What is a “good” program?
In English, describe what this function does.
I’m sorry for this terrible code! It gets better, I promise.

(define (f x y)
  (if (equal? y '())
      #t
    (if (equal? (x (first y)) #t)
      (f x (rest y))
      #f)))

(Your response)
Given a computational problem
Is there a solution?
What is it?
How good is it?
Is it correct?
Is it good for humans?
This is some bad functional programming.

But it’s about as efficient as can be!

(define (f x y) (if (equal? y '()) #t (if (equal? (x (first y)) #t) (f x (rest y)) #f)))
Whitespace helps humans.

(define (f x y)
  (if (equal? y '()) #t
   (if (equal? (x (first y)) #t)
       (f x (rest y)) #f))
Comments help humans.

;; f: returns true if every element of a list matches a predicate
;;   inputs:
;;     x (boolean function of one argument)
;;     y (a list)
;;   outputs:
;;     true if every list element satisfies the predicate
;;     (returns true for the empty list)

(define (f x y)
  (if (equal? y '())
      #t
      (if (equal? (x (first y)) #t)
          (f x (rest y))
          #f)))
Tests help humans.

;; f: returns true if every element of a list matches a predicate
;;   inputs:
;;     x (boolean function of one argument)
;;     y (a list)
;;   outputs:
;;     true if every list element satisfies the predicate
;;     (returns true for the empty list)

(define (f x y)
  (if (equal? y '())
      #t
      (if (equal? (x (first y)) #t)
          (f x (rest y))
          #f)))

(check-expect (f odd? '()) true)
(check-expect (f odd? '(1 2 3 4)) false)
(check-expect (f odd? '(1 3)) true)
Good names help humans.

;; all: returns true if every element of a list matches a predicate
;; inputs:
;;  predicate (boolean function of one argument)
;;  L (a list)
;; outputs:
;;  true if every list element satisfies the predicate
;;  (returns true for the empty list)

(define (all predicate L)
  (if (equal? L '())
      true
      (if (equal? (predicate (first L)) true)
          (all predicate (rest L))
          false)))

(check-expect (all odd? '()) true)
(check-expect (all odd? '(1 2 3 4)) false)
(check-expect (all odd? '(1 3)) true)
Conditions help humans.

;; all: returns true if every element of a list matches a predicate
;;   inputs:
;;       predicate (boolean function of one argument)
;;       L (a list)
;;   outputs:
;;       true if every list element satisfies the predicate
;;       (returns true for the empty list)

(define (all predicate L)
  (if (empty? L)
      true
      (if (predicate (first L))
        (all predicate (rest L))
        false)))

(check-expect (all odd? '()) true)
(check-expect (all odd? '(1 2 3 4)) false)
(check-expect (all odd? '(1 3)) true)
Logical structures help humans.

;; all: returns true if every element of a list matches a predicate
;;   inputs:
;;       predicate (boolean function of one argument)
;;       L (a list)
;;   outputs:
;;       true if every list element satisfies the predicate
;;       (returns true for the empty list)

(define (all predicate L)
  (cond [[(empty? L) true]
           [(not (predicate (first L))) false]
           [else (all predicate (rest L))]]))

(check-expect (all odd? '()) true)
(check-expect (all odd? '(1 2 3 4)) false)
(check-expect (all odd? '(1 3)) true)
Functional programming helps humans.

;; all: returns true if every element of a list matches a predicate
;;   inputs:
;;     predicate (boolean function of one argument)
;;     L (a list)
;;   outputs:
;;     true if every list element satisfies the predicate
;;     (returns true for the empty list)

(define (all predicate L)
  (equal? L (filter predicate L)))

(check-expect (all odd? '()) true)
(check-expect (all odd? '(1 2 3 4)) false)
(check-expect (all odd? '(1 3)) true)
How would you implement negate?

(define (negate f)

(all good? L1)

(all (negate good?) L2))
Functions can return functions, too!

(define (negate f)
  (lambda (x) (not (f x))))

(all good? L1)
(all (negate good?) L2)
Is it efficient?
Decidability
e.g., Can it be solved at all?

Complexity Class
e.g., Can it be solved in polynomial time?

Asymptotic Complexity
e.g., $O(n)$ time, where $n$ is the number of Cows

Exact Theory
e.g., $7n + 2$ milkings, where $n$ is the number of Cows

Empirical Data
e.g., This run took 17.3 seconds on this data.
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
Data: which algorithm is best?

Lower is better
At Xcomp ’15, Professor Bauer of UCNY announces that the new sorting algorithm *WildSort* takes only 0.16 seconds to sort a list of 100,000 names.

Later, at the same conference, Professor Taylor of SUNY SD reports the new algorithm *SneakerSort* takes only 0.03 seconds to sort a list of 100,000 names.

Your conclusion(s)?
Interpreting empirical data

We can measure

- a particular **algorithm**
- written in a particular **language**
- as a particular **program**
- compiled using a particular version of a particular **compiler**
- with particular **settings** (e.g., enabling / disabling optimizations)
- running on a particular **data set**, of a particular **size**
- on a particular **computer**
- with particular **resources** (CPUs, memory, hard drive, …)
- under a particular version of a particular **operating system**
- in a particular **environment**
  e.g., with other programs running in the background
Empirical data + ???
Interpreting a theoretical model

A theory abstracts away certain details

- which details?
- what do we lose?
- what do we gain?
Theory: which algorithm is better?

Suppose we have two sorting algorithms, where for $n$ items

*Algorithm 1* does this many comparisons: $3(nH(n) + 2n)$

*Algorithm 2* does this many comparisons: $\frac{n(n + 1)}{2}$
Theory: which algorithm is better?

Lower is better

\[3(nH(n) + 2n)\]

\[\frac{n(n + 1)}{2}\]
Theory: which algorithm is better?

Lower is better

\[
\frac{n(n+1)}{2}
\]

\[
3(nH(n) + 2n)
\]

Time

Problem Size
Takeaway lessons

- Empirical data + Good theory = Meaning
- Empirical data + No theory = ???
- Empirical data + Naïve theory = Danger!
Decidability
  e.g., Can it be solved at all?

Complexity Class
  e.g., Can it be solved in polynomial time?

Asymptotic Complexity
  e.g., $O(n)$ time, where $n$ is the number of Cows

Exact Theory
  e.g., $7n + 2$ milkings, where $n$ is number of Cows

Empirical Data
  e.g., This run took 17.3 seconds on this data.
Exact theory
From code to math
What's the cost \( T(n) \) for each function?

Where cost is measured in “number of calls” to that function

<table>
<thead>
<tr>
<th>Function</th>
<th>( T(0) )</th>
<th>( T(1) )</th>
<th>( T(2) )</th>
<th>( T(3) )</th>
<th>( T(4) )</th>
<th>( T(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( n/a )</td>
</tr>
<tr>
<td>fact</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>mystery</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>( n + 1 )</td>
</tr>
</tbody>
</table>
| 1 + \( T(n-1) \) | \( 1 + T(n/2) \) | Recurrence relation

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

\begin{align*}
(\text{define (fact n)}) \\
(\text{if} \ (= \ n \ 0) \\
\text{1} \\
(* \ n \ (\text{fact} \ (-n \ 1))))
\end{align*}

\begin{align*}
(\text{define (mystery n)}) \\
(\text{if} \ (= \ n \ 1) \\
\text{0} \\
(+ \ 1 \ (\text{mystery} \ (\text{quotient} \ n \ 2))))
\end{align*}
Decidability
e.g., Can it be solved at all?

Complexity Class
e.g., Can it be solved in polynomial time?

Asymptotic Complexity
e.g., $O(n)$ time, where $n$ is the number of Cows

Exact Theory
e.g., $7n + 2$ milkings, where $n$ is number of Cows

Empirical Data
e.g., This run took 17.3 seconds on this data.
Asymptotic complexity

What happens as input grows without bounds?
The informal definition of “Big O”

A *reasonable* upper bound on (an abstraction of) an algorithm’s performance, for *reasonably* large input sizes.
How hard is the problem?

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

**Intractable problems (exponential)**

**Tractable problems (polynomial)**

**No problem!**
The (in)formal definition of Big O

\[ f(n) \in O(g(n)) \text{ if and only if} \]

there exist positive constants \( c \) and \( N \), such that

\[ 0 \leq f(n) \leq c \cdot g(n) \]

for all \( n \geq N \)

ignoring constant-time speedups…
The (in)formal definition of Big O

\[ f(n) \in O(g(n)) \text{ if and only if} \]

there exist positive constants \( c \) and \( N \), such that
\[ 0 \leq f(n) \leq c \cdot g(n) \]
for all \( n \geq N \)
What’s the $O(n)$ for each function?
where cost is measured in “number of calls” to that function

<table>
<thead>
<tr>
<th></th>
<th>double</th>
<th>fact</th>
<th>mystery</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(0)$</td>
<td>1</td>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>$T(1)$</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$T(2)$</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$T(3)$</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>$T(4)$</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>$T(n)$</td>
<td>1</td>
<td>$n + 1$</td>
<td>$1 + \log_2 n$</td>
</tr>
</tbody>
</table>

$O(1)$  $O(n)$  $O(\log n)$
What's the $O(n)$ for each function? where cost is measured in “number of total function calls”

<table>
<thead>
<tr>
<th></th>
<th>double</th>
<th>fact</th>
<th>mystery</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(0)$</td>
<td>2</td>
<td>2</td>
<td>n/a</td>
</tr>
<tr>
<td>$T(1)$</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$T(2)$</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>$T(3)$</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>$T(4)$</td>
<td>2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>$T(n)$</td>
<td>2</td>
<td>$n + 2$</td>
<td>$2(1 + 2\log_2 n)$</td>
</tr>
</tbody>
</table>

$O(1)$  $O(n)$  $O(\log n)$
Embedded vs Tail Recursion
(code optimizations)
Recursion

\[ n! = \begin{cases} 
  1 & : n = 0 \\
  n \times (n - 1)! & : \text{otherwise}
\end{cases} \]

(define (fact N)
  (if (= N 0)
    1
    (* N (fact (- N 1))))

Is this tail recursion?
Embedded recursion

(define (fact N)
  (if (= N 0)
      1
      (* N (fact (- N 1)))))

Refactor function so that it uses an accumulator

Tail recursion

(define (tail-fact N)
  (tail-fact-helper N 1))

(define (tail-fact-helper N accum)
  (if (equal? N 0)
      accum
      (tail-fact-helper (- N 1) (* accum N))))
Tail recursion

Refactor function so that it uses an accumulator

(define (tail-fact N)
  (tail-fact-helper N 1))

(define (tail-fact-helper N accum)
  (if (equal? N 0)
      accum
      (tail-fact-helper (- N 1) (* accum N))))

base case
recursive step
Use trace to help investigate / debug
Is it good for humans?
Volkswagen Emissions Cheat Exploited 'Test Mode'

The common mode enables the car to enable a different mode in driving cycles, a "clean" mode and "real" mode. The "real" mode will then be used to record data for the emissions tests.

One key part of the unfolding Volkswagen emissions scandal is two different modes: "On Road" and "Real Life." The configurations are known as "clean" and "real" modes.

The 428,000 Volkswagen and Audi cars sold in the United States are set up for emissions testing using a special, "clean" mode. The "real" mode is used to record data for the emissions tests.

Volkswagen's emissions cheating likely caused dozens of deaths in the US

Seth Borenstein, Associated Press

WASHINGTON (AP) - Volkswagen's pollution cheating is not a victimless tinkerer. The company's emissions cheating has likely caused tens of thousands of premature deaths in the United States and global warming, according to a new study

By Irene Chapple and Mark Thompson

Volkswagen HQ raided by German police

Rupert Murdoch: Ben Carson would be a 'real black president'