member

fun member _ [] = false
  | member e (x::xs) = e=x orelse (member e xs);

What is it's type signature?

What does it do?

determines if the first argument is in the second argument
For a list with $k$ elements in it, how many calls are made to `member`?

Depends on the input!

Worst case is when the item doesn’t exist in the list $k+1$ times:
- each element will be examined one time (2nd pattern)
- plus one time for the empty list

How will the run-time grow as the list size increases?

Linearly:
- for each element we add to the list, we’ll have to make one more recursive call
- doubling the size of the list would roughly double the run-time

How fast is it?

For a list with $k$ elements in it, how many calls are made to `member` in the worst case?

Type signature? What do they do? Which is faster? How much faster?
How many calls to \texttt{member} are made for a list of size \(k\), including calls made in uniquify0 as well as recursive calls made in \texttt{member}?

Depends on the values!

\textbf{Worst case}, how many calls to \texttt{member} are made for a list of size \(k\), including calls made in uniquify0 as well as recursive calls made in \texttt{member}?

How many calls are made if the list is empty? \(0\)

Recursive case: Let \(\text{count}_0(i)\) be the number of calls that uniquify0 makes to member for a list of size \(i\).

Can you define the number of calls for a list of size \(k\) \((\text{count}_0(k))\)? Hint: the definition will be recursive?
Recursive case: Let \( \text{count}_0(i) \) be the number of calls that \( \text{uniquify0} \) makes to \( \text{member} \) for a list of size \( i \).

\[
\text{count}_0(k) = (k + 1) + \text{count}_0(k - 1)
\]

Worst case number of calls for 1 call to \( \text{member} \) of size \( k \)
Number of calls for \( \text{uniqify0} \) on a list of size \( k - 1 \)

Recurrence relation

\[
\text{count}_0(k) = \begin{cases} 
0 & \text{if, } k = 0 \\
 k + \text{count}_0(k - 1) & \text{otherwise} 
\end{cases}
\]

\( \text{count}_0(k) = k(k + 1) \) \( \frac{k^2}{2} \) calls to \( \text{member} \)

Can you prove this?
Proof by induction

1. State what you’re trying to prove!
2. State and prove the base case
   - What is the smallest possible case you need to consider?
     Should be fairly easy to prove
3. Assume it’s true for k (or k-1). Write out specifically what this assumption is (called the inductive hypothesis).
4. Prove that it then holds for k+1 (or k)
   a. State what you’re trying to prove (should be a variation on step 1)
   b. Prove it. You will need to use inductive hypothesis.

Proof by induction! \(\text{count}_0(k) = \begin{cases} 0, & \text{if } k = 0 \\ \frac{k(k+1)}{2}, & \text{otherwise} \end{cases}\)

1. \(\text{count}_0(k) = \frac{k(k+1)}{2}\)
2. Base case?

Proof by induction!

1. \(\text{count}_0(k) = \frac{k(k+1)}{2}\)
2. Base case?
3. Assume: \(\text{count}_0(k-1) = \) Inductive hypothesis

Proof by induction!

\[\text{count}_0(k) = \begin{cases} 0, & \text{if } k = 0 \\ \frac{k(k+1)}{2}, & \text{otherwise} \end{cases}\]

1. \(\text{count}_0(k) = \frac{k(k+1)}{2}\)
2. Base case?
3. Assume: \(\text{count}_0(k-1) = \) Inductive hypothesis
Proof by induction! \( \text{count}_i(k) = \begin{cases} 0 & \text{if } k = 0 \\ k \cdot \text{count}_i(k-1) & \text{otherwise} \end{cases} \)

3. assume: \( \text{count}_i(k-1) = \frac{(k-1)k}{2} \) inductive hypothesis

4. prove: \( \text{count}_i(k) = \frac{k(k+1)}{2} \) by definition of count

\[
\begin{align*}
count_i(k) &= k \cdot count_i(k-1) \\
         &= k \cdot \frac{(k-1)k}{2} \\
         &= \frac{2k^2 - k}{2} \\
         &= \frac{k^2 + k}{2} \\
         &= \frac{k(k+1)}{2}
\end{align*}
\]

Done!

uniquify1

```haskell
fun uniquify1 [] = []
  | uniquify1 (x:xs) =
    if member x (uniquify1 xs)
    then uniquify1 xs
    else x:(uniquify1 xs);
```

What is the recurrence relation for calls to member for uniquify1? Write a recursive function called count, that gives the number of calls to member for a list of size k.

\[
\begin{align*}
count_i(k) &= \begin{cases} 0 & \text{if } k = 0 \\ k \cdot count_i(k-1) & \text{otherwise} \end{cases}
\end{align*}
\]
uniquify1

fun uniquify1 [ ] = []
| uniquify1 (x::xs) =
| if member x (uniquify1 xs)
| then uniquify1 xs
| else x::(uniquify1 xs);

\[ count_1(k) = \begin{cases} 
0 & \text{if, } k = 0 \\
2 \cdot count_1(k-1) & \text{otherwise} 
\end{cases} \]

How many calls is that?

\[ count_1(k) = \begin{cases} 
0 & \text{if, } k = 0 \\
1 + 2 \cdot count_1(k-1) & \text{otherwise} 
\end{cases} \]

I claim: \[ count_1(k) = 2^{k+1} - k - 1 \]

Can you prove it?

Prove it!

1. State what you're trying to prove!
2. State and prove the base case
3. Assume it's true for k (or k-1) (and state the inductive hypothesis!)
4. Show that it holds for k+1 (or k)

\[ count_1(k) = \begin{cases} 
0 & \text{if, } k = 0 \\
2 \cdot count_1(k-1) & \text{otherwise} 
\end{cases} \]

Proof by induction!

1. \[ count_1(k) = 2^{k+1} - k - 2 \]
2. Base case:
Proof by induction! $\text{count}(k) = \begin{cases} 0 & \text{if } k = 0 \\ k + 2 \cdot \text{count}(k - 1) & \text{otherwise} \end{cases}$

1. $\text{count}(k) = 2^{k+1} - k - 2$

2. Base case: $k = 0$
   
   $\text{count}(k) = 0$
   
   from definition of count,

   $\text{count}(k) = $

3. Assume: $\text{count}(k - 1) = 2^k - (k - 1) - 2$

   by inductive hypothesis

4. Prove: $\text{count}(k) = 2^{k+1} - k - 2$

   $\text{count}(k) = k + 2 \cdot \text{count}(k - 1)$
   
   by definition of count,

   $\text{count}(k) = k + 2(2^k - k - 1)$
   
   by inductive hypothesis

   $\text{count}(k) = k + 2^{k+1} - 2k - 2$
   
   math (multiply through by 2)

   $\text{count}(k) = 2^{k+1} - k - 2$
   
   math

   \text{done!}
Does it matter?

\[
\text{count}_0(k) = \frac{k(k+1)}{2} \quad \text{count}_1(k) = 2^{k+1} - k - 2
\]

\[
\begin{array}{c|cccccccccc}
    k & 0 & 1 & 2 & 3 & 4 & \ldots & 10 & \ldots & 100 \\
    \text{count}_0(k) & 0 & 1 & 2 & 3 & 4 & \ldots & 10 & \ldots & 100 \\
\end{array}
\]

Does it matter?

\[
\text{count}_0(k) = \frac{k(k+1)}{2} \quad \text{count}_1(k) = 2^{k+1} - k - 2
\]

\[
\begin{array}{c|cccccccccc}
    k & 0 & 1 & 2 & 3 & 4 & \ldots & 10 & \ldots & 100 \\
    \text{count}_0(k) & 0 & 1 & \ldots & 10 & \ldots & 100 \\
    \text{count}_1(k) & 0 & 1 & ? & \ldots & ? & \ldots & ? \\
\end{array}
\]
Does it matter?

\[
\begin{align*}
\text{count}_0(k) &= \frac{k(k+1)}{2} \\
\text{count}_1(k) &= 2^{k+1} - k - 2
\end{align*}
\]

<table>
<thead>
<tr>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>10</th>
<th>...</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>count(_0)(k)</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td>...</td>
<td>10</td>
<td>...</td>
<td>?</td>
</tr>
<tr>
<td>count(_1)(k)</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td>...</td>
<td>10</td>
<td>...</td>
<td>?</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{count}_0(k) &= \frac{k(k+1)}{2} \\
\text{count}_1(k) &= 2^{k+1} - k - 2
\end{align*}
\]

<table>
<thead>
<tr>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>10</th>
<th>...</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>count(_0)(k)</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>15</td>
<td>...</td>
<td>55</td>
<td>...</td>
<td>5050</td>
</tr>
<tr>
<td>count(_1)(k)</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>57</td>
<td>...</td>
<td>2036</td>
<td>...</td>
<td>2.5 \times 10^{30}</td>
</tr>
</tbody>
</table>

Maybe it's not that bad

2.5 \times 10^{30} \text{ calls to member for a list of size 100}

Roughly how long will that take?

Maybe it's not that bad

2.5 \times 10^{30} \text{ calls to member for a list of size 100}

- Assume 10^9 \text{ calls per second}
- \approx 3 \times 10^7 \text{ seconds per year}
- \approx 3 \times 10^{17} \text{ calls per year}
- \approx 10^{13} \text{ years to finish!}
  
  Just to be clear: 1,000,000,000,000 years

In practice

On my laptop, starts to slow down with lists of length 22 or so

Undo

fun unique1 [] = []
unique1 (x::xs) =
  if member x (unique1 xs)
  then unique1 xs
  else x::(unique1 xs);

What's the problem?
Can we fix it?
Undo

fun unify1 [] = []
  | unify1 (x::xs) =
      if member x (unify1 xs)
      then unify1 xs
      else x::(unify1 xs);

fun unify2 [] = []
  | unify2 (x::xs) =
      let val recResult = unify2 xs;
      in
        if member x recResult
        then recResult
        else x::recResult
      end;

Which is faster?

fun unify@ [] = []
  | unify@ (x::xs) =
      if member x xs
      then unify0 xs
      else x::(unify@ xs);

fun unify2 [] = []
  | unify2 (x::xs) =
      let val recResult = unify2 xs;
      in
        if member x recResult
        then recResult
        else x::recResult
      end;

Big O: Upper bound

\[ O(g(n)) \] is the set of functions:

\[
O(g(n)) = \left\{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \right\}
\]

We can bound the function \( f(n) \) above by some constant factor of \( g(n) \); constant factors don't matter!
Big O: Upper bound

\( O(g(n)) \) is the set of functions:

\[
O(g(n)) = \left\{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0 \right\}
\]

We can bound the function \( f(n) \) above by some constant factor of \( g(n) \): constant factors don’t matter!

For some increasing range: we’re interested in long-term growth

Visually

Visually: upper bound

Big-O

member is \( O(n) \) – linear
- \( n+1 \) is \( O(n) \)

unify0 is \( O(n^2) \) – quadratic
- \( n(n+1)/2 = n^2/2 + n/2 \) is \( O(n^2) \)

unify1 is \( O(2^n) \) – exponential
- \( 2^{n+1} - n + 2 \) is \( O(2^n) \)

unify2 is \( O(n^2) \) – quadratic
Runtime examples

<table>
<thead>
<tr>
<th>n</th>
<th>n</th>
<th>n log n</th>
<th>n^2</th>
<th>2^n</th>
<th>n^3</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 10</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>n = 30</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
</tr>
<tr>
<td>n = 100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>10^3 years very long</td>
</tr>
<tr>
<td>n = 1000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
</tr>
<tr>
<td>n = 10,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
</tr>
<tr>
<td>n = 100,000</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
</tr>
<tr>
<td>n = 1,000,000</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
</tr>
</tbody>
</table>

(adapted from [5], Table 2.1, pg. 34)

Some examples

- O(1) – constant. Fixed amount of work, regardless of the input size
  - add two 32 bit numbers
  - determine if a number is even or odd
  - sum the first 20 elements of an array
  - delete an element from a doubly linked list

- O(log n) – logarithmic. At each iteration, discards some portion of the input (i.e. half)
  - binary search

- O(n) – linear. Do a constant amount of work on each element of the input
  - find an item in an array (unsorted) or linked list
  - determine the largest element in an array

- O(n log n) log-linear. Divide and conquer algorithms with a linear amount of work to recombine
  - Sort a list of number with MergeSort
  - FFT

- O(n^2) – quadratic. Double nested loops that iterate over the data
  - Insertion sort

- O(2^n) – exponential
  - Enumerate all possible subsets
  - Traveling salesman using dynamic programming

- O(n!) – enumerate all permutations
  - determinant of a matrix with expansion by minors
STOPPED HERE

This is as far as I made it in lecture. There are two additional examples of proofs by induction that I won’t cover, but I’ll leave them in the notes in case you want to see more examples.

An aside

My favorite thing in python!

What do these functions do?

```python
def fibrec(n):
    if n == 1:
        return 1
    else:
        return fibrec(n-2) + fibrec(n-1)

def fibiter(n):
    prev2, prev1 = 0, 1
    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)
    return prev1
```

Runtime

```python
def fibrec(n):
    if n == 1:
        return 1
    else:
        return fibrec(n-2) + fibrec(n-1)

def fibiter(n):
    prev2, prev1 = 0, 1
    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)
    return prev1
```

Which is faster?
What is the big-O runtime of each function in terms of n, i.e. how does the runtime grow w.r.t. n?
**Runtime**

```
def fibiter(n):
    prev2, prev1 = 0, 1
    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)
    return prev1
```

**Informal justification:**
The for loop does n iterations and does just a constant amount of work for each iteration. An increase in n will see a corresponding increase in the number of iterations.

---

**Runtime**

```
def fibrec(n):
    if n == 1:
        return 1
    else:
        return fibrec(n-2) + fibrec(n-1)
```

**Guess?**

```
f(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
1 + f(n-2) + f(n-1) & \text{otherwise}
\end{cases}
```

Slightly different than the recurrence relation for uniquify1.
Proof:

\[
f(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
1 + f(n-2) + f(n-1) & \text{otherwise}
\end{cases}
\]

We want to prove that \( f(n) \) is \( O(2^n) \)
Show that \( f(n) \leq 2^n - 1 \)

Why is this sufficient?

\[ O(g(n)) = \begin{cases} 
f(n) : & \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0 
\end{cases}\]

Proof by induction:

\[
f(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
1 + f(n-2) + f(n-1) & \text{otherwise}
\end{cases}
\]

We want to prove that \( f(n) \) is \( O(2^n) \)
Show that \( f(n) \leq 2^n - 1 \)

\[ f(n) \leq 2^n - 1 \leq 2^n \text{ for all } n \geq 0 \]

\[ O(g(n)) = \begin{cases} 
f(n) : & \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0 
\end{cases}\]

1. Prove: \( f(n) \leq 2^n - 1 \)
2. Base case:
\[
n = 1
\]
\[
f(1) = 1 \quad \text{by definition of } f(n)
\]
\[
f(1) = 2^1 - 1 = 1 \quad \text{by math}
\]
Proof by induction \( f(n) = \begin{cases} \frac{1}{1 + f(n-2) + f(n-1)} & \text{if } n \neq 1 \\ \text{otherwise} \end{cases} \)

1. Prove: \( f(n) \leq 2^n - 1 \)

3. Inductive hypothesis:

   Assume: \( f(n) \leq 2^n - 1 \)

4. Prove:

   \( n + 1 \): \( f(n+1) \leq 2^{n+1} - 1 \)
We proved that $f(n)$ is $O(2^n)$

Is this sufficient to prove that $f(n)$ takes an exponential amount of time?

No. This is only an upper bound!

Most of the time, this is what we’re worried about, talking about bounding the running time of our algorithm, i.e. no worse than.

How would we prove that $f(n)$ is exponential, i.e. always takes exponential time?

Using induction, can prove $f(n) \geq \frac{1}{2} 2^{\frac{n}{2}}$
Base cases

\[
\text{fibrec}(n) = \text{fibiter}(n)
\]

def fibiter(n):
    prev1, prev2 = 0, 1
    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)
    return prev1

n = 0 and n = 1

Loop doesn't execute at all
prev1 = 1 and is returned

n = 0: 1

Base cases

n = 0: 1
n = 1: 1
Inductive hypotheses

Assume:
- $\text{fibrec}(n-1) = \text{fibiter}(n-1)$
- $\text{fibrec}(n-2) = \text{fibiter}(n-2)$

Prove:
- $\text{fibrec}(n) = \text{fibiter}(n)$

Definition of for loops

Assume:
- $\text{fibiter}(n-2) = \text{fibrec}(n-2)$
- $\text{fibiter}(n-1) = \text{fibrec}(n-1)$

Prove:
- $\text{fibiter}(n) = \text{fibrec}(n)$

What is $\text{prev1}$ after this?

Assume:
- $\text{fibiter}(n-2) = \text{fibrec}(n-2)$
- $\text{fibiter}(n-1) = \text{fibrec}(n-1)$

Prove:
- $\text{fibiter}(n) = \text{fibrec}(n)$

prev1 = $\text{fibiter}(n-2)$

by inductive hypothesis:

prev1 = $\text{fibrec}(n-2)$
Assume: \( \text{fibiter}(n-2) = \text{fibrec}(n-2) \)
\( \text{fibiter}(n-1) = \text{fibrec}(n-1) \)

Prove: \( \text{fibiter}(n) = \text{fibrec}(n) \)

Assume:
\[
\text{fibiter}(n-2) = \text{fibrec}(n-2)
\text{fibiter}(n-1) = \text{fibrec}(n-1)
\]

Prove:
\[
\text{fibiter}(n) = \text{fibrec}(n)
\]

What is \( \text{prev2} \) after this?

prev2 = \( \text{fibrec}(n-2) \)

by inductive hypothesis

prev1 = \( \text{fibrec}(n-1) \)

What is \( \text{prev1} \) after this?

prev2 = \( \text{fibrec}(n-2) \)
prev1 = \( \text{fibrec}(n-1) \)

prev1 = \( \text{fibrec}(n-2) \)

by inductive hypothesis

prev1 = \( \text{fibrec}(n-1) \)

prev2 = \( \text{fibrec}(n-2) \)
prev1 = \( \text{fibrec}(n-1) \)

Done!