BIG-O

David Kauchak
CS52 – Spring 2016

Admin

Assignment 4 graded

Assignment 5

Academic honesty

member

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

What is it's 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

member

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

How fast is it?

For a list with k elements in it, how many calls are made to member?

Depends on the input!

member

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

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

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

member

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

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

Uniquify

```
fun uniquify0 []      = []
  | uniquify0 (x::xs) =
    if member x xs
      then uniquify0 xs
      else x::(uniquify0 xs);

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

Type signature?

What do they do?

Which is faster?

How much faster?

uniquify0

```
fun uniquify0 []      = []
  | uniquify0 (x::xs) =
    if member x xs
      then uniquify0 xs
      else x::(uniquify0 xs);
```

How many calls to member are made for a list of size k, including calls made in uniquify0 *as well as* recursive calls made in member?

Depends on the values!

uniquify0

```
fun uniquify0 []      = []
  | uniquify0 (x::xs) =
    if member x xs
      then uniquify0 xs
      else x::(uniquify0 xs);
```

Worst case, how many calls to member are made for a list of size k, including calls made in uniquify0 as well as recursive calls made in member?

uniquify0

```
fun uniquify0 []      = []
  | uniquify0 (x::xs) =
    if member x xs
      then uniquify0 xs
      else x::(uniquify0 xs);
```

How many calls are made if the list is empty?

0

uniquify0

```
fun uniquify0 []      = []
  | uniquify0 (x::xs) =
    if member x xs
      then uniquify0 xs
      else x::(uniquify0 xs);
```

Recursive case:
Let $count_0(i)$ 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 ($count_0(k)$)? Hint: the definition will be recursive?

uniquify0

```
fun uniquify0 []        = []
  | uniquify0 (x::xs) =
    if member x xs
      then uniquify0 xs
      else x::(uniquify0 xs);
```

Recursive case:
Let $count_0(i)$ be the number of calls that uniquify0 makes to member for a list of size i.

$$count_0(k) = (k+1) + count_0(k-1)$$

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

$$count_0(k) = \begin{cases} 0 & \text{if, } k = 0 \\ k + count_0(k-1) & \text{otherwise} \end{cases}$$

How many calls is this?

Recurrence relation

$$count_0(k) = \begin{cases} 0 & \text{if, } k = 0 \\ k + count_0(k-1) & \text{otherwise} \end{cases}$$

$$count_0(k) = k + count(k-1)$$

$$= k + k - 1 + count_0(k-2)$$

$$= k + k - 1 + k - 2 + count_0(k-3)$$

$$= k + k - 1 + k - 2 + \ldots + 1 + count_0(0)$$

$$= k + k - 1 + k - 2 + \ldots + 1 + 0$$

Recurrence relation

$$count_0(k) = \begin{cases} 0 & \text{if, } k = 0 \\ k + count_0(k-1) & \text{otherwise} \end{cases}$$

$$count_0(k) = \frac{k(k+1)}{2} \approx \frac{k^2}{2}$$ 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!

$$count_0(k) = \begin{cases} 0 & \text{if, } k = 0 \\ k + count_0(k-1) & \text{otherwise} \end{cases}$$

1. $count_0(k) = \frac{k(k+1)}{2}$

2. base case?

Proof by induction!

$$count_0(k) = \begin{cases} 0 & \text{if, } k = 0 \\ k + count_0(k-1) & \text{otherwise} \end{cases}$$

1. $count_0(k) = \frac{k(k+1)}{2}$

2. k = 0

$count_0(k) = 0$ from definition of $count_0$

$count_0(k) = \frac{0(0+1)}{2} = 0$ by math

1. $count_0(k) = \frac{k(k+1)}{2}$

$$count_0(k) = \begin{cases} 0 & \text{if, } k = 0 \\ k + count_0(k-1) & \text{otherwise} \end{cases}$$

3. assume: $count_0(k-1) =$ inductive hypothesis

1. $count_0(k) = \frac{k(k+1)}{2}$

$$count_0(k) = \begin{cases} 0 & \text{if, } k = 0 \\ k + count_0(k-1) & \text{otherwise} \end{cases}$$

3. assume: $count_0(k-1) = \frac{(k-1)k}{2}$ inductive hypothesis

1. $count_0(k) = \frac{k(k+1)}{2}$

$$count_0(k) = \begin{cases} 0 & \text{if, } k = 0 \\ k + count_0(k-1) & \text{otherwise} \end{cases}$$

3. assume: $count_0(k-1) = \frac{(k-1)k}{2}$ inductive hypothesis

4. prove: $count_0(k) = \frac{k(k+1)}{2}$

$$count_0(k) = k + count_0(k-1)$$ by definition of $count_0$

$$= k + \frac{(k-1)k}{2}$$ inductive hypothesis

$$= \frac{2k + k^2 - k}{2}$$ math (k = 2k/2, multiply (k-1)k)

Proof by induction!

$$count_0(k) = \begin{cases} 0 & \text{if, } k = 0 \\ k + count_0(k-1) & \text{otherwise} \end{cases}$$

3. assume: $count_0(k-1) = \frac{(k-1)k}{2}$ inductive hypothesis

4. prove: $count_0(k) = \frac{k(k+1)}{2}$

$$= \frac{2k + k^2 - k}{2}$$

$$= \frac{k^2 + k}{2}$$ more math (subtraction)

$$= \frac{k(k+1)}{2}$$ more math (factor out k)

Done!

uniquify1

```
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 $count_1$ that gives the number of calls to member for a list of size k.

$$count_1(k) = \begin{cases} & \text{if, } k = 0 \\ & \text{otherwise} \end{cases}$$

uniquify1

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

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

How many calls is that?

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

I claim: $count_1(k) = 2^{k+1} - k - 2$

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)

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

1. $count_1(k) = 2^{k+1} - k - 2$

Proof by induction!

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

1. $count_1(k) = 2^{k+1} - k - 2$
2. Base case:

Proof by induction!

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

1. $count_1(k) = 2^{k+1} - k - 2$

2. Base case: k = 0

$count_1(k) = 0$ from definition of $count_1$

$count_1(k) =$

Proof by induction!

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

1. $count_1(k) = 2^{k+1} - k - 2$

2. Base case: k = 0

$count_1(k) = 0$ from definition of $count_1$

$count_1(k) = 2^1 - 0 - 2 = 0$ by math

Proof by induction!

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

1. $count_1(k) = 2^{k+1} - k - 2$

---

3. assume: $count_1(k-1) = 2^k - (k-1) - 2$

inductive hypothesis

$= 2^k - k - 1$

Proof by induction!

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

3. assume: $count_1(k-1) = 2^k - k - 1$ inductive hypothesis

4. prove: $count_1(k) = 2^{k+1} - k - 2$

$count_1(k) = k + 2count_1(k-1)$ by definition of $count_1$

$= k + 2(2^k - k - 1)$ inductive hypothesis

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

$= 2^{k+1} - k - 2$ Done! math

Does it matter?

```
fun uniquify0 []       = []
  | uniquify0 (x::xs) =
    if member x xs
      then uniquify0 xs
      else x::(uniquify0 xs);
```

$$count_0(k) = \frac{k(k+1)}{2}$$

vs.

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

$$count_1(k) = 2^{k+1} - k - 2$$

Does it matter?

$$count_0(k) = \frac{k(k+1)}{2} \qquad count_1(k) = 2^{k+1} - k - 2$$

k	0	1	2	3	4	… 10 …	100
$count_0(k)$	?						
$count_1(k)$							

Does it matter?

$$count_0(k) = \frac{k(k+1)}{2} \qquad count_1(k) = 2^{k+1} - k - 2$$

k	0	1	2	3	4	… 10 …	100
$count_0(k)$	0	?					
$count_1(k)$	0						

Does it matter?

$$count_0(k) = \frac{k(k+1)}{2} \qquad count_1(k) = 2^{k+1} - k - 2$$

k	0	1	2	3	4	… 10 …	100
$count_0(k)$	0	1	?				
$count_1(k)$	0	1					

Does it matter?

$count_0(k) = \frac{k(k+1)}{2}$ $count_1(k) = 2^{k+1} - k - 2$

k	0	1	2	3	4	…	10	…	100
$count_0(k)$	0	1	3	?					
$count_1(k)$	0	1	4						

Does it matter?

$count_0(k) = \frac{k(k+1)}{2}$ $count_1(k) = 2^{k+1} - k - 2$

k	0	1	2	3	4	…	10	…	100
$count_0(k)$	0	1	3	6	15	…	?		
$count_1(k)$	0	1	4	11	57	…			

Does it matter?

$count_0(k) = \frac{k(k+1)}{2}$ $count_1(k) = 2^{k+1} - k - 2$

k	0	1	2	3	4	…	10	…	100
$count_0(k)$	0	1	3	6	15	…	55	…	?
$count_1(k)$	0	1	4	11	57	…	2036	…	

Does it matter?

$count_0(k) = \frac{k(k+1)}{2}$ $count_1(k) = 2^{k+1} - k - 2$

k	0	1	2	3	4	…	10	…	100
$count_0(k)$	0	1	3	6	15	…	55	…	5050
$count_1(k)$	0	1	4	11	57	…	2036	…	2.5×10^{30}

Maybe it's not that bad

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

Roughly how long will that take?

Maybe it's not that bad

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

- Assume 10^9 calls per second
- ~3×10^7 seconds per year
- ~3×10^{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

Undo

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

What's the problem?
Can we fix it?

Undo

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

fun uniquify2 []      = []
  | uniquify2 (x::xs) =
    let
        val recResult = uniquify2 xs;
    in
        if member x recResult
          then recResult
          else x::recResult
    end;
```

Which is faster?

```
fun uniquify0 []      = []
  | uniquify0 (x::xs) =
    if member x xs
      then uniquify0 xs
      else x::(uniquify0 xs);

fun uniquify2 []      = []
  | uniquify2 (x::xs) =
    let
        val recResult = uniquify2 xs;
    in
        if member x recResult
          then recResult
          else x::recResult
    end;
```

Big O: Upper bound

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

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Big O: Upper bound

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

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

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

Big O: Upper bound

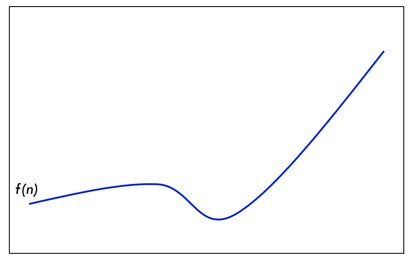
$O(g(n))$ is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

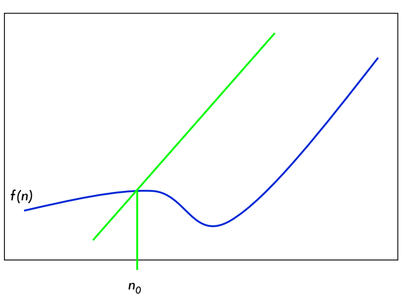
We can bound the function $f(n)$ above by some constant factor of $g(n)$: constant factors don't matter!

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)

uniquify0 is O(n²) – quadratic
- n(n+1)/2 = n²/2 + n/2 is O(n²)

uniquify1 is O(2ⁿ) – exponential
- 2ⁿ⁺¹-n-2 is O(2ⁿ)

uniquify2 is O(n²) – quadratic

Runtime examples

	n	$n \log n$	n^2	n^3	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 18 min	10^{25} years
$n = 100$	< 1 sec	< 1 sec	1 sec	1s	10^{17} years	very long
$n = 1000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long

(adapted from [2], 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

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?

```
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

```
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
```

O(n) – linear

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?

Runtime

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

Guess: O(2^n) – for each call, makes two recursive calls

What is the recurrence relation?

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

Runtime

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

Guess: O(2^n) – for each call, makes two recursive calls

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

Slightly different than the recurrence relation for uniquify1.

Proof

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

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

Show that $f(n) \le 2^n - 1$

Why is this sufficient?

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Proof

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

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

Show that $f(n) \le 2^n - 1$

$f(n) \le 2^n - 1 \le 2^n$ ($c = 1$, for all $n \ge 0$)

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Proof

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

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

Show that $f(n) \le 2^n - 1$

How do we prove this? Induction!

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Proof by induction

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

1. Prove: $f(n) \le 2^n - 1$
2. Base case:

$n = 1$

$f(1) = 1$ by definition of f(n)

$f(1) = 2^1 - 1 = 1$ by math

Proof by induction

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

1. Prove: f(n) ≤ 2^n-1

3. Inductive hypothesis:

Assume: $f(n) \le 2^n - 1$

4. Prove:

n+1: $f(n+1) \le 2^{n+1} - 1$

Proof by induction

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

Assume: $f(n) \le 2^n - 1$ Prove: $f(n+1) \le 2^{n+1} - 1$

definition of f(n)

$f(n+1) = 1 + f(n-1) + f(n) \le f(n-1) + 2^n - 1$ inductive hypothesis

What do we do with ?

Proof by induction

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

1. Prove: f(n) ≤ 2^n-1

3. Inductive hypothesis:

Assume: $f(n) \le 2^n - 1$

$\boxed{f(n-1) \le 2^{n-1} - 1}$ strong induction

4. Prove:

n+1: $f(n+1) \le 2^{n+1} - 1$

Proof by induction

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

Assume: $f(n) \le 2^n - 1$ Prove: $f(n+1) \le 2^{n+1} - 1$

$f(n-1) \le 2^{n-1} - 1$

definition of f(n)

$f(n+1) = 1 + f(n-1) + f(n) \le 2^{n-1} - 1 + 2^n - 1$ inductive hypotheses

$\le 2^{n-1} + 2^n - 2$ math

$\le 2^n + 2^n - 2$ $2^{n-1} < 2^n$

$\le 2 \cdot 2^n - 2$ more math

$\le 2^{n+1} - 2 \le 2^{n+1} - 1$ Done!

Proving exponential runtime

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

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

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We proved that *f(n)* is $O(2^n)$

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

$f(n) \ge c2^n$, for some c

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

Proving correctness

```
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 fibrec(n) = fibiter(n)?

Prove it! fibrec(n) = 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 ≤ k
4. Show that it holds for k+1

```
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
```

Base cases

fibrec(n) = fibiter(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

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

n = 0 and n = 1

?

Base cases

fibrec(n) = fibiter(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

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

n = 0 and n = 1

?

n = 0: 1

n = 1: 1

Base cases

fibrec(n) = fibiter(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

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

n = 0 and n = 1

Loop doesn't execute at all

prev1 = 1 and is returned

n = 0: 1

n = 0: 1

n = 1: 1

Base cases

fibrec(n) = fibiter(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

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

n = 0 and n = 1

Loop executes once

prev1 = 1 + 0 = 1

n = 1: 1

n = 0: 1

n = 1: 1

Inductive hypotheses

fibrec(n) = fibiter(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

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

Assume:

fibrec(n-1) = fibiter(n-1)

fibrec(n-2) = fibiter(n-2)

Prove:

fibrec(n) = fibiter(n)

Assume: fibiter(n-2) = fibrec(n-2)

fibiter(n-1) = fibrec(n-1)

Prove: fibiter(n) = fibrec(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n):
        prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

Definition of for loops

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n - 2):
        prev2, prev1 = prev1, (prev1 + prev2)

    # iteration n-1
    prev2, prev1 = prev1, (prev1 + prev2)

    # iteration n
    prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

Assume: fibiter(n-2) = fibrec(n-2)

fibiter(n-1) = fibrec(n-1)

Prove: fibiter(n) = fibrec(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n - 2):
        prev2, prev1 = prev1, (prev1 + prev2)

    # iteration n-1
    prev2, prev1 = prev1, (prev1 + prev2)

    # iteration n
    prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

What is prev1 after this?

Assume: fibiter(n-2) = fibrec(n-2)

fibiter(n-1) = fibrec(n-1)

Prove: fibiter(n) = fibrec(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n - 2):
        prev2, prev1 = prev1, (prev1 + prev2)

    # iteration n-1
    prev2, prev1 = prev1, (prev1 + prev2)

    # iteration n
    prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

prev1 = fibiter(n-2)

by inductive hypothesis:

prev1 = fibrec(n-2)

Assume: fibiter(n-2) = fibrec(n-2)

fibiter(n-1) = fibrec(n-1)

Prove: fibiter(n) = fibrec(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n - 2):
        prev2, prev1 = prev1, (prev1 + prev2)
                                        prev1 = fibrec(n-2)
    # iteration n-1
    prev2, prev1 = prev1, (prev1 + prev2)

    # iteration n
    prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

What is prev2 after this?

assignment

prev2 = fibrec(n-2)

Assume: fibiter(n-2) = fibrec(n-2)

fibiter(n-1) = fibrec(n-1)

Prove: fibiter(n) = fibrec(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n - 2):
        prev2, prev1 = prev1, (prev1 + prev2)
                                        prev1 = fibrec(n-2)
    # iteration n-1
    prev2, prev1 = prev1, (prev1 + prev2)
                                        prev2 = fibrec(n-2)
    # iteration n
    prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

What is prev1 after this?

by inductive hypothesis

prev1 = fibrec(n-1)

Assume: fibiter(n-2) = fibrec(n-2)

fibiter(n-1) = fibrec(n-1)

Prove: fibiter(n) = fibrec(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n - 2):
        prev2, prev1 = prev1, (prev1 + prev2)
                                        prev1 = fibrec(n-2)
    # iteration n-1
    prev2, prev1 = prev1, (prev1 + prev2)
                                        prev2 = fibrec(n-2)
    # iteration n                       prev1 = fibrec(n-1)
    prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

What is prev1 after this?

Assume: fibiter(n-2) = fibrec(n-2)

fibiter(n-1) = fibrec(n-1)

Prove: fibiter(n) = fibrec(n)

```
def fibiter(n):
    prev2, prev1 = 0, 1

    for i in range(n - 2):
        prev2, prev1 = prev1, (prev1 + prev2)
                                        prev1 = fibrec(n-2)
    # iteration n-1
    prev2, prev1 = prev1, (prev1 + prev2)
                                        prev2 = fibrec(n-2)
    # iteration n                       prev1 = fibrec(n-1)
    prev2, prev1 = prev1, (prev1 + prev2)

    return prev1
```

Done!

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