Roughly how much time are you spending on your CS 42 assignment each week?
(define (remove e L)
    (if (empty? L)
        empty
        (let* ([it (first L)]
               [lose-it (rest L)]
               [lose-it-solution (remove e lose-it)]
               [use-it-solution (cons it (remove e lose-it))])
            (if (equal? e it)
                lose-it-solution
                use-it-solution))))
Base case(s):
\[ e \ '() \Rightarrow \ '() \]

\[ F = \text{remove} \]

\[ F( e \ '() \Rightarrow \ '() ) \]

if \( e \neq \text{it} \) then:

else:

solution
How “good” are these solutions?
Are they efficient?
Do they “cost” more than they should?
Interpreting a theoretical model

Key take-away: it’s lossy!

A theory abstracts away certain details.

cost metric:

- corresponds to one “step”
- highlights the essence of the work
e.g., multiplications, comparisons, function calls…
- serves as a proxy for an empirical measurement

Instead of measuring time, we count steps.
e.g., “This algorithm costs $n^2$ multiplications.”
Asymptotic Analysis

(Big O)
Asymptotic analysis

We’re always answering the same question:

How does the cost \textit{scale} (when we try larger and larger inputs)?

\textbf{Not:}

- Exactly how many steps will it execute?
- How many seconds will it take?
- How many megabytes of memory will it need?
The informal definition of “Big O”

A *reasonable* upper bound on (an abstraction of) a problem’s difficulty or a solution’s performance, for *reasonably* large input sizes.
In the limit (for VERY LARGE inputs)

The running time is bounded regardless of the input size. \(O(1)\)

An input twice as big takes no more than twice as long. \(O(n)\)

An input twice as big takes no more than four times as long. \(O(n^2)\)

An input one bigger takes no more than twice as long. \(O(2^n)\)
If We Only Care About Scalability...

What are the consequences?

Constant factors can be ignored.

\[ n \] and \[ 6n \] and \[ 200n \] scale identically ("linearly")

Small summands can be ignored.

\[ n^2 \] and \[ n^2 + n + 999999 \] are indistinguishable when \( n \) is huge.
Grouping Algorithms by Scalability

O(1)
- takes 6 steps
- takes 1 (big) step
- no more than 4000 steps
- somewhere between 2 and 47 steps, depending on the input

O(n)
- takes 100n + 3 steps
- takes n/20 + 10,000,000 steps
- anywhere between 3 and 68 steps per item, for n items.

O(n^2)
- takes 2n^2 + 100n + 3 steps
- takes n^2/17 steps
- somewhere between 1 and 40 steps per item, for n^2 items
- anywhere between 1 and 7n steps per item, for n items.
How hard is the problem?

- $O(n^n)$
- $O(n!)$
- $O(2^n)$
- $O(n^3)$
- $O(n^2)$
- $O(n \log(n))$
- $O(n)$
- $O(\sqrt{n})$
- $O(\log(n))$
- $O(1)$

**Intractable problems** (exponential)

- $O(n^n)$
- $O(n!)$
- $O(2^n)$

**Tractable problems** (polynomial)

- $O(n^3)$
- $O(n^2)$
- $O(n \log(n))$
- $O(n)$
- $O(\sqrt{n})$
- $O(\log(n))$

**No problem!**
logs aren’t scary!
They’re our friends.

\[ \log_2 N = p \iff 2^p = N \]
log is the inverse of exponentiation.

How many times can I cut N in half?
Can I avoid looking at all the input?!

\[
\begin{align*}
\log_2(1) &= 0 \quad // 2^0 = 1 \\
\log_2(2) &= 1 \quad // 2^1 = 2 \\
\log_2(3) &\approx 1.58 \\
\log_2(4) &= 2 \quad // 2^2 = 4 \\
\log_2(5) &\approx 2.32 \\
\log_2(6) &\approx 2.58 \\
\log_2(7) &\approx 2.81 \\
\log_2(8) &= 3 \quad // 2^3 = 8
\end{align*}
\]
## How hard are these problems?

<table>
<thead>
<tr>
<th>cost metric</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>double</strong></td>
<td>multiplications</td>
</tr>
<tr>
<td>multiply a number by 2</td>
<td></td>
</tr>
<tr>
<td><strong>sum</strong></td>
<td>additions</td>
</tr>
<tr>
<td>sum a list of numbers</td>
<td></td>
</tr>
<tr>
<td><strong>half-count</strong></td>
<td>divisions</td>
</tr>
<tr>
<td>divide a positive number by 2 until you get 1</td>
<td></td>
</tr>
</tbody>
</table>
### How hard are these problems?

<table>
<thead>
<tr>
<th><strong>cost metric</strong></th>
<th><strong>cost</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>double</strong></td>
<td></td>
</tr>
<tr>
<td>multiply a number by 2</td>
<td>multiplications</td>
</tr>
<tr>
<td><strong>sum</strong></td>
<td></td>
</tr>
<tr>
<td>sum a list of numbers</td>
<td>additions</td>
</tr>
<tr>
<td><strong>half-count</strong></td>
<td></td>
</tr>
<tr>
<td>divide a positive number by 2 until you get 1</td>
<td>divisions</td>
</tr>
</tbody>
</table>
What’s the cost, T, for each solution?

\[
\begin{array}{|c|c|c|c|}
\hline
\text{input size} & \text{double multiplications} & \text{sum additions} & \text{half-count divisions} \\
\hline
T(0) & & & n/a \\
T(1) & & & \\
T(2) & & & \\
T(3) & & & \\
T(4) & & & \\
\vdots & & & \\
T(n) & & & \\
\hline
\end{array}
\]

(\text{define (double n)}
\begin{align*}
& (* n 2) \\
\end{align*})

(\text{define (sum n)}
\begin{align*}
& (if (= n 0) \\
& \quad 0 \\
& \quad (+ n (sum (- n 1)))) \\
\end{align*})

(\text{define (half-count n)}
\begin{align*}
& (if (= n 1) \\
& \quad 0 \\
& \quad (+ 1 (half-count (\text{quotient n 2})))))
\end{align*})
What’s the cost, $T$, for each solution?

<table>
<thead>
<tr>
<th>input size</th>
<th>double multiplications</th>
<th>sum additions</th>
<th>half-count divisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(0)$</td>
<td>1</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>$T(1)$</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$T(2)$</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$T(3)$</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$T(4)$</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$T(n)$</td>
<td>1</td>
<td>$n$</td>
<td>$\lfloor \log_2 n \rfloor$</td>
</tr>
</tbody>
</table>

Can we prove it?
https://www.simplelongboards.com/boards/platypus/
Recurrence Relations
(translating code to math)
Translating recursion to recurrence relations

For a given cost metric: additions

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size $N$
   Define $T(N)$ recursively, in terms of smaller cost.

\[
\text{(define (sum n)}
\begin{cases}
  \text{if } (= \ n \ 0) & 0 \\
  (+ \ n \ (\text{sum} \ (- \ n \ 1))))
\end{cases}
\]

\[
\begin{align*}
T(0) &= 1 \\
T(N) &= 3 + T(N-1)
\end{align*}
\]
Translating recursion to recurrence relations

For a given cost metric: additions

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define $T(N)$ recursively, in terms of smaller cost.

\[
T(N) = 1 + T(N-1)
\]

\[
T(N) = 1 + 1 + T(N-2)
\]

\[
T(N) = 1 + 1 + 1 + T(N-3)
\]

\[
\vdots
\]

\[
T(N) = 1 + 1 + 1 + \ldots + 1 + T(N-N)
\]

\[
= 1 \cdot 1 + T(N-1)
\]

\[
= 2 \cdot 1 + T(N-2)
\]

\[
= 3 \cdot 1 + T(N-3)
\]

\[
\vdots
\]

\[
= N \cdot 1 + T(N-N) = N \in O(N)
\]
Translating recursion to recurrence relations

For a given cost metric: arithmetic operations and comparisons

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define T(N) recursively, in terms of smaller cost.

\[
(\text{define } (\text{sum } n) \\
(\text{if } (= \ n \ 0) \\
\quad 0 \\
\quad (+ \ n \ (\text{sum } (- \ n \ 1)))))
\]

\[
\begin{align*}
T(0) & = 0 \\
T(N) & = 2 + T(N-1)
\end{align*}
\]
Translating recursion to recurrence relations

For a given cost metric: arithmetic operations and comparisons

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define T(N) recursively, in terms of smaller cost.

\[
T(N) = 3 + T(N-1) = 1 \cdot 3 + T(N-1) = 2 \cdot 3 + T(N-2) = 3 \cdot 3 + T(N-3) = N \cdot 3 + T(N-N) = N \cdot 3 + T(N-N) = 3N + 1 \in O(N)
\]

(base case) \rightarrow (recursive case) \rightarrow (recurrence relation)
Translating recursion to recurrence relations

For a given cost metric: divisions

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define T(N) recursively, in terms of smaller cost.

\[
\text{(define (half-count n)} \quad \text{base case} \rightarrow \quad \begin{align*}
T(1) &= 1 \\
T(N) &= 3 + T(N-1)
\end{align*}
\text{recursive case} \rightarrow \quad \text{recurrence relation}
\text{(if (= n 1)} \quad \begin{align*}
0 \\
(+ 1 (\text{half-count (quotient n 2)})�)
\end{align*}
\text{input size)}
\]
Translating recursion to recurrence relations

For a given cost metric: **divisions**

1. Translate the base case(s), using specific **input sizes**
   How many steps does this base case take?

2. Translate the recursive case(s), using **input size** \(N\)
   Define \(T(N)\) recursively, in terms of smaller cost.

\[
\begin{align*}
T(1) &= 0 \\
T(N) &= 1 + T(N/2)
\end{align*}
\]

\[
\begin{align*}
T(N) &= 1 + T(N/2) \\
&= 1 + 1 + T(N/4) \\
&= 1 + 1 + 1 + T(N/8) \\
&\ldots \\
&= 1 + 1 + 1 + \ldots 1 + T(N/N) \\
&= \log_2 N + T(N/N) = \log_2 N \in O(\log N)
\end{align*}
\]
<table>
<thead>
<tr>
<th></th>
<th>cost metric</th>
<th>work to do</th>
<th>predicted cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove</td>
<td>number of comparisons made</td>
<td>visit every element</td>
<td>O(n)</td>
</tr>
<tr>
<td>uniq</td>
<td>number of comparisons made</td>
<td>compare each element to all the other elements</td>
<td>O(n²)</td>
</tr>
<tr>
<td>sublists</td>
<td>number of sublists created</td>
<td>construct 2ⁿ lists</td>
<td>O(2ⁿ)</td>
</tr>
</tbody>
</table>
The cost of remove measured in list-element comparisons

\[
\text{(define (remove e L)}
\text{ (if (empty? L)}}
\text{ empty)
\text{ (let* ([it (first L)]
\text{ [lose-it (rest L)]
\text{ [lose-it-solution (remove e lose-it)]
\text{ [use-it-solution (cons it lose-it-solution)]])
\text{ (if (equal? e it)}
\text{ lose-it-solution
\text{ use-it-solution)))))
\]

\[
T(0) = 0
\]
\[
T(N) = 1 + T(N-1)
\]
\[
\in O(N)
\]
The cost of `uniq` measured in list-element comparisons

\[
\text{Th} 
\begin{align*}
T(0) &= 0 \\
T(N) &= N-1 + T(N-1) \\
&= N-1 + N-2 + T(N-2) \\
&\in O(N^2)
\end{align*}
\]

(\text{define} \ (\text{uniq} \ L) \\
(\text{if} \ (\text{empty?} \ L) \ ('()) \\
(\text{let*} \ ([\text{it} \ (\text{first} \ L)] \\
[\text{lose-it} \ (\text{rest} \ L)] \\
[\text{lose-it-soln} \ (\text{uniq} \ \text{lose-it})] \\
[\text{use-it-soln} \ (\text{cons} \ \text{it} \ \text{lose-it-soln})]) \\
(\text{if} \ (\text{member} \ \text{it} \ \text{lose-it-soln}) \\
\text{lose-it-soln} \\
\text{use-it-soln})))))
The cost of sublists measured in number of sublists created, i.e., calls to \texttt{cons} and \texttt{empty}

\begin{align*}
T(0) &= 1 \\
T(N) &= 2^{N-1} + 2^{N-1} + T(N-1) \\
&= 2^N + T(N-1)
\end{align*}

\[ T(N) \in O(2^N) \]
## Problems vs solutions

<table>
<thead>
<tr>
<th></th>
<th>cost metric</th>
<th>work to do</th>
<th>predicted cost</th>
<th>UloLI cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>remove</strong></td>
<td>number of comparisons made</td>
<td>visit every element</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>uniq</strong></td>
<td>number of comparisons made</td>
<td>compare each element to all the other elements</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
</tr>
<tr>
<td><strong>sublists</strong></td>
<td>number of sublists created</td>
<td>construct 2^n lists</td>
<td>O(2^n)</td>
<td>O(2^n)</td>
</tr>
</tbody>
</table>