

# CPS 305

**Data Structures**

**Prof. Alex Ufkes**

**Topic 2:** Continuing Lisp, complexity analysis

# Notice!

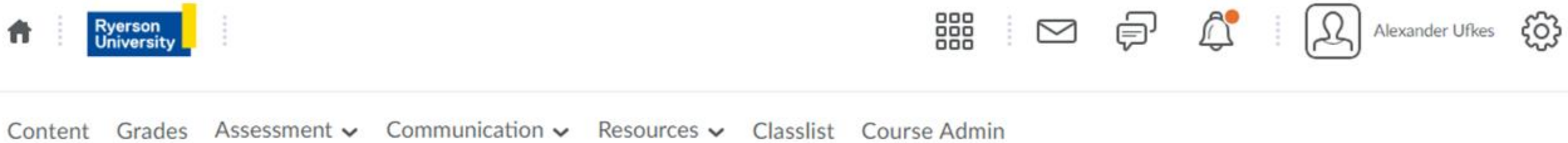
---

## **Obligatory copyright notice in the age of digital delivery and online classrooms:**

*The copyright to this original work is held by Alex Ufkes. Students registered in course CPS 305 can use this material for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.*

# Course Administration

---



- Labs start this week.
- Before the lab, install SBCL, Vscode, etc.

# Today

---

- More advanced Lisp
- Complexity/asymptotic analysis
- Big-O notation



# Common Lisp: Resources

---

**Common Lisp Cookbook:**

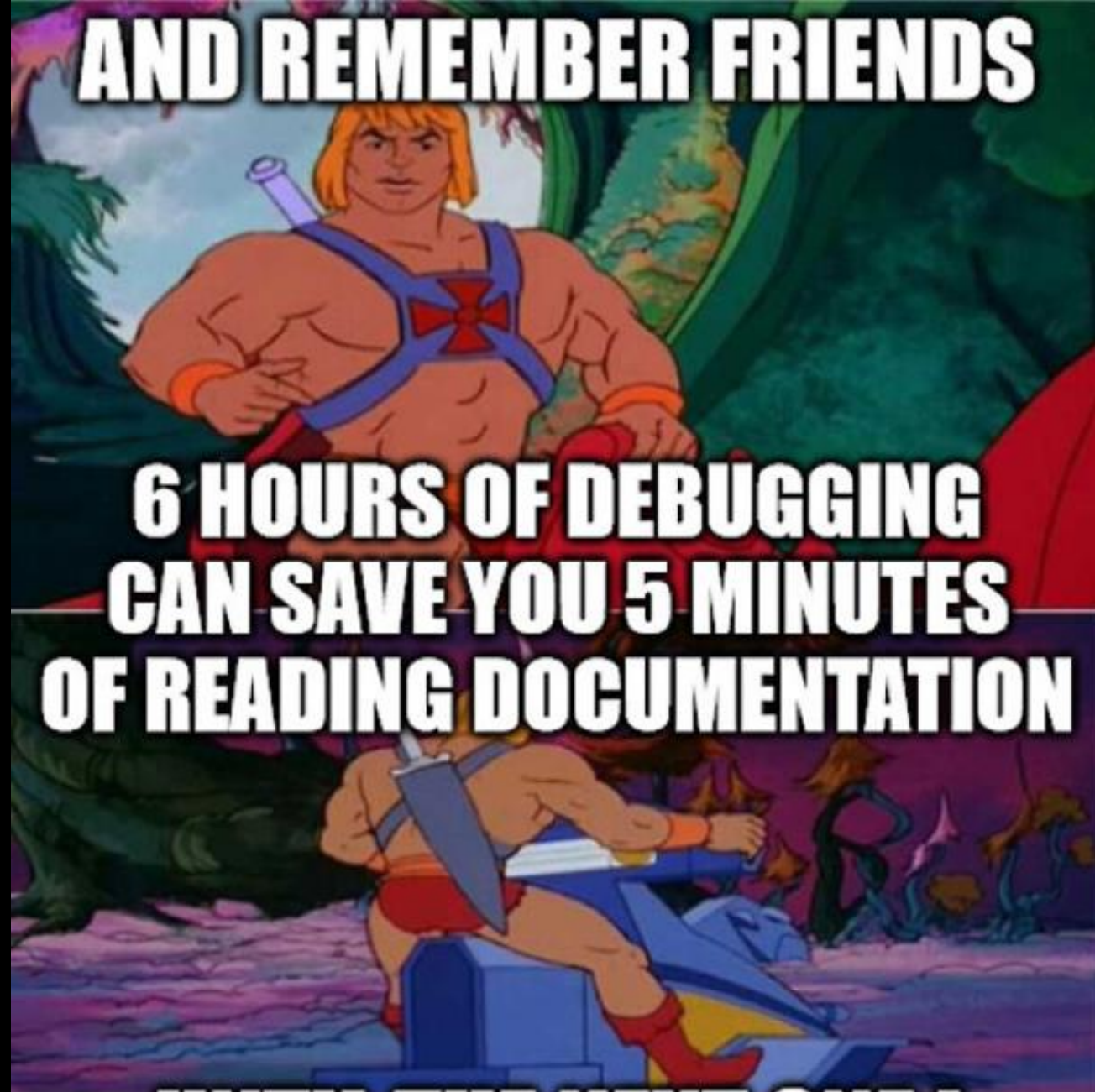
<https://lispcookbook.github.io/cl-cookbook/>

**CL HyperSpec (complete documentation):**

<http://clhs.lisp.se/Front/index.htm>

**Lisp Tutorialspoint**

<https://www.tutorialspoint.com/lisp/index.htm>





# **Last class:**

## **Forms & basic syntax**



# Recall: Lisp Forms

---

**Let's play a game:** We think of a number, and Lisp tries to guess it!

1. Define lower and upper bounds
2. Choose a number between these bounds
3. If the player says smaller, lower the upper bound
4. If the player says larger, raise the lower bound

The game will be played in the REPL, we'll use global variables for the bounds.

# Recall: Lisp Forms

---

```
* (defvar *lowerb* 1)
*LOWERB*
* (defvar *upperb* 100)
*UPPERB*
* *lowerb*
1
* *upperb*
100
```

- Asterisks, or “earmuffs”, are part of the variable name.
- This is convention for global variables in Lisp.

# Recall: Lisp Forms

---

- Lisp will use binary search to guess the number.
- Pick halfway point, raise or lower bounds accordingly.
- Use the **ash** function to efficiently divide/multiply by 2:

```
* (ash 5 -1)
```

```
2
```

```
* (ash 6 -1)
```

```
3
```

```
* (ash 6 1)
```

```
12
```

**Note:** *Bit-shifting is MUCH faster than division when the 2<sup>nd</sup> operand is a power of 2.*

# Recall: Lisp Forms

---

Use **setf** to update bounds:

```
* (defvar *lowerb* 1)
* LOWERB*
* (defvar *upperb* 100)
* UPPERB*
* *lowerb*
1
* *upperb*
100
```

```
* (setf *lowerb* 25)
25
* (setf *upperb* 50)
50
* *lowerb*
25
* *upperb*
50
```

# Recall: Lisp Forms

---

```
(defun guess-my-number ()  
  (ash (+ *upperb* *lowerb*) -1))
```

```
(defun smaller ()  
  (setf *upperb* 1- (guess-my-number)))  
  (guess-my-number))
```

```
(defun bigger ()  
  (setf *lowerb* 1+ (guess-my-number)))  
  (guess-my-number))
```

Functions for increment/decrement

λ guess-number.lisp &gt; ...

```
1 (defvar *lowerb* 1)
2 (defvar *upperb* 100)
3
4 (defun guess ()
5   "Returns the integer mean of *upperb* and *lowerb*"
6   (ash (+ *upperb* *lowerb*) -1))
7
8 (defun smaller ()
9   (setf *upperb* (1- (guess)))
10  (guess))
11
12 (defun bigger ()
13   (setf *lowerb* (1+ (guess)))
14   (guess))
15
16 (defun start-over ()
17   (setf *lowerb* 1)
18   (setf *upperb* 100)
19   (guess))
20
21
```

Pick a number in your head:

- Let's say... 42!

```
* (guess)
50
* (smaller)
25
* (bigger)
37
* (bigger)
43
* (smaller)
40
* (bigger)
41
* (bigger)
42
* (start-over)
50
*
```

# Branching Variations

---

`(if testForm thenForm [elseForm])`

`* (if nil 'hello 'world)`

WORLD

`* (if (< 2 3) "2 smaller" "3 smaller")`

"2 smaller"

`* (when (> 2 3) (print "no way"))`

NIL

`* (unless (< 4 2) 10)`

10

# Continuing on...



# Special Forms: Cond

---

- We don't have **elif** ladders in Lisp the way we do in Python.
- Testing a long chain of conditions can get messy:

```
(defun whereis (city)
  (if (eq city 'toronto) 'canada
      (if (eq city 'beijing) 'china
          (if (eq city 'moscow) 'russia
              (if (eq city 'miami) 'usa
                  'unknown)))))
```

```
* (whereis 'toronto)
CANADA
* (whereis 'beijing)
CHINA
* (whereis 'moscow)
RUSSIA
* (whereis 'miami)
USA
* (whereis 'london)
UNKNOWN
```

# Special Forms: Cond

---

Use **cond** instead:

```
(defun whereis (city)
  (if (eq city 'toronto) 'canada
      (if (eq city 'beijing) 'china
          (if (eq city 'moscow) 'russia
              (if (eq city 'miami) 'usa
                  'unknown)))))
```

```
(defun whereis-cond (city)
  (cond ((eq city 'toronto) 'canada)
        ((eq city 'beijing) 'china)
        ((eq city 'moscow) 'russia)
        ((eq city 'miami) 'usa)
        (t 'unknown)))
```

Easier to stack conditions, parentheses don't pile up on us

# Tangent: Testing Equality

---

`(eq city 'toronto) (= 5 7) (equal "hello" "hello")`

**EQ**

Compares identity. Same pointer value.

**EQUAL**

Compares value, works on lists/strings/arrays

- Recursively tests values of elements

**EQUALP**

Works on structs and hash tables

- Beware! NOT case sensitive on strings!

`* (equalp "hello" "HELLO")`

`T`

# Tangent: Testing Equality

---

`(eq city 'toronto) (= 5 7) (equal "hello" "hello")`

**EQ**

Compares identity. Same pointer value.

**EQUAL**

Compares value, works on lists/strings/arrays

- Recursively tests values of elements

**EQUALP**

Works on structs and hash tables

- Beware! NOT case sensitive on strings!

**=**

Use for numeric values only.

**There are others. A good article that goes more in depth:**

<https://anticrisis.github.io/2017/09/08/equality-in-common-lisp.html>

# Repetition: Looping & Recursion

---



# Simple Recursion

---

- If we can branch, we can separate base case from recursive case.
- If we can call a function, we can make a recursive call.

## In Python?

```
def fact(n):  
    if n < 2:  
        return 1  
    else:  
        return n*fact(n-1)
```

## In Lisp?

```
(defun fact (x)  
  (if (< x 2) 1  
      (* x (fact (- x 1)))))
```

```
(if testForm thenForm [elseForm])
```

# Simple Recursion: trace

---

```
(defun fact (x)
  (if (< x 2) 1
      (* x (fact (- x 1)))))
```

```
* (trace fact)
(FACT)
* (fact 4)
  0: (FACT 4)
    1: (FACT 3)
      2: (FACT 2)
        3: (FACT 1)
          3: FACT returned 1
        2: FACT returned 2
      1: FACT returned 6
    0: FACT returned 24
24
```

# Simple Repetition: `dotimes`

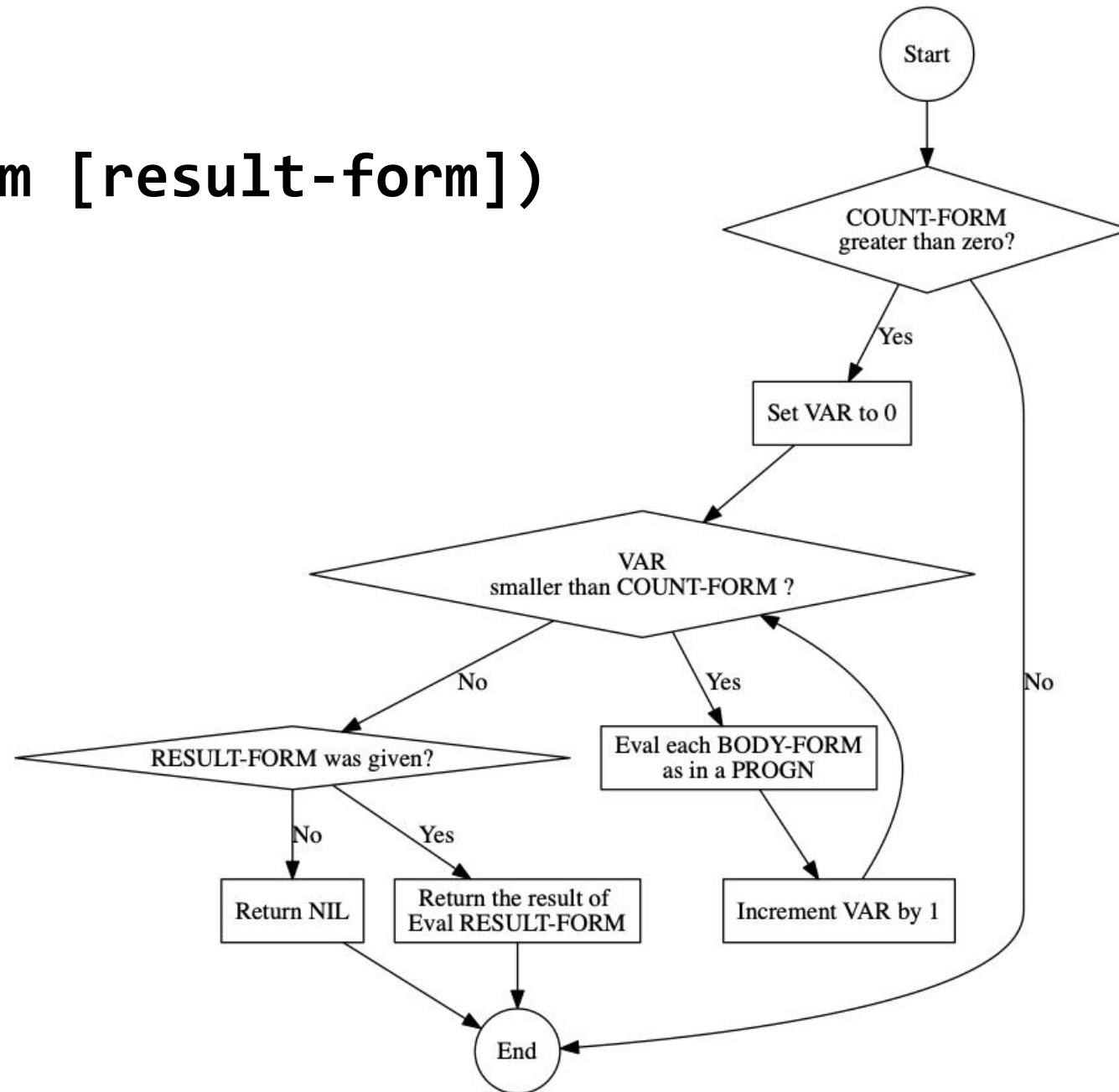
---

`(DOTIMES (var count-form [result-form]) body-form*)`

- Returns value of the optional RESULT-FORM, otherwise returns NIL
- `body-form*` can be any number of forms, corresponding to the loop body.



(DOTIMES  
  (var count-form [result-form])  
  body-form\*)




# Simple Repetition: dotimes

---

**(DOTIMES (var count-form [result-form]) body-form\*)**

- Returns value of the optional RESULT-FORM
- otherwise returns NIL

```
(dotimes (n 6)
  (print n) (prin1 (* n n))
)
```



0	0
1	1
2	4
3	9
4	16
5	25

```
(print  
  (dotimes (n 6)  
    (print n) (prin1 (* n n))  
  )  
)
```

```
0 0  
1 1  
2 4  
3 9  
4 16  
5 25  
NIL
```

```
(print  
  (dotimes (n 6 "done")  
    (print n) (prin1 (* n n))  
  )  
)
```

```
0 0  
1 1  
2 4  
3 9  
4 16  
5 25  
"done"
```

# Simple Repetition: dotimes

---

## Factorial:

```
(defun factorial (n)
  (let ((fact 1))
    (dotimes (i n fact)
      (setf fact (* fact (+ i 1)))
    )
  )
)
```

- **fact** set to 1 for use in **dotimes** form
- **i** starts at 0, increments up to n-1
- Update fact using **setf**: multiply by i+1
- **dotimes** returns **fact**

```
λ dotimes.lisp  factorial.lisp ×
λ factorial.lisp > ...
1  (defun factorial (n)
2    (let ((fact 1))
3      (dotimes (i n fact)
4        (setf fact (* fact (+ i 1))))
5      )
6    )
7  )
8
9  (print (factorial 1))
10 (print (factorial 2))
11 (print (factorial 3))
12 (print (factorial 4))
13 (print (factorial 5))
14 (print (factorial 6))
15 (print (factorial 7))
16 (print (factorial 8))
17
```

```
sbcl --script factorial.lisp
1
2
6
24
120
720
5040
40320
```

# Looping with RETURN

---

When a loop hits a return form that has no argument, it exits and returns nil

```
(dotimes (i 6 "done")  
  (if (= i 3)  
    (return) ; RETURN without an argument forces the  
    (print i) ; enclosing loop to return NIL  
  )  
)
```

```
* (dotimes (i 6 "done") (if (= i 3) (return) (print i)))  
0  
1  
2  
NIL  
* |
```

```
Windows PowerShell
SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.
* (dotimes (i 6 "done") (if (= i 3) (return) (print i)))

0
1
2
NIL
* (dotimes (i 6 "done") (if (= i 3) (return "exited") (print i)))

0
1
2
"exited"
* |
```

Return a value other than nil

# More Examples: String Capitalizer

---

- Use **DOTIMES** to iterate over the characters of a string
- We'll use a couple of new functions to help us:
- **CHAR-CODE** returns the ASCII code of a character (**CODE-CHAR** does the inverse)
- **CHAR>=** and **CHAR<=** are for comparing characters
- the accessor (**AREF str i**) returns the character at index **i** in the string **str**.
- For example: (**AREF "fdsa" 1**) => **#\d**



# More Examples: String Capitalizer

---

- **CHAR-CODE** returns the ASCII code of a character
- **CODE-CHAR** does the inverse
- **CHAR>=** and **CHAR<=** are for comparing characters
- **(AREF str i)** returns the character at index **i** in the string **str**.

```
(defun cpt-char (c)
  (if (and (char>= c #\a) (char<= c #\z))
      (code-char (- (char-code c) 32))
      c)
)

(defun capitalize (s)
  (dotimes (i (length s) s)
    (setf (aref s i) (cpt-char (aref s i))))
)
)
```

```
* (capitalize "Hello, world!")
"HELLO, WORLD!"
```

# More Examples: Vowel Capitalizer

---

```
(defun is-vowel (c)
  (or (char= c #\a)
      (char= c #\e)
      (char= c #\i)
      (char= c #\o)
      (char= c #\u))
)
```

```
(defun cap-vowels (s)
  (dotimes (i (length s) s)
    (if (is-vowel (aref s i))
        (setf (aref s i) (cpt-char (aref s i)))
    )
  )
)
```

```
(defun cpt-char (c)
  (if (and (char>= c #\a) (char<= c #\z))
      (code-char (- (char-code c) 32))
      c)
)
```

# Simple Repetition: do

---

```
(DO (var-definition*)  
    (end-test result-form*)  
    body-form*)
```

## Recall:

- `form*` means zero or more forms
- `[form]` means optional form

# Simple Repetition: do and do\*

---

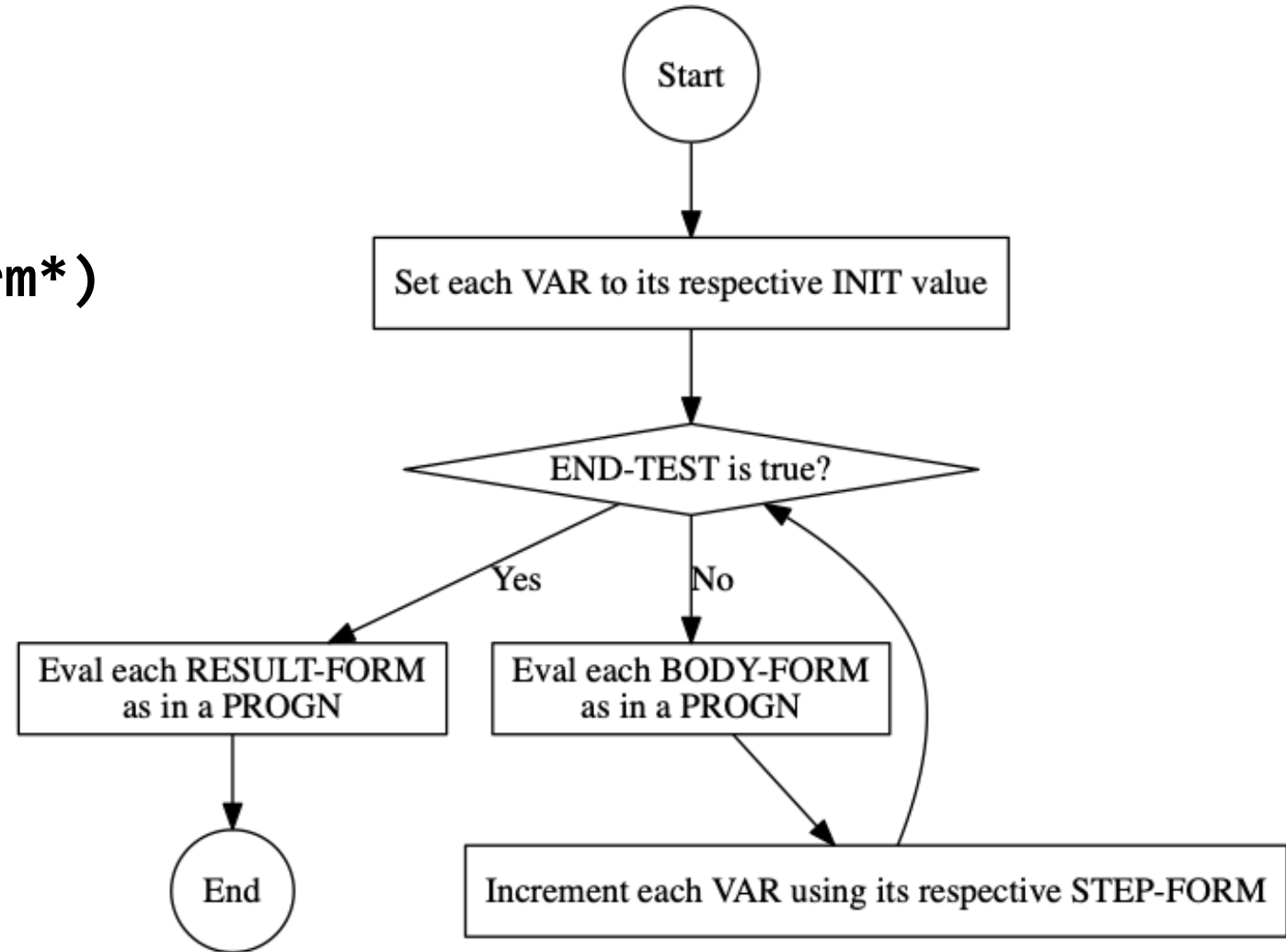
`(DO (var-definition*)  
 (end-test result-form*)  
 body-form*)`

`(DO* (var-definition*)  
 (end-test result-form*)  
 body-form*)`

DO vs DO\* (Same as LET vs LET\*)

- `(DO (var-definition*)` Vars assigned in parallel
- `(DO* (var-definition*)` Vars assigned in series

**(DO (var-definition\*)  
(end-test result-form\*)  
body-form\*)**



# DO Example: Count Occurrences

---

Count the number of occurrences of character **c** in string **s**

```
; Using dotimes
(defun count-chars (c s)
  (let ((acc 0))
    (dotimes (i (length s) acc)
      (when (char= (aref s i) c)
        (incf acc) ; equivalent to (setf acc (+ 1 acc))
      )
    )
  )
)
```

What about using **do**?

# DO Example: Count Occurrences

---

Count the number of occurrences of character **c** in string **s**

```
(DO (var-definition*)  
  (end-test result-form*)  
  body-form*)  
)
```

```
(defun count-char (c s)  
  (do ((i 0 (1+ i)) (acc 0))  
    ((= i (length s)) acc)  
    (when (char= (aref s i) c)  
      (incf acc)))  
  ))
```

λ do.lisp ×

λ do.lisp > ...

```
1 (defun count-chars-dotimes (c s)
2   (let ((acc 0))
3     (dotimes (i (length s) acc)
4       (when (char= (aref s i) c)
5         (incf acc) ; equivalent to (setf acc (+ 1 acc))
6       ) ) ) )
7
8 (defun count-chars-do (c s)
9   (do ((i 0 (1+ i)) (acc 0))
10     ((= i (length s)) acc)
11     (when (char= (aref s i) c)
12       (incf acc))
13   ))
14
15
```

```
* (count-chars-dotimes #\a "abba")
2
* (count-chars-dotimes #\b "abba")
2
* (count-chars-dotimes #\c "abba")
0
* (count-chars-do #\a "abba")
2
* (count-chars-do #\b "abba")
2
* (count-chars-do #\c "abba")
0
*
```



# DO Example: Count Vowels & Consonants

---

Update is-vowel to include uppercase:

```
(defun is-vowel (c)
  (or (char= c #\a) (char= c #\A)
      (char= c #\e) (char= c #\E)
      (char= c #\i) (char= c #\I)
      (char= c #\o) (char= c #\O)
      (char= c #\u) (char= c #\U))
)
```

# DO Example: Count Vowels & Consonants

---

Base `is-cons` on `is-vowel`:

```
(defun is-cons (c)
  (and (not (is-vowel c))
    (or (and (char>= c #\a) (char<= c #\z))
        (and (char>= c #\A) (char<= c #\Z))
    ) ) )
```

# DO Example: Count Vowels & Consonants

- Three variables,
- i has an update

- Exit when i = length of s
- Return two values! (kinda)
- (values accv accc)

```
(defun count-vc (s)
  (do ((i 0 (1+ i)) (accv 0) (accc 0))
      ((equal i (length s)) (values accv accc))
      (if (is-vowel (aref s i)) (incf accv)
          (if (is-cons (aref s i)) (incf accc)))
  ) ) )
```

**Vowel?**  
Increment accv

**Consonant?**  
Increment accc

λ count-vc.lisp ×

λ count-vc.lisp > ...

```
1  (defun is-vowel (c)
2      (or (char= c #\a) (char= c #\A)
3          (char= c #\e) (char= c #\E)
4          (char= c #\i) (char= c #\I)
5          (char= c #\o) (char= c #\O)
6          (char= c #\u) (char= c #\U)
7      )
8  )
9
10 (defun is-cons (c)
11     (and (not (is-vowel c))
12         (or (and (char>= c #\a) (char<= c #\z))
13             (and (char>= c #\A) (char<= c #\Z))
14         )
15     )
16 )
17
18 (defun count-vc (s)
19     (do ((i 0 (1+ i)) (accv 0) (accc 0))
20         ((equal i (length s)) (values accv accc))
21         (if (is-vowel (aref s i)) (incf accv)
22             (if (is-cons (aref s i)) (incf accc))
23         )
24     )
25 )
26
```

This is SBCL 2.3.2, an implementation of ANSI Common Lisp.  
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.  
It is mostly in the public domain; some portions are provided under  
BSD-style licenses. See the CREDITS and COPYING files in the  
distribution for more information.

```
* (count-vc "Hello, world!")
3
7
* (count-vc "How are you doing today?")
9
10
*
```

# LISP: Key Points

---



- Interact with LISP via script or REPL
- LISP source code is comprised of lists (forms)
- **List evaluation:** First element is a function, other elements are arguments to that function
- **Special forms:** These have evaluation rules that differ from other forms (quote, let, defun, etc.)

**We'll learn more LISP as we go, but at this point we're ready to move on**

# Common Lisp: Resources

---

**Common Lisp Cookbook:**

<https://lispcookbook.github.io/cl-cookbook/>

**CL HyperSpec (complete documentation):**

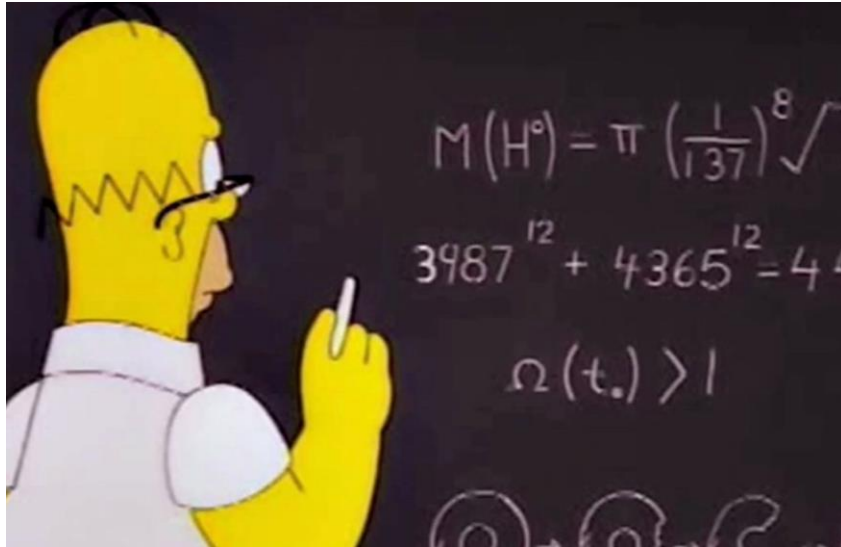
<http://clhs.lisp.se/Front/index.htm>

**Lisp Tutorialspoint**

<https://www.tutorialspoint.com/lisp/index.htm>

# Recall: Quantitative Analysis

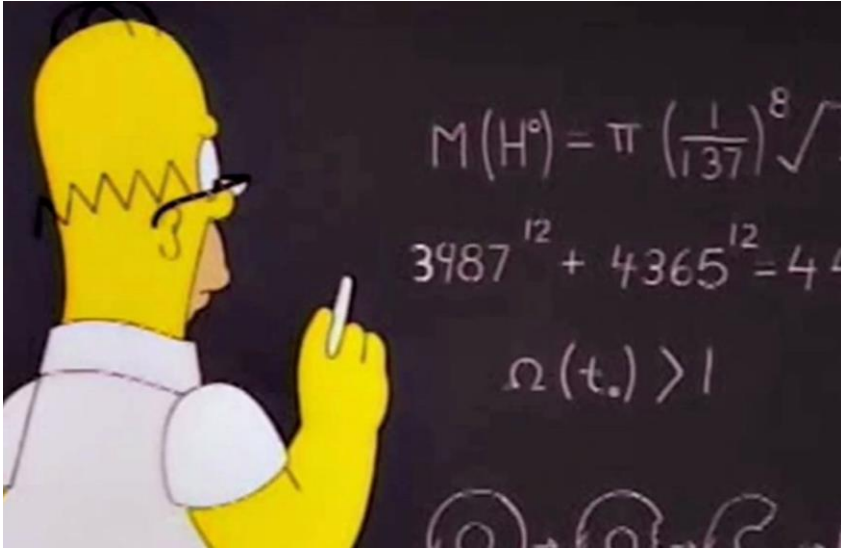
---



- We want to be able to say: Algorithm A is “better” than algorithm B...
- ...and justify it ***quantitatively***.
- This analysis is based on the ***algorithm***, NOT the ***programming language***
- Analysis should ***not*** depend on the platform (Windows, Linux, etc.)

# Recall: Quantitative Analysis

---



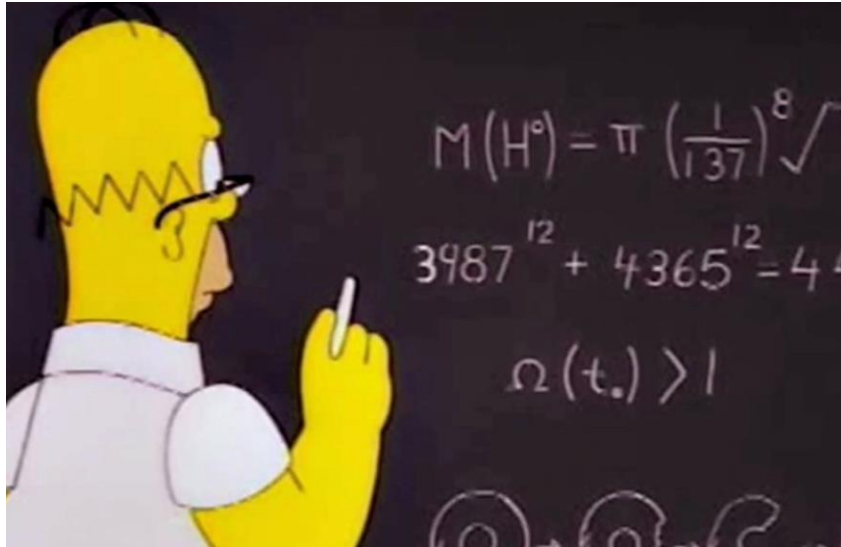
## So... what are we measuring?

- Runtime is the most common metric, but not the only one:
- How much memory is used?
  - (Space complexity)
- How difficult is the implementation?
- How much network traffic is used?
- How big is the source code?
- And so on.



# Recall: Quantitative Analysis

---



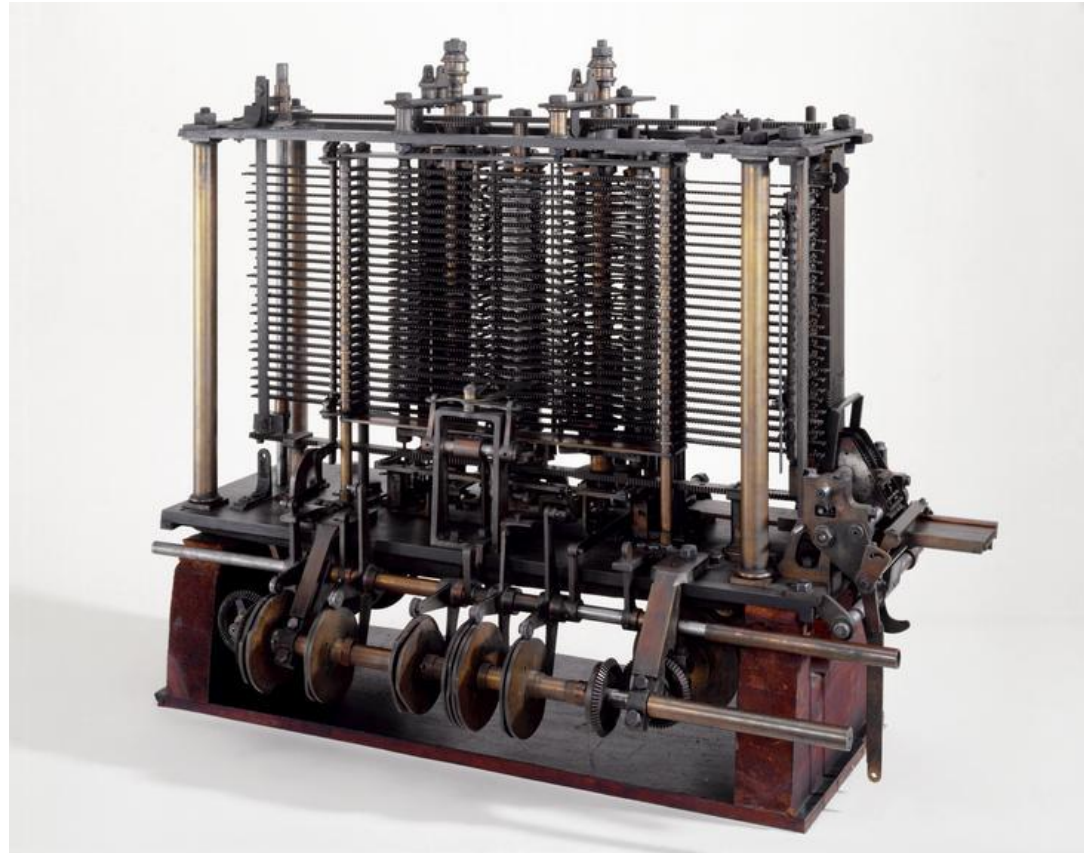
## Runtime is the most common metric:

- Under what conditions?
- What is the size of the problem?
- How does it scale with problem size?
- How do we represent the runtime cost of an algorithm independent of the language and platform?
- ***Time complexity analysis***

# Time Complexity



*“ As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? ” — Charles Babbage (1864)*



# A Tale of Two Algorithms

---

**Task:** Write a function that computes the sum of the first  $n$  integers

**What algorithm should we use for this?**

$$1 + 2 + 3 + 4 + 5 + \dots + (n-2) + (n-1) + n$$

- VS -

$$\frac{n(n+1)}{2}$$

# A Tale of Two Algorithms

---

$$1+2+\dots+(n-1)+n \quad - \text{VS} - \quad \frac{n(n+1)}{2}$$

It's obvious that the 2<sup>nd</sup> one is better... *or is it?*

**What if n = 3?** Two additions VS an addition, multiplication, and division?

**What if n = 5?** Four additions VS an addition, multiplication, and division?

What if n = 10? 20? Arbitrarily large?

What is the relative cost of an integer addition vs an integer multiplication?

At what value of n does option 2 become more efficient?

# Going Deeper?

---

## Rather than counting arithmetic operations...

- We could count machine instructions!
- In practice, we can zoom in as far as CPU cycles.
- Even if every arithmetic operation in our algorithm executes in a single instruction cycle...
- That doesn't mean they execute in the same number of **CPU** cycles.
- Some machine instructions take 2, 3, 4 cycles.  
Depends on the instruction set architecture.
- **HOWEVER!** We want our analysis to be platform-independent, so we won't go this far.

# A Tale of Two Algorithms

---

(DO (var-definition\*) (end-test result-form\*) body-form\*)

```
(defun first-n-lin (n)
  (do ( (i 1 (1+ i))
        (sum 0 (+ i sum)) )
      ( (> i n) sum )
  )
)
```

# A Tale of Two Algorithms

```
λ first-n-sum.lisp ×
λ first-n-sum.lisp > ...
1  (defun first-n-const (n)
2    (/ (* n (+ n 1)) 2)
3  )
4
5  (defun first-n-lin (n)
6    (do ( (i 1 (1+ i))
7        (sum 0 (+ i sum)))
8        ((> i n) sum)
9    )
10 )
11
12 (print (first-n-const 10))
13 (print (first-n-lin 10))
14
15
```

```
sbcl --script first-n-sum.lisp
55
55
```

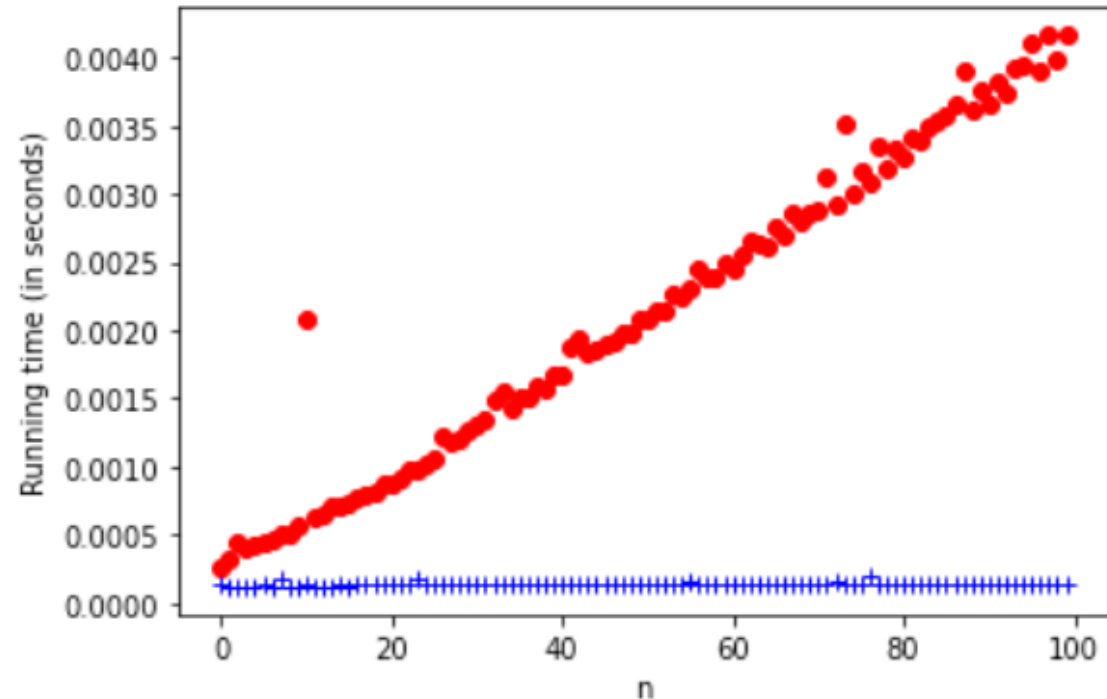
**Which is faster?**

What happens if we execute each these for larger and larger values of n?



- Number of iterations gets larger as  $n$  increases
- Number of iterations scales linearly with the value of  $n$

```
(defun first-n-lin (n)
  (do ((i 1 (1+ i))
      (sum 0 (+ i sum)))
      ((> i n) sum)
  )
)
(defun first-n-const (n)
  (/ (* n (+ n 1)) 2)
)
```



**Cost is the same regardless of the value of  $n$**

- The operations performed do not change with the value of  $n$

# Big O

notation

**Upper-bound analysis**

# Big-O Notation

---

- We can use Big-O notation to represent the **cost** of an algorithm with respect to input size, **n**.
- Let's assume we have a **cost function** for our algorithm:

$$\text{Let } f(n) = 2n^2 + 4n + 9$$

- This represents the number of “steps” our algorithm takes to complete, as a function of **n** (size of input).
- **Steps?** Could be anything. Common metric is counting comparisons, lines of code, machine instructions, etc.

# Counting Steps: Simple Example

Count “steps” (usually just a line of code) that runs in constant time:

```
min=arr[0], i=0;
while (i<arr.length)
{
    if (arr[i]<min)
        min=arr[i];
    i++;
}
```

1 step

1 step (loop overhead)

3 steps inside loop

$$f(n) = 1 + \sum_{i=0}^{n-1} 4$$

$$f(n) = 1 + 4n$$

# Counting Steps: Simple Example

---

Counting steps: Not always simple

```
int min=arr[0], i=0;
while (i<10) ←
{
    if (arr[i]<min)
        min=arr[i];
    i++;
}
```

Loop disconnected  
from input size

```
int min=arr[0], i=1;
while (i<arr.length)
{
    if (arr[i]<min)
        min=arr[i];
    i*=2; ←
}
```

Iterations not linear  
with input size

# Big-O Notation

---

**However,** Let's assume we have a *cost function* for our algorithm:

$$f(n) = 2n^2 + 4n + 9$$

Given a cost function, we can find the Big-O time complexity

**Two simple steps:**

- 1) If  $f(n)$  is the sum of several terms, keep only the fastest growing.
- 2) If the remaining term is a product of several factors, remove any coefficients.

# Big-O Notation

---

## Two simple steps:

- 1) If  $f(n)$  is the sum of several terms, keep only the fastest growing.
- 2) If the remaining term is a product of several factors, remove any coefficients.

$$f(n) = 2n^2 + 4n + 9$$

- The remaining term is our Big-O complexity class.
- In this case,  $O(n^2)$
- The cost of the algorithm grows *quadratically* with  $n$

# Big-O Notation

---

But why? Doesn't this stuff matter?

$$\text{Let } f(n) = \underline{2n^2} + \underline{4n} + 9$$

**Big-O is a measure of algorithm cost as  $n$  approaches infinity:**

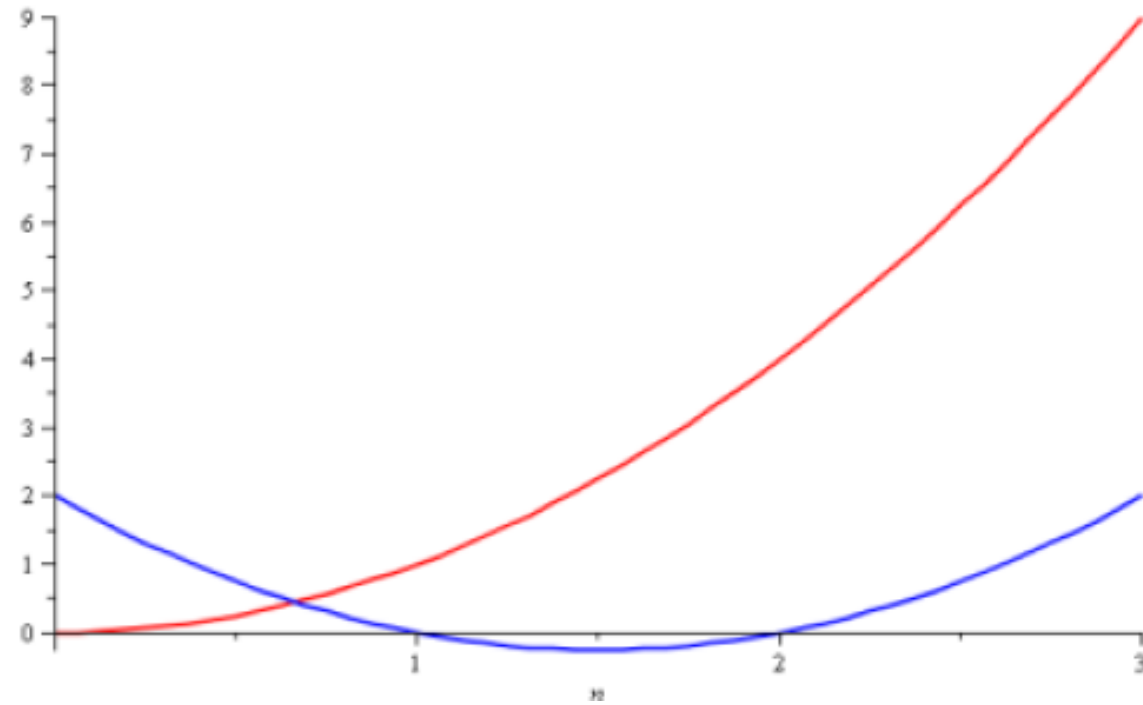
- As  $n$  gets ***larger***, the proportional contribution of the removed terms to the total run-time gets ***smaller***.
- As  $n$  approaches ***infinity***, the proportional contribution of the removed terms approaches ***zero***.
- High order coefficient? The ratio asymptotically approaches *that coefficient*.



1. As  $n$  approaches infinity, the proportional contribution of the removed terms approaches zero.

Consider two functions:  $f(n) = n^2$        $g(n) = n^2 - 3n + 2$

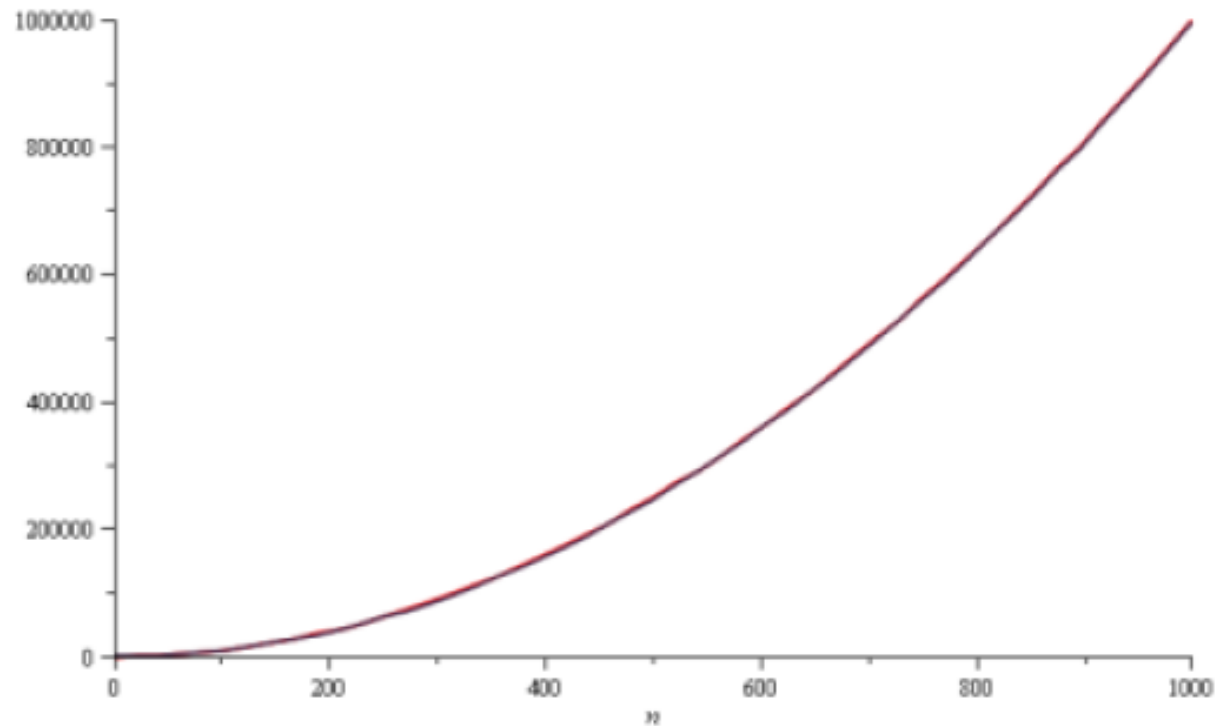
**Around  $n = 0$ :**  
Very different



1. As  $n$  approaches infinity, the proportional contribution of the removed terms approaches zero.

Consider two functions:  $f(n) = n^2$        $g(n) = n^2 - 3n + 2$

**Around  $n = 1000$ :**  
Indistinguishable



1. As  $n$  approaches infinity, the proportional contribution of the removed terms approaches zero.

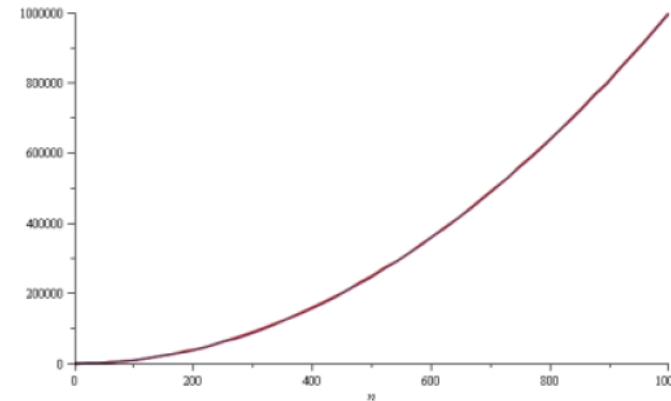
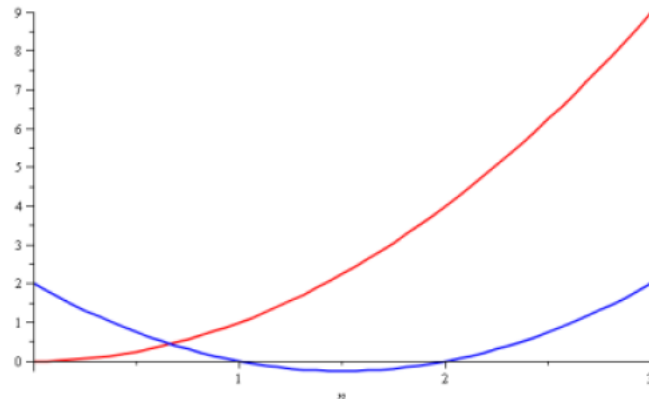
*Absolute* difference is large:

$$f(1000) = 1000000, \quad g(1000) = 997002$$

*Relative* difference is small:

$$\text{abs}(f(1000) - g(1000)) / f(1000) = 0.002998 < 0.3\%$$

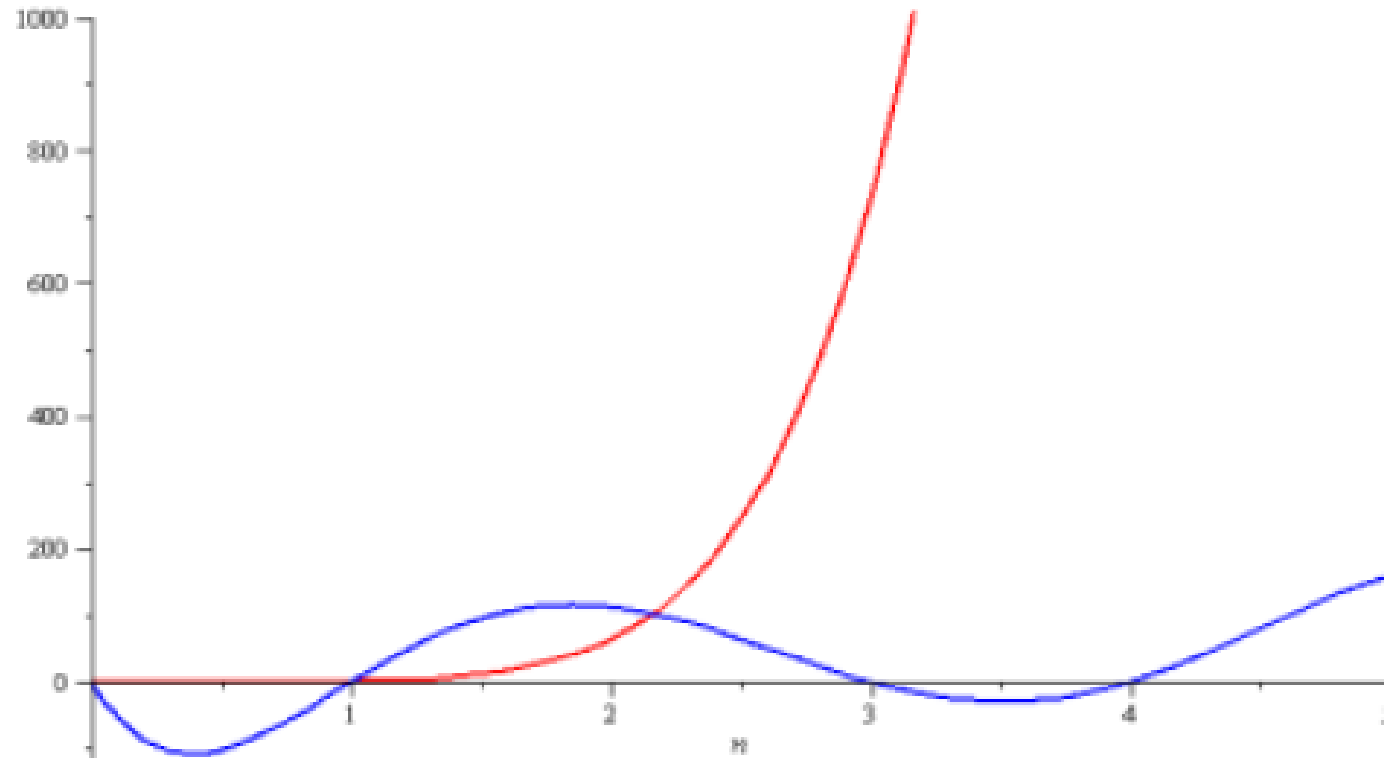
This relative difference approaches zero as  $n$  approaches  $\infty$



# Polynomial Growth

$$f(n) = n^6 \quad g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$$

Around  $n = 0$ :  
Very different



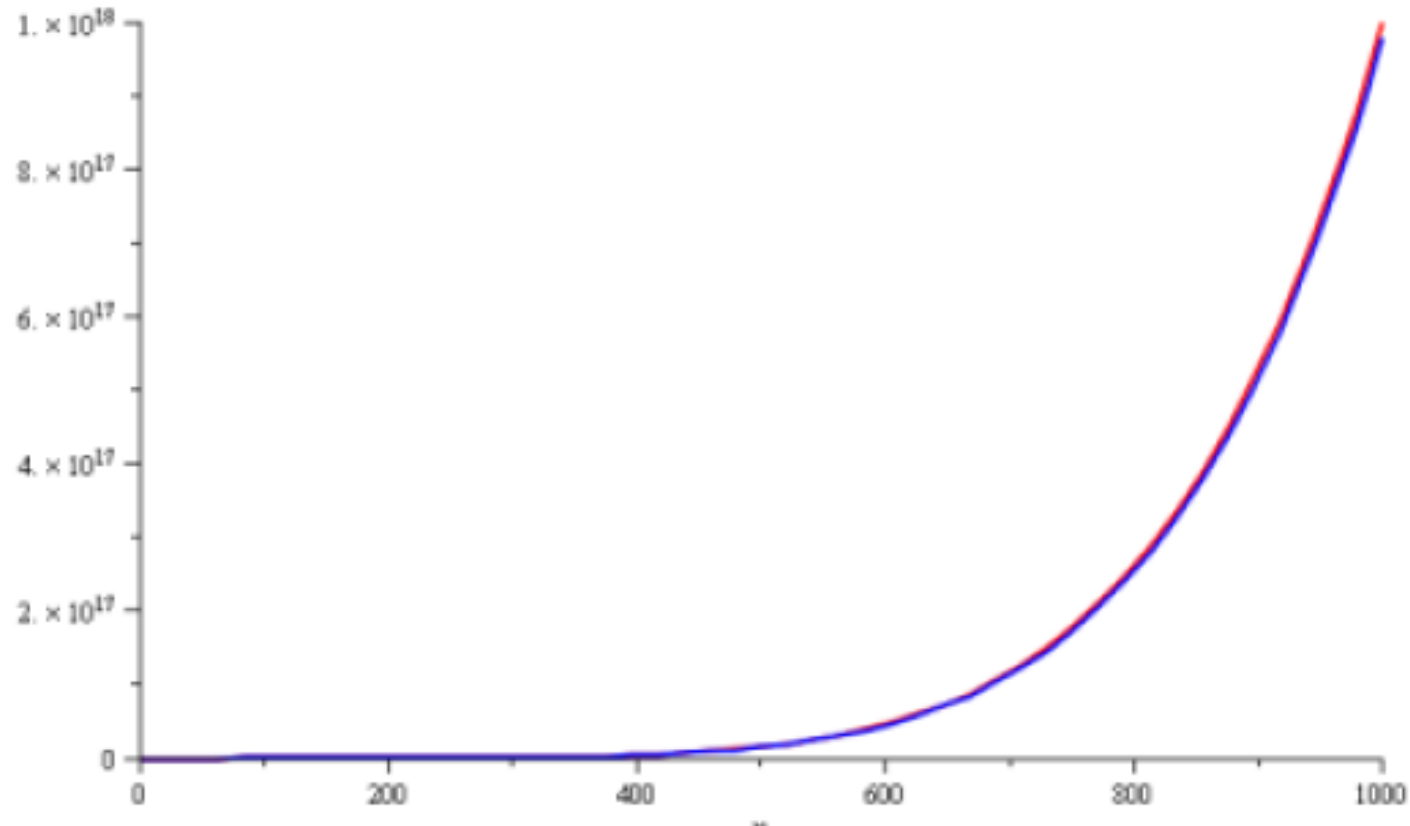
# Polynomial Growth

---

$$f(n) = n^6 \quad g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$$

Around  $n = 1000$ :

Difference < 3%



# Big-O Notation

---

What about the coefficient?

$$\text{Let } f(n) = \underline{2}n^2 + 4n + 9$$

**Surely, we care about one algorithm being twice as fast as another?**

- In practice, of course we do. Mathematically? Not so much.
- $2n^2$ ,  $3n^2$ ,  $5n^2$ ,  $1000000n^2$ , are all the same complexity:  $O(n^2)$
- You'll see the mathematical justification for this in CPS616.

# Common Complexity Classes

---

All log bases  
are equivalent

$O(1)$

constant

$O(\log_b(n))$

logarithmic

$O(n)$

linear

$O(n \ln(n))$

log-linear, “n-log-n”

$O(n^2)$

quadratic

$O(n^3)$

cubic

$O(n^b)$

“polynomial”

$O(b^n)$

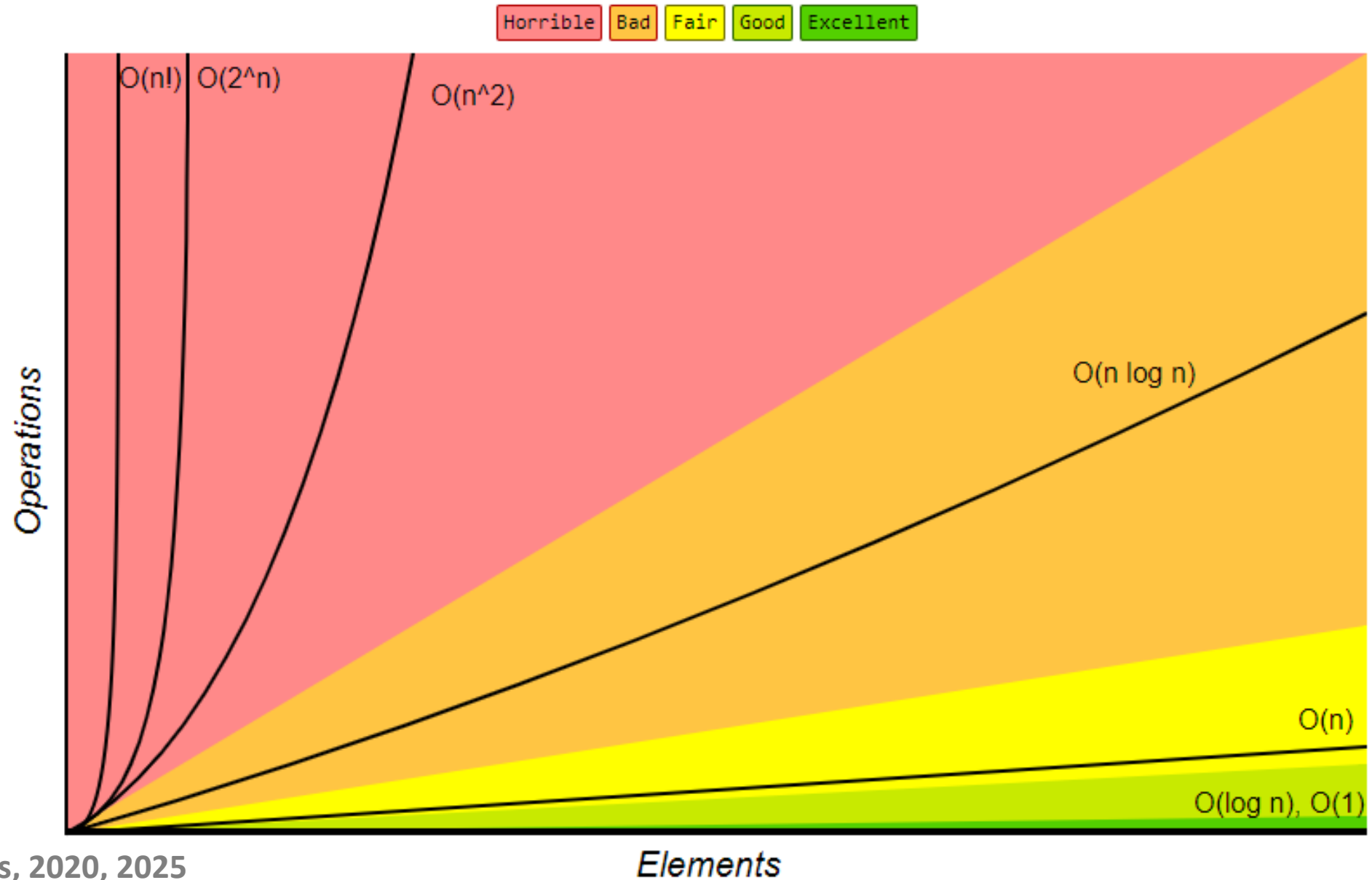
“exponential”

$O(n!)$

factorial

**Careful!** Polynomials  
and exponentials with  
different values of **b** are  
in different classes.

# Big-O Complexity Chart





# Big-O: Upper Bound

---

Big-O notation provides an upper bound (worst case)

- If an algorithm is  $O(n)$ , it runs in linear time *at worst*.
- That doesn't mean it can't run faster, depending on the input!

**Consider the following...**

# Big-O: Upper Bound

---

Write a function that accepts two arguments: a list and a value.

- Return TRUE if the value is in the list, and FALSE otherwise.

**Let's say the number of comparisons is our metric.**

- How many comparisons will we make in the worst case?
- How many comparisons will we make in the best case?

2	1	0	3	4	6	8	9	7	5	9	8	4	3	6	9	0	0	1	2	1	0	3	4	6	8	9	7	5	9	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Big-O: Upper Bound

---

Write a function that accepts two arguments: a list and a value.

- Return TRUE if the value is in the list, and FALSE otherwise.
- We make  $n$  comparisons in the worst case, where  $n$  is the length of the list (check every element, return FALSE).
- In the best case? We make a single comparison (found the element at the first position, return TRUE)

2	1	0	3	4	6	8	9	7	5	9	8	4	3	6	9	0	0	1	2	1	0	3	4	6	8	9	7	5	9	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Big-O: Upper Bound

---

Write a function that accepts two arguments: a list and a value.

- Return TRUE if the value is in the list, and FALSE otherwise.

**Thus:** Linear search is  $O(n)$

**The analysis is similar for binary search:**

- Best case? One comparison. Worst case?  $\log_2(n)$  comparisons
- Binary search is  $O(\log n)$

2	1	0	3	4	6	8	9	7	5	9	8	4	3	6	9	0	0	1	2	1	0	3	4	6	8	9	7	5	9	8
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Examples! LISP Code to Big-O

---

In imperative languages like Python and Java, this is easy:

**Count nested loops, taking care to identify cases where:**

- Iteration not linear with respect to problem size, or disconnected from problem size entirely
- Function calls hiding non-constant time computations.
- `find()`, `indexOf()`, `min()`, `max()`, etc.

```
int min=arr[0], i=0;
while (i<10)
{
    if (arr[i]<min)
        min=arr[i];
    i++;
}
```

```
int min=arr[0], i=1;
while (i<arr.length)
{
    if (arr[i]<min)
        min=arr[i];
    i*=2;
}
```

# Examples! LISP Code to Big-O

---

LISP's syntax is unfamiliar, so it will feel trickier

## Some basics:

- If the form does not perform repetition, or contain an inner form that performs repetition, it is constant time -  **$O(1)$**
- For example, (**SETF** **var** **form**) is  $O(1)$ , as long as **form** is constant time.
- Also true of **LET** and **LET\***

# Examples! LISP Code to Big-O

---

Also true of **LET** and **LET\***

```
(let ((a 5) (b 6) (c 10) (d 0) (e 40)
      (k 1) (x 1) (y 1) (v 1) (w 1) (z 1))
  (setf a (* (/ b c e) (+ d k x y v w z))))
```

# Examples! LISP Code to Big-O

---

`(DOTIMES preamble bodyForm*)` –or– `(DO preamble bodyForm*)`

- If **preamble** form is constant, analysis can be based solely on **bodyForm\***
- Below, **i** goes from 0-(n-1). Both body forms are  $O(1)$

```
(dotimes (i n)
  (setf x (+ i 1))
  (setf y (* x x)))
```

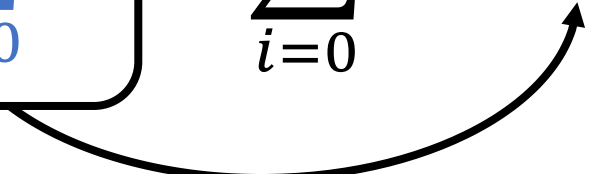
$$T(n) = \sum_{i=0}^{n-1} (1 + 1) = 2n = O(n)$$



# Examples! LISP Code to Big-O

---

```
(dotimes (i n) ; Repeats n times
  (setf x (+ i 1)) ; Constant time
  (dotimes (j n) ; Repeats n times
    (setf y (+ j x)))) ; Constant time
```

$$T(n) = \sum_{i=0}^{n-1} (1 + \sum_{j=0}^{n-1} 1) = \sum_{i=0}^{n-1} (1 + n) = (n + n^2) = \mathbf{O}(n^2)$$


# Examples! LISP Code to Big-O

---

```
(let ((a 5) (b 6) (c 10) (d 0) (k 1) (x 1) (y 1) (v 1) (w 1) (z 1))
  (dotimes (i n)
    (dotimes (j n)
      (setf z (* i i))
      (setf y (/ j j))
      (setf z (+ i j))
    )
  )
  (dotimes (k n)
    (setf w (+ (* a k) 45))
    (setf v (* b b))
  )
  (setf d 33)
)
```

# Examples! LISP Code to Big-O

```
(let ((a 5) (b 6) (c 10) (d 0) (k 1) (x 1) (y 1) (v 1) (w 1) (z 1))
  (dotimes (i n)
    (dotimes (j n)
      (setf z (* i i))
      (setf y (/ j j))
      (setf z (+ i j))
    )
  )
  (dotimes (k n)
    (setf w (+ (* a k) 45))
    (setf v (* b b))
  )
  (setf d 33)
)
```

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 3$$

$$\sum_{k=0}^{n-1} 2$$

$$1$$

# Examples! LISP Code to Big-O

```
(let ((a 5) (b 6) (c 10) (d 0) (k 1) (x 1) (y 1) (v 1) (w 1) (z 1))
  (dotimes (i n)
    (dotimes (j n)
      (setf z (* i i))
      (setf y (/ j j))
      (setf z (+ i j))
    )
  )
  (dotimes (k n)
    (setf w (+ (* a k) 45))
    (setf v (* b b))
  )
  (setf d 33)
)
```

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 3 + \sum_{k=0}^{n-1} 2 + 1$$

$$\sum_{i=0}^{n-1} 3n + 2n + 1$$

$$3n^2 + 2n + 1 = O(n^2)$$

# Examples! LISP Code to Big-O

---

```
(setf k 0)
(do ((i 1 (* i 2))) ; update is i*i! Oh no!
    ((>= i n))
    (setf k (+ k (+ 2 2))))
)
```

$$\sum_{i=1}^n (...)$$

- Summation notation assumes an increment of 1
- Here, our update is  $i=i^2$  (i grows exponentially)
- We have a bit more work to do

# Examples! LISP Code to Big-O

---

```
(setf k 0)
(do ((i 1 (* i 2))) ; update is i*i! Oh no!
    ((>= i n))
    (setf k (+ k (+ 2 2)))
  )
```

If **i** starts at 1, how many times must we multiply by 2 before it reaches n?

$$\sum_{i=1}^? (\dots) \quad \begin{array}{l} 2^? = n \\ ? = \log_2 n \end{array}$$

$$T(n) = 1 + \sum_{i=1}^{\log_2 n} 1 = O(\log n)$$

# Examples! LISP Code to Big-O

---

```
(dotimes (i n)
  (do ((j 1 (* j 2)))
      ((>= j n))
      (setf a (+ i j))
  )
)
```

# Examples! LISP Code to Big-O

---

```
(dotimes (i n)
  (do ((j 1 (* j 2)))
      ((>= j n))
      (setf a (+ i j)))
  )
)
```

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=1}^{\log_2 n} 1$$

$$T(n) = \sum_{i=0}^{n-1} \log_2 n$$

$$T(n) = n \log_2 n = O(n \log n)$$



# Practice at Home

---

```
(defun f (a b)
  (let ((acc 0))
    (dotimes (i a)
      (incf acc)
    ) ) )
```

What is the time complexity  
in Big-O notation?

```
(defun g (n)
  (dotimes (i n)
    (dotimes (j n (f i j))
      (setf x (+ i j))
    ) ) )
```

# In Summary

---

- Continuing list, more advanced forms
- Time complexity analysis and Big-O notation
- Determining time complexity of Lisp code
- Examples!

