \chi \geq \frac{11}{5} \) at all times.
- Why not \(\frac{2}{5}\)? \(k\)-then ins, del, ins, del at boundary very expensive.

<table>
<thead>
<tr>
<th>Table Shrinkage during a series of delete operations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k)</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**aggregate:** when table reaches \(\frac{1}{5}m\) (or \(2^k\) in general) we need to shrink
otherwise just delete element

\[ \text{cost } (i) = \begin{cases} 2^k & i = 2^k \\ 1 & \text{else} \end{cases} \Rightarrow T(n) = \text{same as above} \]

(assuming \(n\) deletions brings us to an empty list)

**Deletions and Insertions:** worst case

\[ \begin{array}{cccccccc}
M=5 & m=10 & \uparrow & DD & \downarrow & \uparrow & DD & \downarrow & \ldots \\
\text{Worst case series of operations.} \\
\text{cost: } 1 \ 1 \ 1 \ 4 \ 1 \ 1 \ 2 \ 1 \ 1 \ 4 \ 1 \ 1 \ 2 \ \ldots \\
\text{Repeating block cost= 6+4=10} \\
\text{ops = 6} \\
\text{amortized } = \frac{10}{6} = \frac{5}{3} \\
\end{array} \]
Rabin Karp

goal: string search = find pattern \( p \) in string \( S \).

idea: treat each letter as a digit in base \( b \).
then the pattern is a number in base \( b \) of length \( \text{len}(p) \) digits.
see if that number occurs in \( S \) by a rolling hash.
use a simple modulo hash.

- String of digits: \( b = 10 \)
  find [123] in \( 87124612349001 \)
  \[ h(p) = 123 \mod 13 \]
  \[ 815135 \]
  \[ S = 10 \]
  \[ 8151356103548 \]

when you find a potential match: check \( p \) vs \( S \) at that spot to verify.
- need to compute \( h(S[i\cdots j]) \) from \( h(S[i-1\cdots j-1]) \) quickly.
  this is where rolling hashes come in:

\[
\begin{align*}
  a b c d & \Rightarrow 10^3 a + 10^2 b + 10 c + d = h(s_1) \\
  a b c d & \Rightarrow 10^4 a + 10^3 b + 10^2 c + 10 d = 10 h(s_1) \\
  a b c d e & \Rightarrow 10^4 a + 10^3 b + 10^2 c + 10 d + e = 10 h(s_1) + e
\end{align*}
\]

so \( h(s_2) = 10 h(s_1) + e - 10^4 a \)
which is linear.

work through example above.

- general case
\[
( h(\text{new}) = b h(\text{old}) + d_{\text{new}} - b^\text{len}(p) d_{\text{old}} ) \mod m
\]

- time: \( O(\text{len}(p)) \) to compute hash
  \( O(\text{len}(p) + n) \) to compute hashes in \( S \)
  \( O(\text{len}(p) \cdot \# \text{potential matches}) \) to verify potential match

- binary example
\[
\begin{array}{c}
001 \\
\end{array}
\text{in } \begin{array}{c}
1100110110000110 \\
\end{array} \mod 3