member

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

What is its type signature?

What does it do?

member

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

'a -> 'a list -> bool

Determines if the first argument is in the second argument
How fast is it?

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

Uniquify

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

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

 Depends on the values!

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

How many calls are made if the list is empty?

0

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

Can you define the number of calls for a list of size \( k \) \( (\text{count}_k[k]) \)? Hint: the definition will be recursive?
uniquify0

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

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

worst case number of calls for
1 call to member of size \( k \)
number of calls for uniquify0
on a list of size \( k - 1 \)

Recurrence relation:

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

How many calls is this?

Recurrence relation:

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

1. \( \text{count}_0(k) = \begin{cases} 
0 & \text{if } k = 0 \\
\frac{k(k+1)}{2} & \text{otherwise} 
\end{cases} \)
2. \( \text{base case?} \)
3. \( \text{assume: } \text{count}_0(k-1) = \) Inductive hypothesis

Proof by induction!

1. \( \text{count}_0(k) = \begin{cases} 
0 & \text{if } k = 0 \\
\frac{k(k+1)}{2} & \text{otherwise} 
\end{cases} \)
2. \( \text{base case?} \)
3. \( \text{assume: } \text{count}_0(k-1) = \) Inductive hypothesis
1. \( \text{count}_0(k) = \frac{k(k+1)}{2} \) \quad \text{count}_0(k) = \begin{cases} 0 & \text{if } k = 0 \\ k + \text{count}_0(k-1) & \text{otherwise} \end{cases}

3. assume: \( \text{count}_0(k-1) = \frac{(k-1)k}{2} \) \quad \text{inductive hypothesis}

4. prove:

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

Proof by induction!

1. \( \text{count}_0(k) = \frac{k(k+1)}{2} \) \quad \text{count}_0(k) = \begin{cases} 0 & \text{if } k = 0 \\ k + \text{count}_0(k-1) & \text{otherwise} \end{cases}

3. assume: \( \text{count}_0(k-1) = \frac{(k-1)k}{2} \) \quad \text{inductive hypothesis}

4. prove:

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

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

inductive hypothesis

4. prove:

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

Done!

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.

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

Fun uniquify1 nil = nil
| 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.

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

Donel
uniquify1

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

\[
\begin{align*}
\text{count}(k) &= \begin{cases} 
0 & \text{if}, k = 0 \\
2k + 1 \text{ count}(k-1) & \text{otherwise}
\end{cases}
\end{align*}
\]
```

How many calls is that?

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

I claim: \(\text{count}(k) = 2^{k+1} - k - 2\)

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 \(k\))

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

Proof by induction!

1. \(\text{count}(k) = 2^{k+1} - k - 2\)
2. Base case: \(k = 0\)
   \(\text{count}(k) = 0\)

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

from definition of count,
what we’re trying to prove
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: \( k = 0 \)
   \( \text{count}(k) = 0 \)
   \( \text{count}(k) = 2^0 - 0 - 2 = 0 \)

From definition of \( \text{count} \), what we’re trying to prove.

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

4. Prove: \( \text{count}(k) = 2^{k+1} - k - 2 \)
   By definition of \( \text{count} \),
   \( \text{count}(k) = k + 2 \cdot \text{count}(k-1) \)
   \( = k + 2(2^k - k - 1) \)
   \( = k + 2^{k+1} - 2k - 2 \)
   \( = 2^{k+1} - k - 2 \) Done!

---

**Proof by induction!**

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

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

4. Prove: \( \text{count}(k) = 2^{k+1} - k - 2 \)
   By definition of \( \text{count} \),
   \( \text{count}(k) = k + 2 \cdot \text{count}(k-1) \)
   \( = k + 2(2^k - k - 1) \)
   \( = k + 2^{k+1} - 2k - 2 \)
   \( = 2^{k+1} - k - 2 \) Done!

---

**Does it matter?**

\[
\text{fun unify0 nil} = \text{nil}\\
\text{unify0 (x:xs)} = \\
\text{if member x xs} \\
\text{then unify0 xs} \\
\text{else x::(unify0 xs);}
\]

\[
\text{fun unify1 nil} = \text{nil}\\
\text{unify1 (x:xs)} = \\
\text{if member x (unify1 xs)} \\
\text{then unify1 xs} \\
\text{else x::(unify1 xs);}
\]
### Does it matter?

<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>( \text{count}_0(k) )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>...</td>
<td>10</td>
<td>...</td>
<td>100</td>
</tr>
<tr>
<td>( \text{count}_1(k) )</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>...</td>
<td>?</td>
<td>...</td>
<td>?</td>
</tr>
</tbody>
</table>
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>(\ldots)</th>
<th>(10)</th>
<th>(\ldots)</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>\ldots</td>
<td>55</td>
<td>\ldots</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>\ldots</td>
<td>2036</td>
<td>\ldots</td>
<td>(2.5 \times 10^{30})</td>
</tr>
</tbody>
</table>

Maybe it's not that bad

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

<table>
<thead>
<tr>
<th>(k)</th>
<th>(0)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(\ldots)</th>
<th>(10)</th>
<th>(\ldots)</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>\ldots</td>
<td>55</td>
<td>\ldots</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>\ldots</td>
<td>2036</td>
<td>\ldots</td>
<td>(2.5 \times 10^{30})</td>
</tr>
</tbody>
</table>

2.5 x 10^{30} calls to member for a list of size 100

Roughly how long will that take?
Maybe it's not that bad

2.5 x 10^{30} calls to member for a list of size 100

- Assume 10^9 calls per second
- ~3 x 10^7 seconds per year
- ~3 x 10^{17} calls per year
- ~10^{13} years to finish!
  Just to be clear: 10,000,000,000,000 years

In practice

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

In practice

In practice

In practice

In practice

What’s the problem?
Can we fix it?

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

```haskell
fun uniquify2 nil = nil
    | uniquify2 (x::xs) =
      | let recResult = uniquify2 xs;
      | in
      | if member x recResult
      | then recResult
      | else x::recResult
      | end;
```

```haskell
fun uniquify3 nil = nil
    | uniquify3 (x::xs) =
      | if x = recResult
      | then x::recResult
      | else x::uniquify3 xs;  
```
Which is faster?

Big O: Upper bound

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

$O(g(n)) = \{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } \forall n \geq n_0, 0 \leq f(n) \leq c g(n) \}$

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 $O(n)$

uniquify0 is $O(n^2)$ – quadratic
- $n(n+1)/2 = n^2/2 + n/2$ is $O(n^2)$

uniquify1 is $O(2^n)$ – exponential
- $2^{n+1} - n - 2$ is $O(2^n)$

uniquify2 is $O(n^2)$ – quadratic

Runtime examples

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

(adapted from [3], 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

Some examples

- $O(n)$ – 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

Some examples

- $O(n^2)$ – 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

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
```

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

```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
```

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.
**NOTE**

I did not cover the following proof in class, but left it in the notes as another example of an inductive proof.
We want to prove that \( f(n) \) is \( O(2^n) \)

Show that \( f(n) \leq 2^n - 1 \)

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

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

We want to prove that \( f(n) \) is \( O(2^n) \)

Show that \( f(n) \leq 2^n - 1 \)

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

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

**Proof by induction**

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

2. Base case:
   
   \( n = 1 \)
   
   \( f(1) = 2^1 - 1 = 1 \)
   
   What we're trying to prove

3. Inductive hypothesis:

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

4. Prove:

   \( n+1 \):
   
   \( f(n+1) \leq 2^{n+1} - 1 \)
Proof by induction

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

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

**Prove:** \( f(n+1) \leq 2^{n+1} - 1 \)

**Inductive hypothesis:**

\[ f(n+1) = 1 + f(n-1) + f(n) \leq 2^{n-1} + 2^n - 1 \]

What do we do with it?

---

**Proof by induction**

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

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

**Prove:** \( f(n+1) \leq 2^{n+1} - 1 \)

**Inductive hypothesis:**

\[ f(n) \leq 2^n - 1 \]

\[ f(n-1) \leq 2^{n-1} - 1 \]

**Prove:**

\[ f(n+1) \leq 2^{n+1} - 1 \]

---

**Proof by induction**

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

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

**Prove:** \( f(n+1) \leq 2^{n+1} - 1 \)

**Inductive hypothesis:**

\[ f(n) \leq 2^n - 1 \]

\[ f(n-1) \leq 2^{n-1} - 1 \]

**Prove:**

\[ f(n+1) \leq 2^{n+1} - 1 \]

---

**Proving exponential runtime**

\[ \Omega(g(n)) = \begin{cases} 
\text{there exists positive constants } c \text{ and } n_0 \text{ such that } \\
f(n) - \text{if } f(n) \leq cg(n) \text{ for all } n \geq n_0 \\
\end{cases} \]

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.
Proving exponential runtime

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

We proved that \( f(n) \) is \( O(2^n) \)

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

\( f(n) \geq c2^n \), for some \( c \)

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

Proving correctness

```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
```

Can you prove that these two functions give the same result, i.e. that \( \text{fibrec}(n) = \text{fibiter}(n) \)?

Prove it!

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

1. State what you’re trying to prove!
2. State and prove the base case(s)
3. Assume it’s true for all values \( \leq k \)
4. Show that it holds for \( k+1 \)

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

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

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

$n = 1$: 1

$n = 0$: 1

$n = 1$: 1
Inductive hypotheses  
\[ \text{fibrec}(n) = \text{fibiter}(n) \]

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 prev 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 = fibiter(n-2)  
by inductive hypothesis: 
prev1 = fibrec(n-2)
Assume: \( \text{fibiter}(n-2) = \text{fibrec}(n-2) \)
\( \text{fibiter}(n-1) = \text{fibrec}(n-1) \)

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

What is prev2 after this?
prev2 = \( \text{fibrec}(n-2) \)

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

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

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

What is prev1 after this?
prev1 = \( \text{fibrec}(n-1) \)

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

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

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

What is prev1 after this?
prev1 = \( \text{fibrec}(n-2) \)

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

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

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

Done!