

# CS10 Final Review

## Programming

CS10 Final Review by [Glenn Sugden](#) is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](#).



# Concept Review

- Loops & Variables
- Conditionals
- Lists
- Algorithms & Complexity
- Concurrency
- Recursion
- Data Structures
- Hash Tables
- Lamdas & HOFs

# Loops & Variables



Multiplying operand1 by operand2

Correct?

# Loops & Variables



Multiplying operand1 by operand2

No! Why?

# Loops & Variables



Multiplying operand1 by operand2

Uninitialized Variable

# Loops & Variables



Multiplying operand1 by operand2

Initialized Variable!

# Loops & Variables

Be sure that your  
variables are initialized to a  
correct / known / sane value!

# Loops & Variables



Multiplying operand1 by operand2

Correct now?

# Loops & Variables



Multiplying operand1 by operand2

What if operand1 is negative? Or  
a “real” number like 2.3?

# Loops & Variables

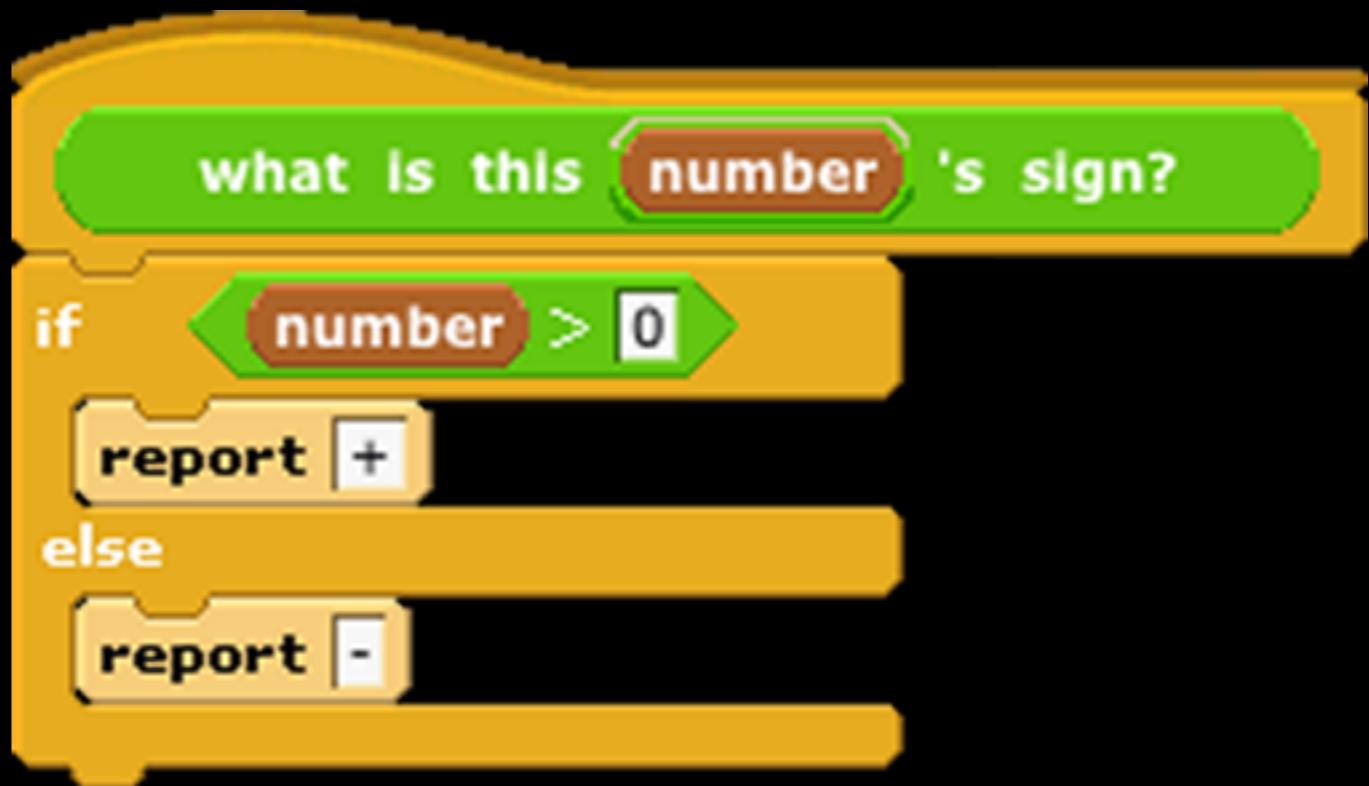
Be sure that your  
looping conditions  
are valid!



# Conditionals

Reports “+” for positive numbers, “-” for negative numbers.

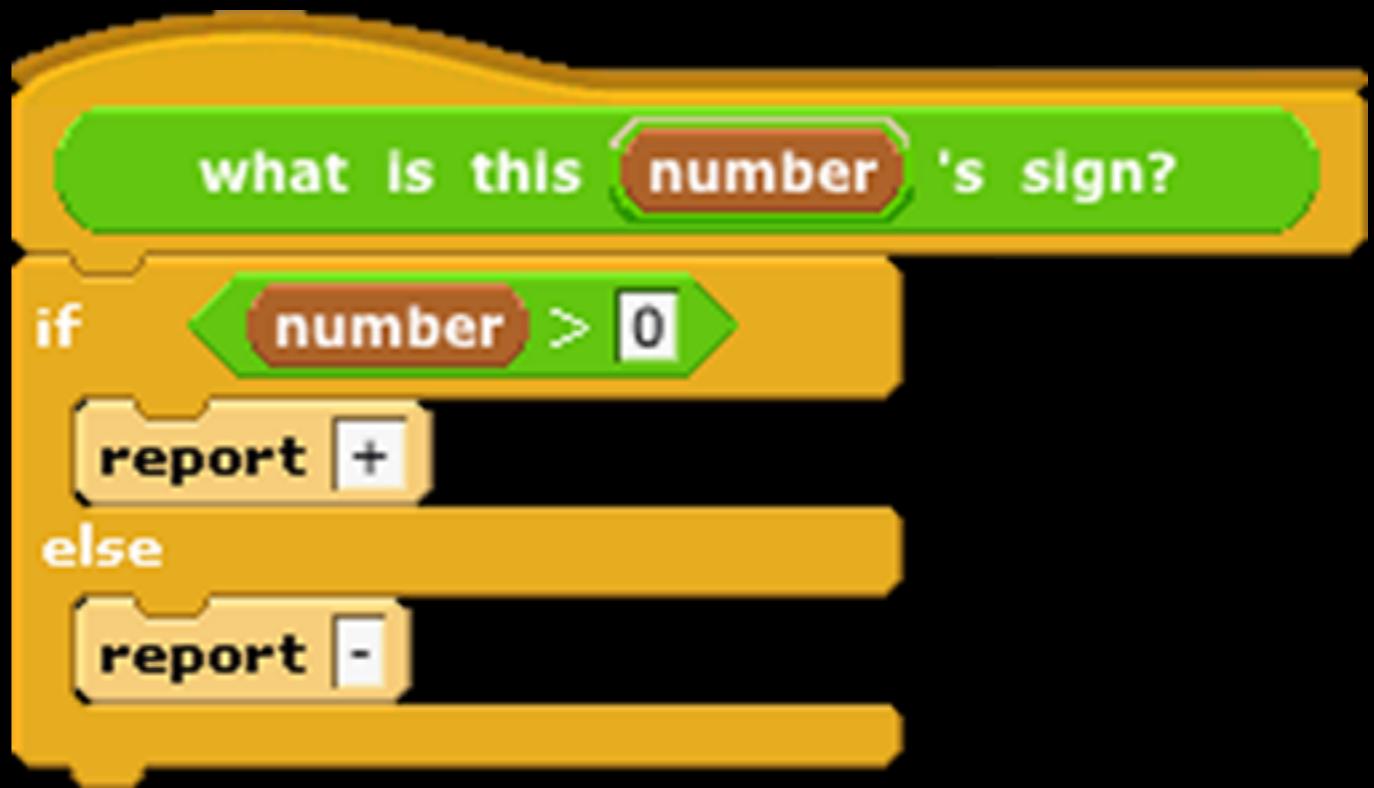
Correct?



# Conditionals

Reports “+” for positive numbers, “-” for negative numbers.

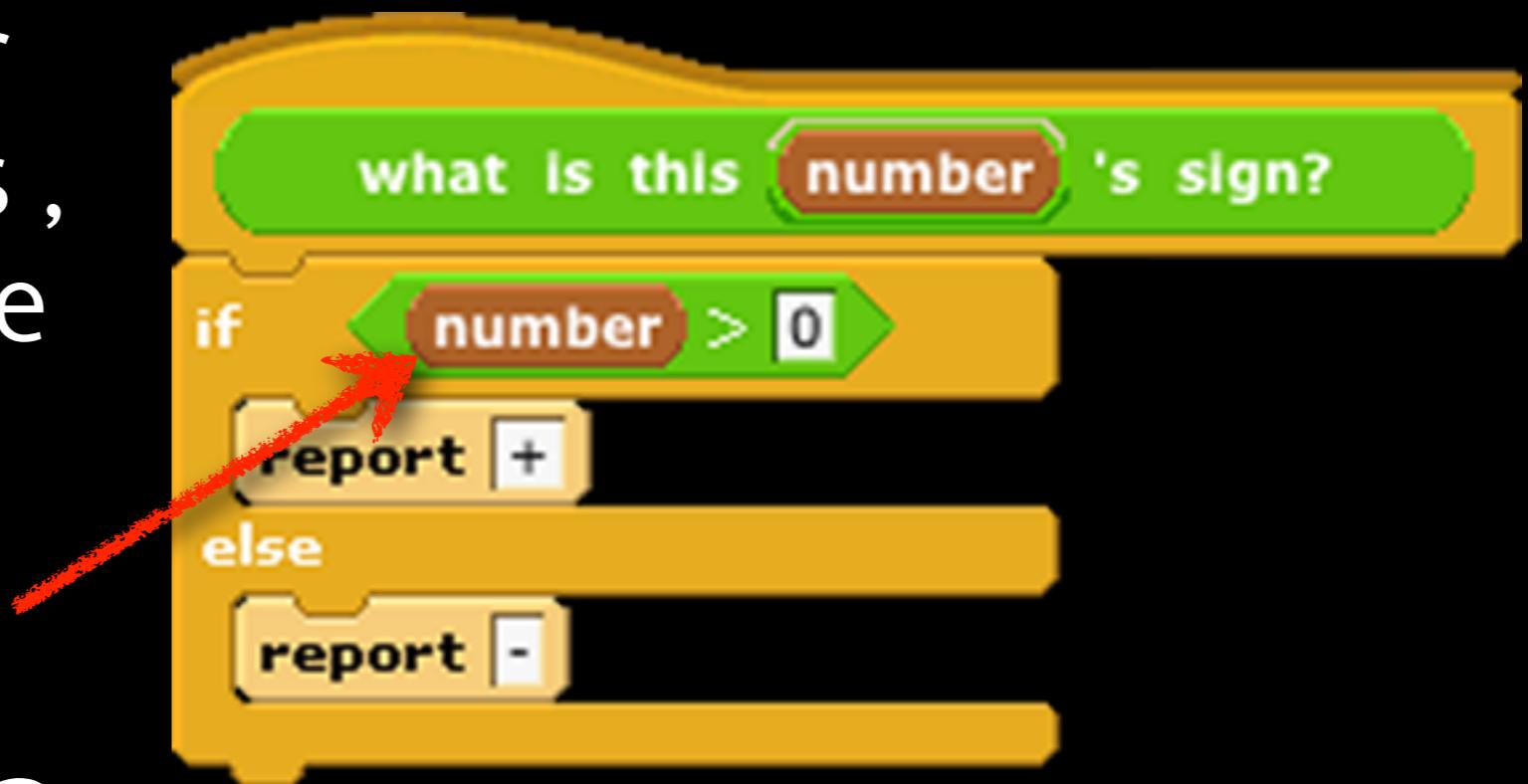
No! Why?



# Conditionals

Reports “+” for positive numbers, “-” for negative numbers.

What about  
number = 0?



# Conditionals Loops & Variables

- Be sure that your conditionals handle ***all*** possible cases!
- Double-check edge cases, or inputs that fall on either side of your predicate.

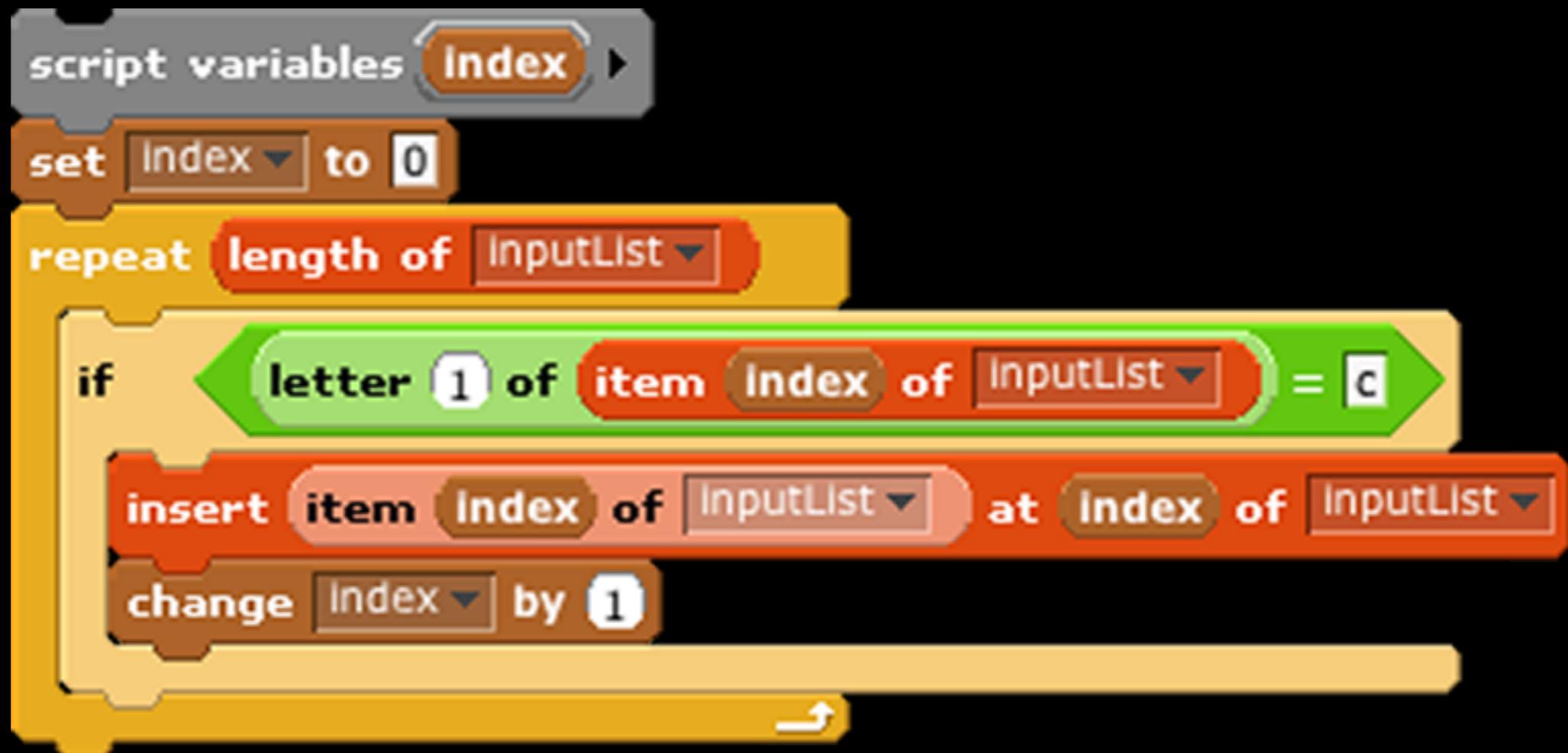
# Lists



Duplicate words  
beginning with ‘c’

Does this correctly loop over list?

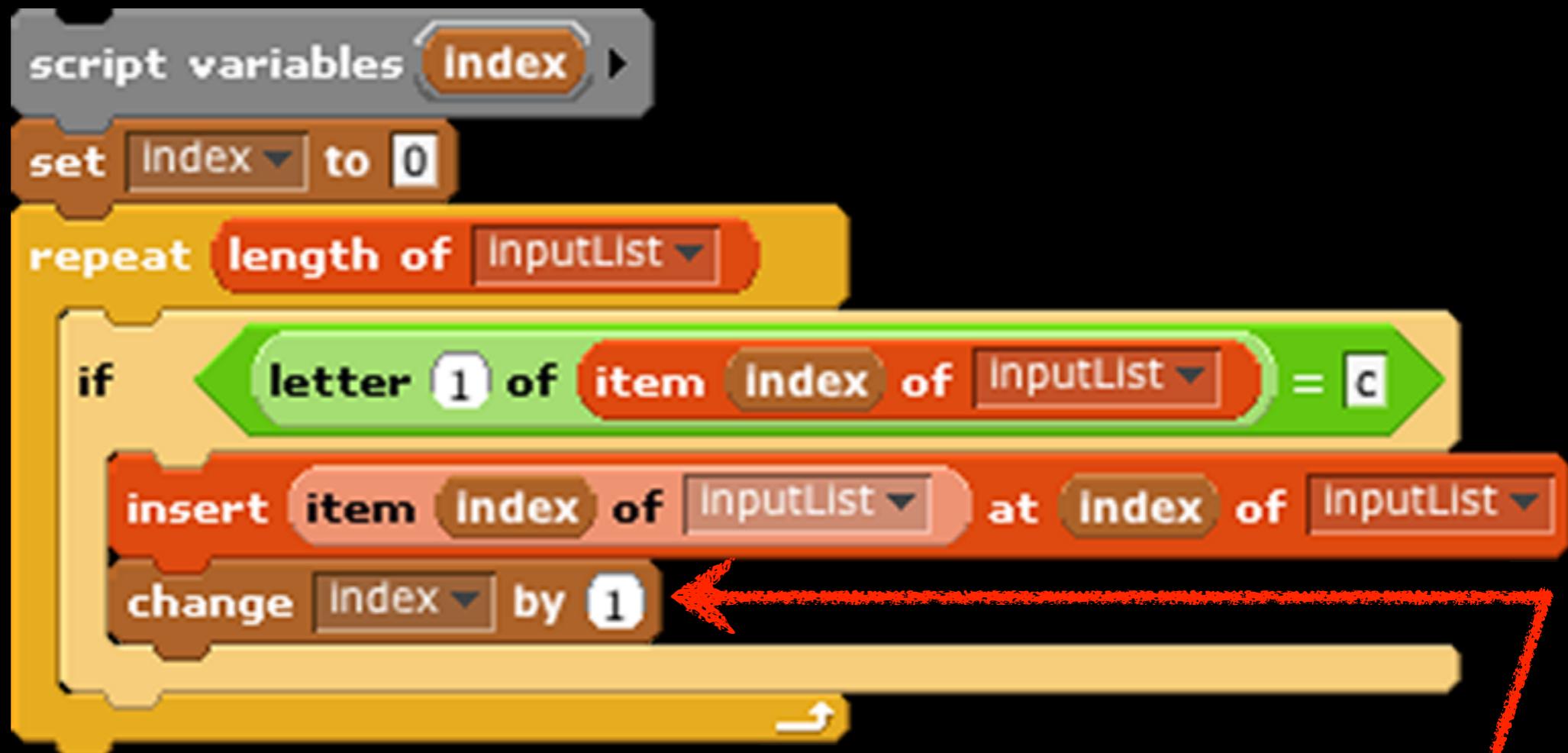
# Lists



Duplicate words  
beginning with ‘c’

No! Why?

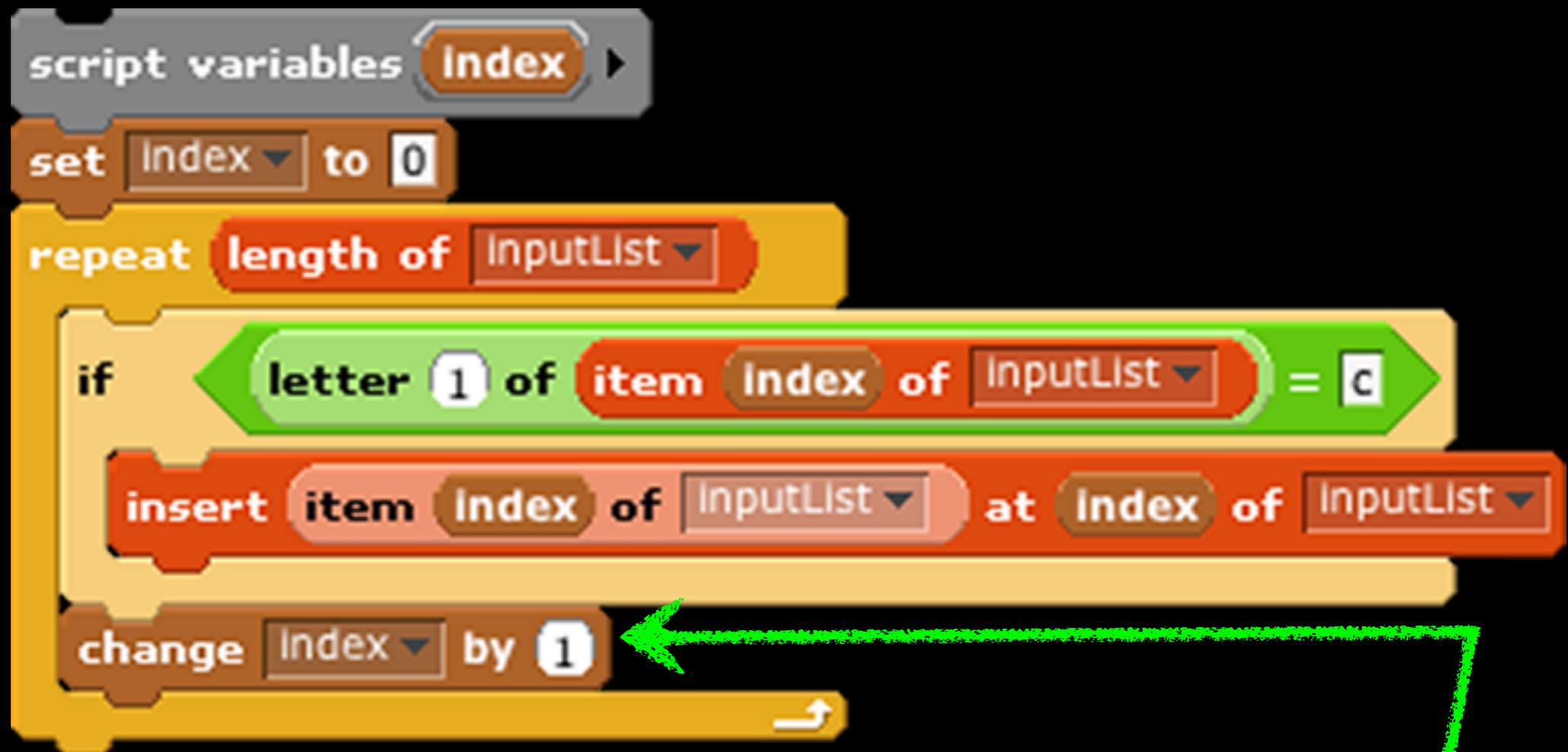
# Lists



Duplicate words  
beginning with ‘c’

Index within conditional!

# Lists



Duplicate words  
beginning with ‘c’

Correct now?

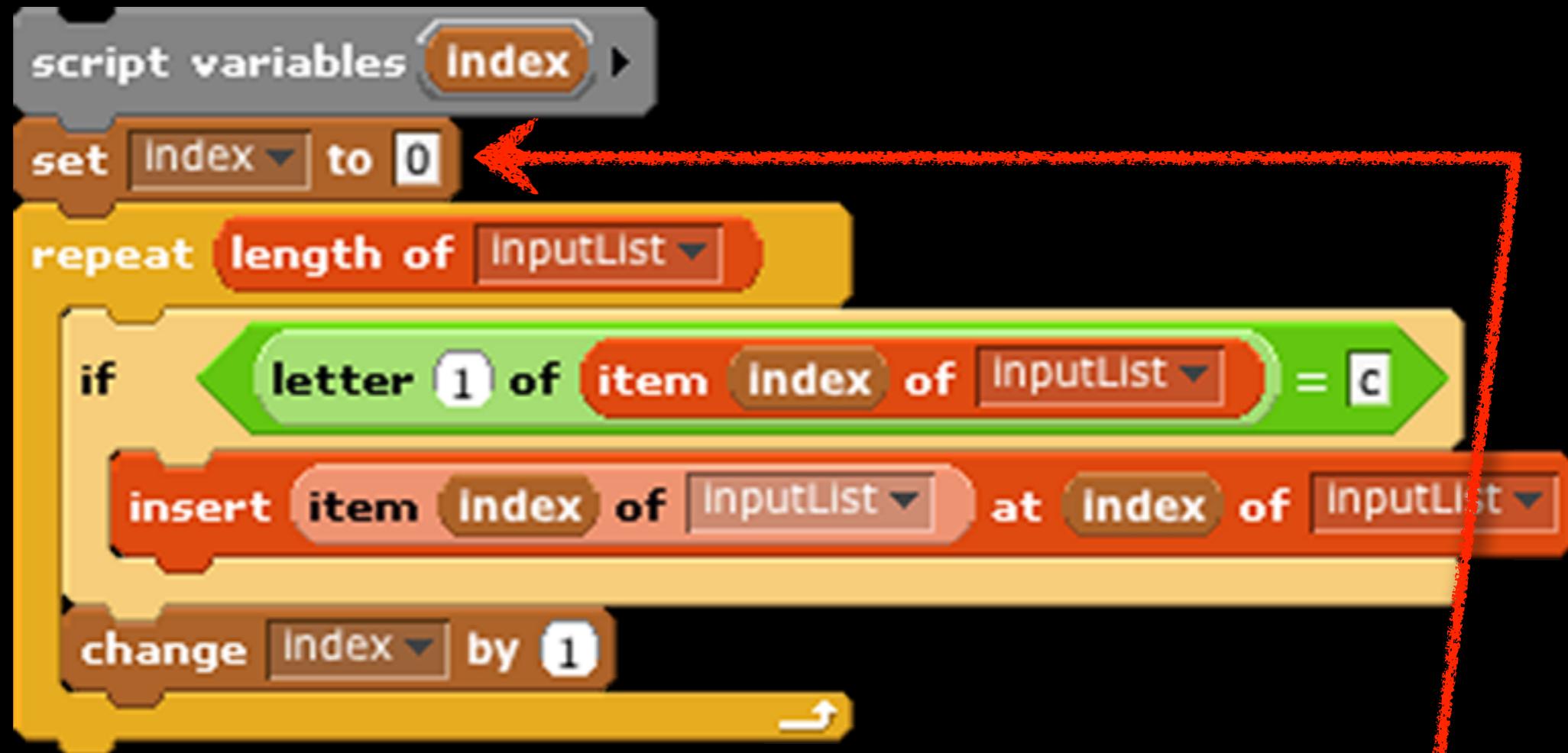
# Lists



Duplicate words  
beginning with 'c'

No! Why?

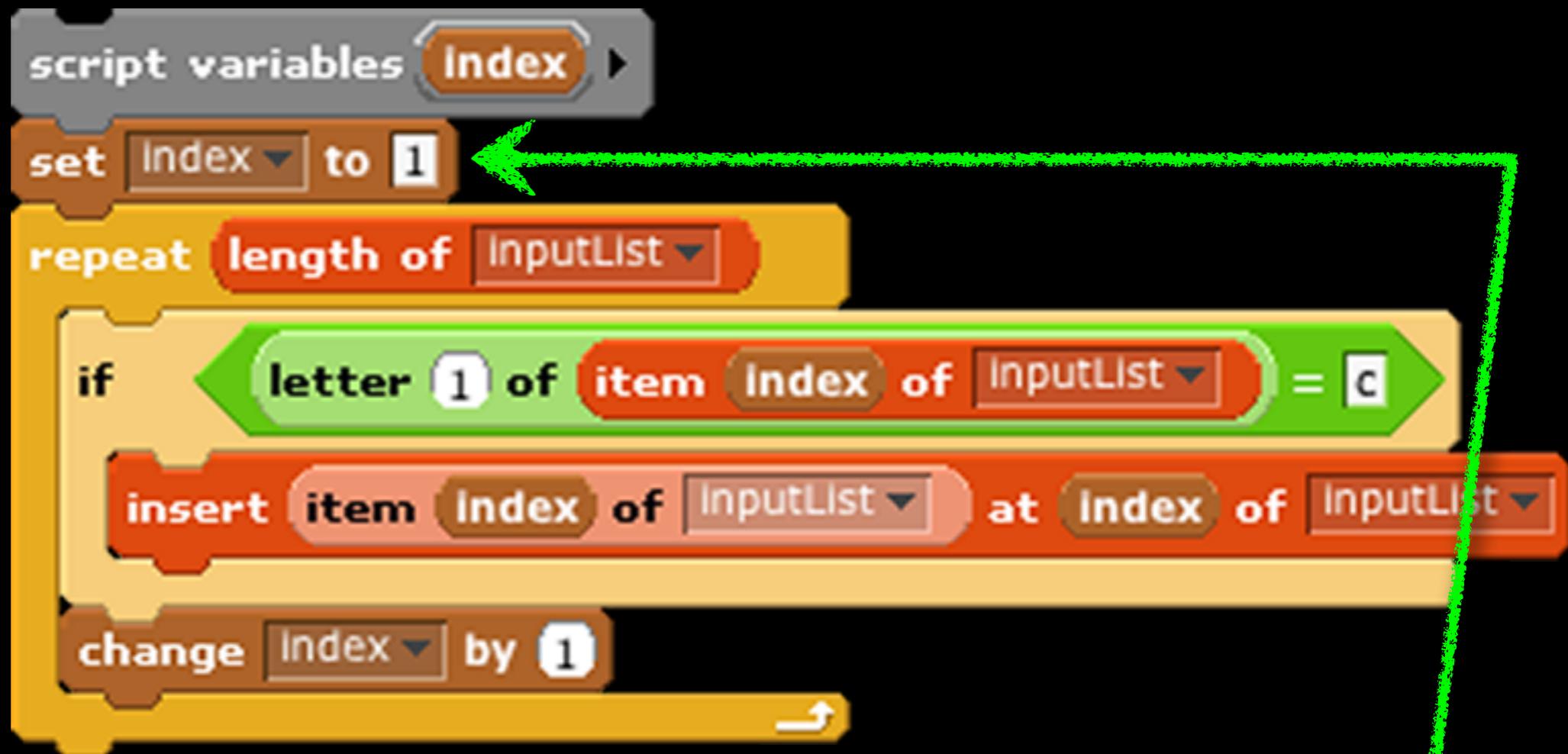
# Lists



Duplicate words  
beginning with 'c'

Off by 1! —

# Lists



Duplicate words  
beginning with ‘c’

Correct now?

# Lists

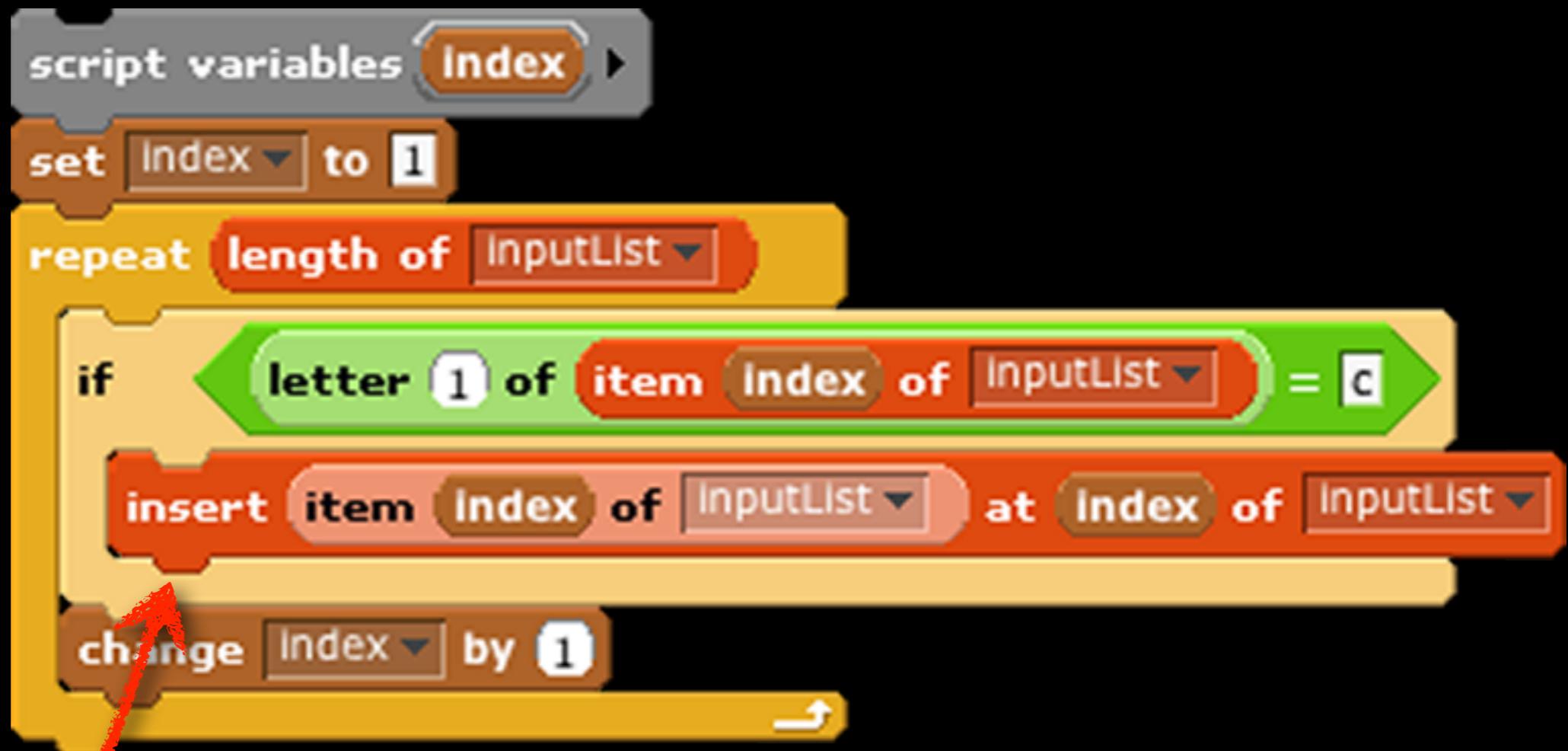


```
script variables [Index v]
set [Index v] to [1]
repeat (length of [InputList v])
  if (letter [1] of [item v] [Index v] of [InputList v]) = [c]
    insert [item v] [Index v] of [InputList v] at [Index v] of [InputList v]
    change [Index v] by [1]
end
```

Duplicate words  
beginning with 'c'

No! Why?

# Lists



Duplicate words  
beginning with ‘c’

The list keeps changing size!

# Lists

inputList

1	ace
2	blast
3	burger
4	coward
5	culling
6	dragon
7	dynam0

+ length: 7

Duplicate words  
beginning with 'c'



inputList

1	ace
2	blast
3	burger
4	coward
5	coward
6	coward
7	coward
8	coward
9	culling
10	dragon
11	dynam0

+ length: 11

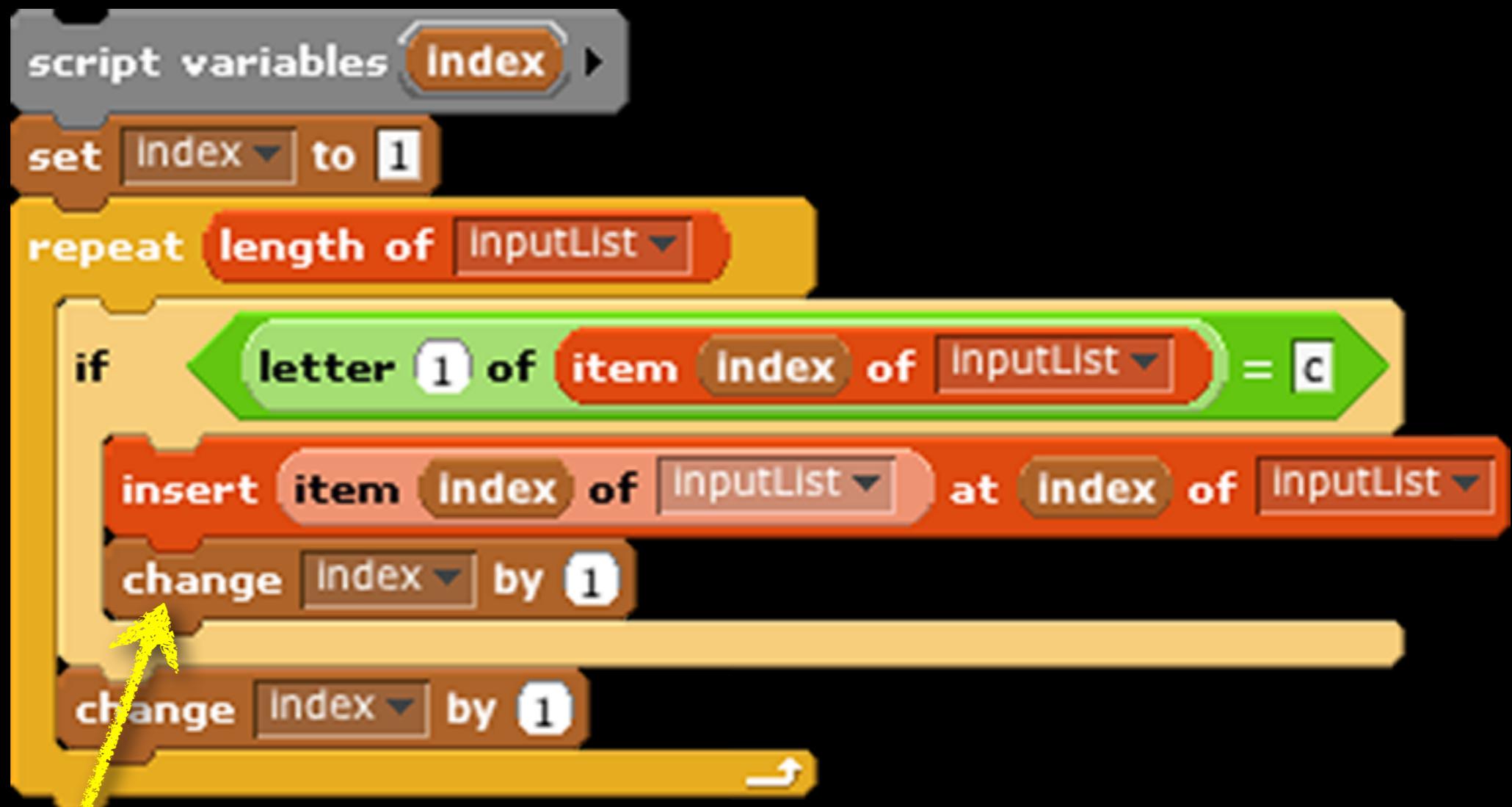
The list keeps changing size!

# Lists

How do you correct it?

Here is one solution...

# Lists



Duplicate words  
beginning with ‘c’

Push the index past the inserted item...

# Lists

Seems pretty “kludgy” though...

Here is a much, much better solution...



# Lists

**inputList**

1	ace
2	blast
3	burger
4	coward
5	culling
6	dragon
7	dynam0

+ length: 7

The Scratch script uses a repeat loop to iterate through each item in the **inputList**. Inside the loop, it checks if the first letter of the current item is 'c'. If true, it adds the item to the **resultingList**. The **resultingList** is initially set to empty.

```
script variables [Index v]
set [Index v] to [1]
repeat (length of [inputList v])
    add [item [Index v] of [inputList v]] to [resultingList v]
    if [letter [1] of [item [Index v] of [inputList v]] = [c]] then
        add [item [Index v] of [inputList v]] to [resultingList v]
    end
    change [Index v] by [1]
end
end
```

Duplicate words  
beginning with ‘c’

Make a new, resulting list!

**resultingList**

(empty)

+ length: 0

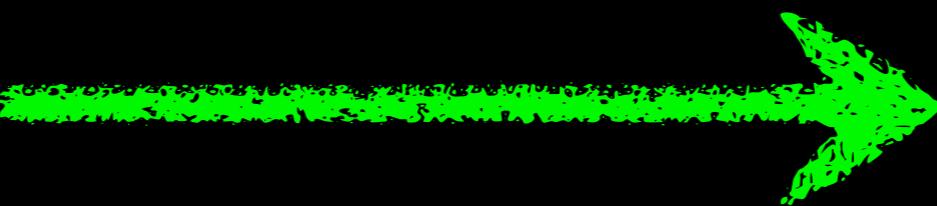
# Lists

inputList

1	ace
2	blast
3	burger
4	coward
5	culling
6	dragon
7	dynam0

+ ━ length: 7

Duplicate words  
beginning with 'c'



resultingList

1	ace
2	blast
3	burger
4	coward
5	coward
6	culling
7	culling
8	dragon
9	dynam0

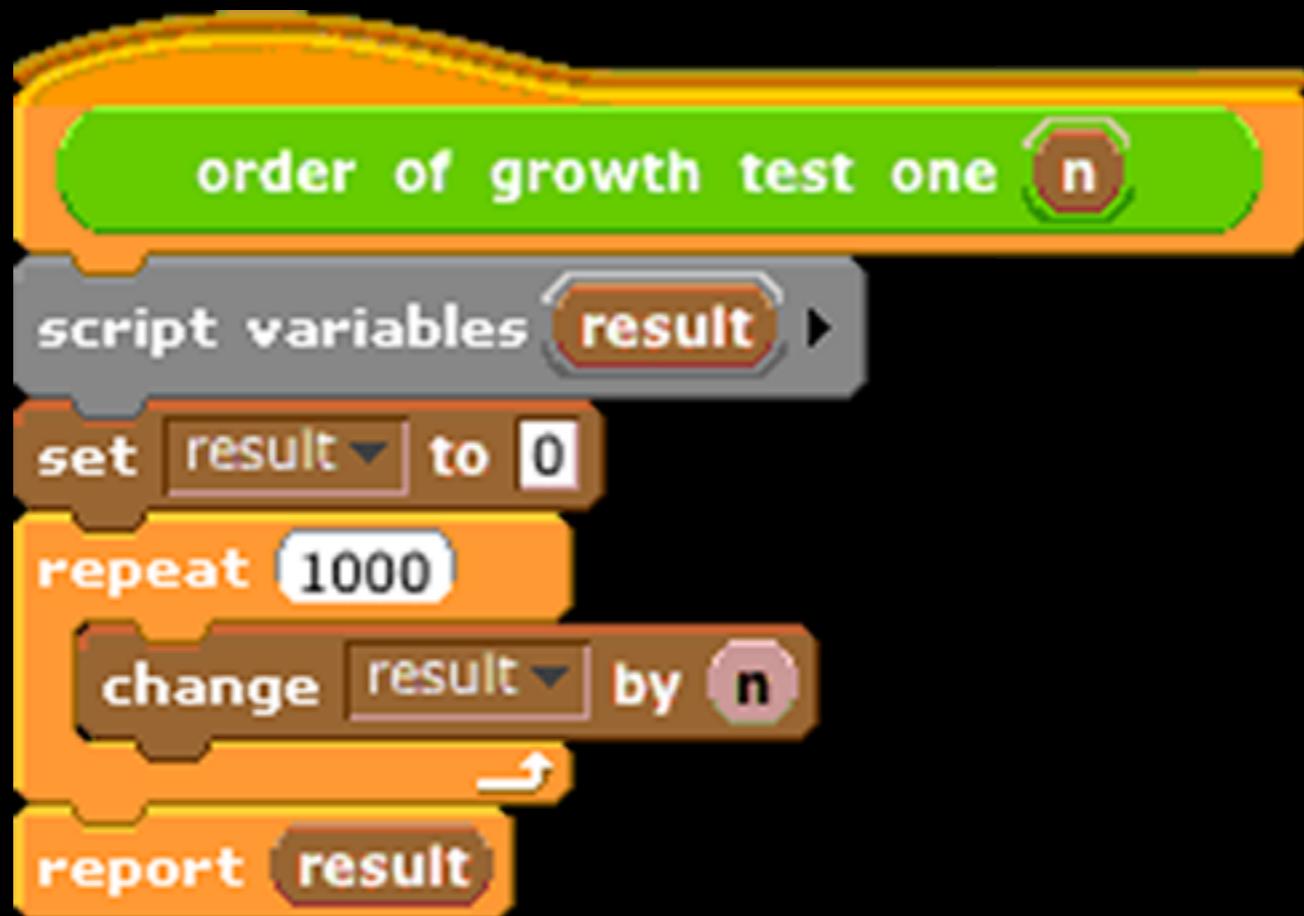
+ ━ length: 9

Much better! And our  
original list is still  
intact!

# Check-Lists

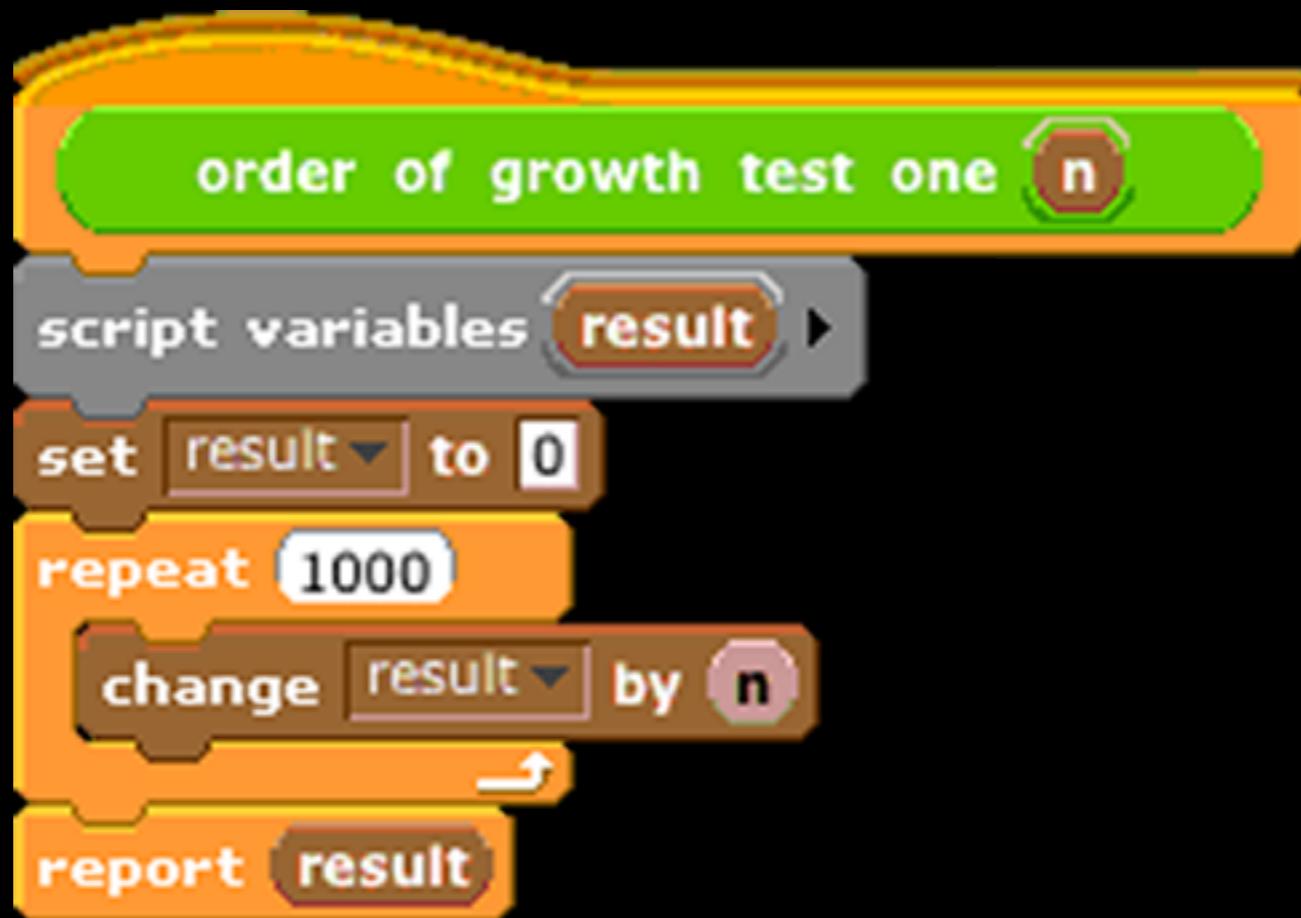
- Be sure indexes are correct during the *entire* loop.
- BYOB starts list indexing at 1
- If the input list changes size during the loop, your index will be off!

# Algorithms & Complexity



Order of growth?

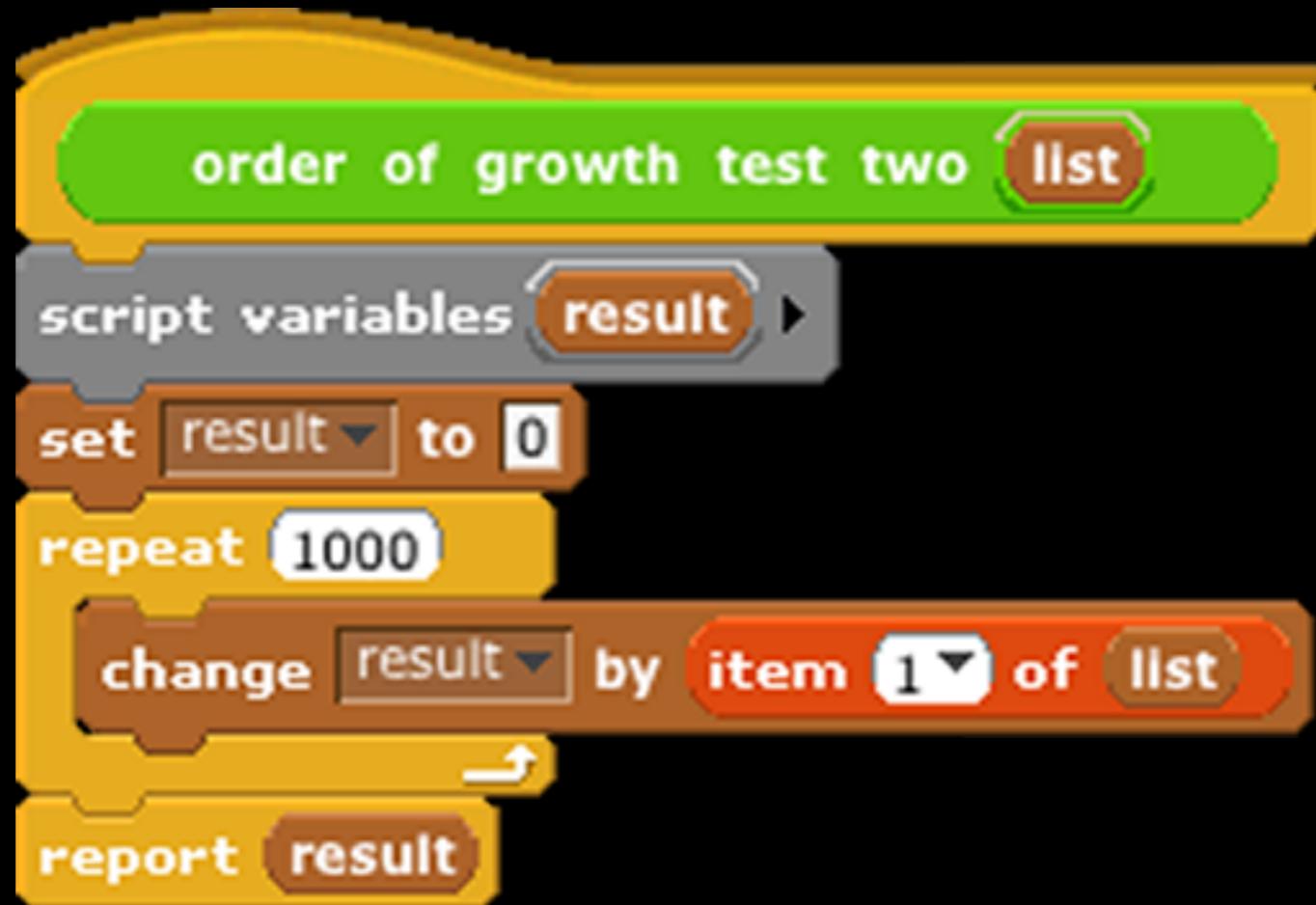
# Algorithms & Complexity



Constant:  $O(c)$

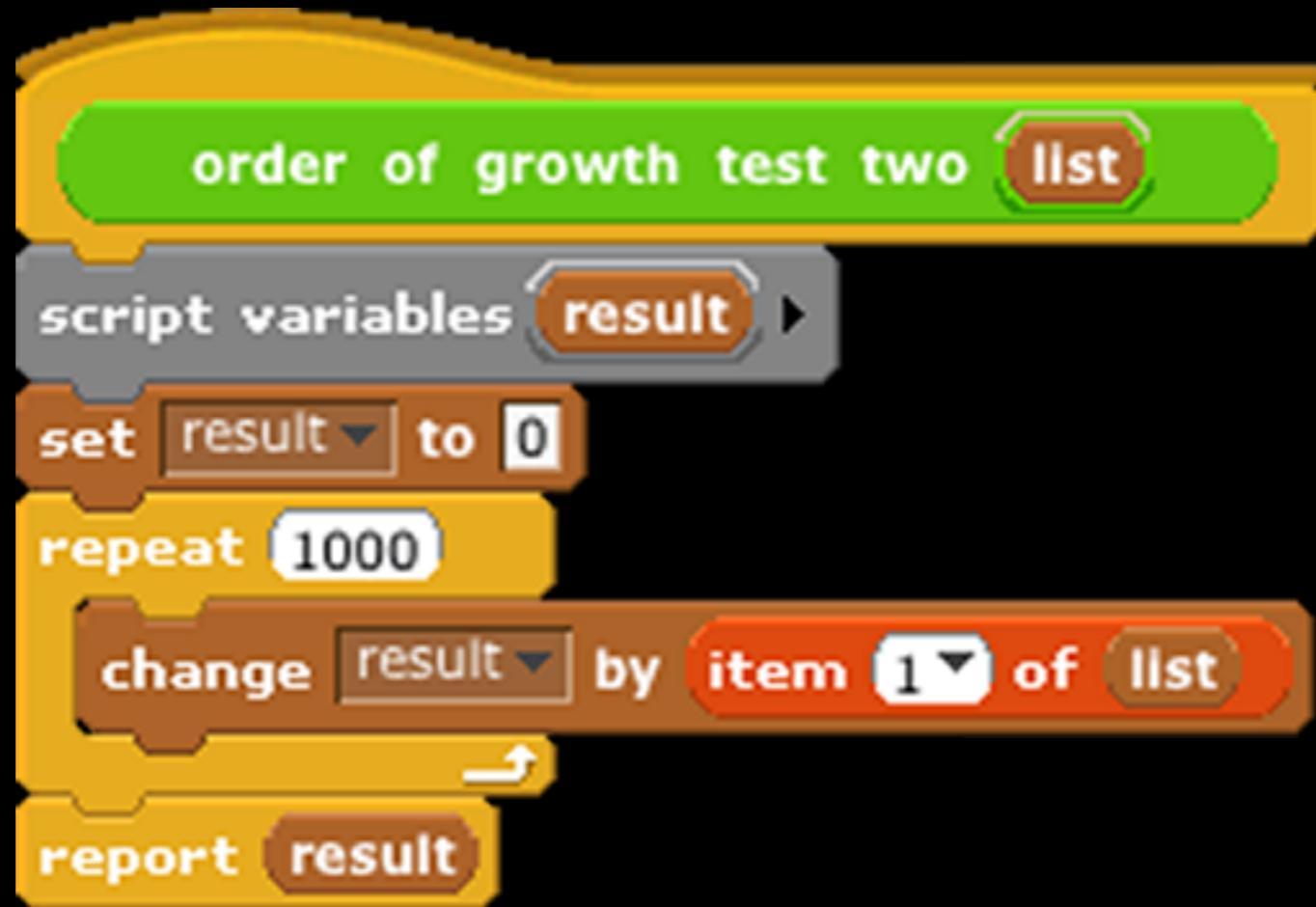
Reason: No matter what “n” is, the loop *always* repeats 1000 times.

# Algorithms & Complexity



Order of growth?

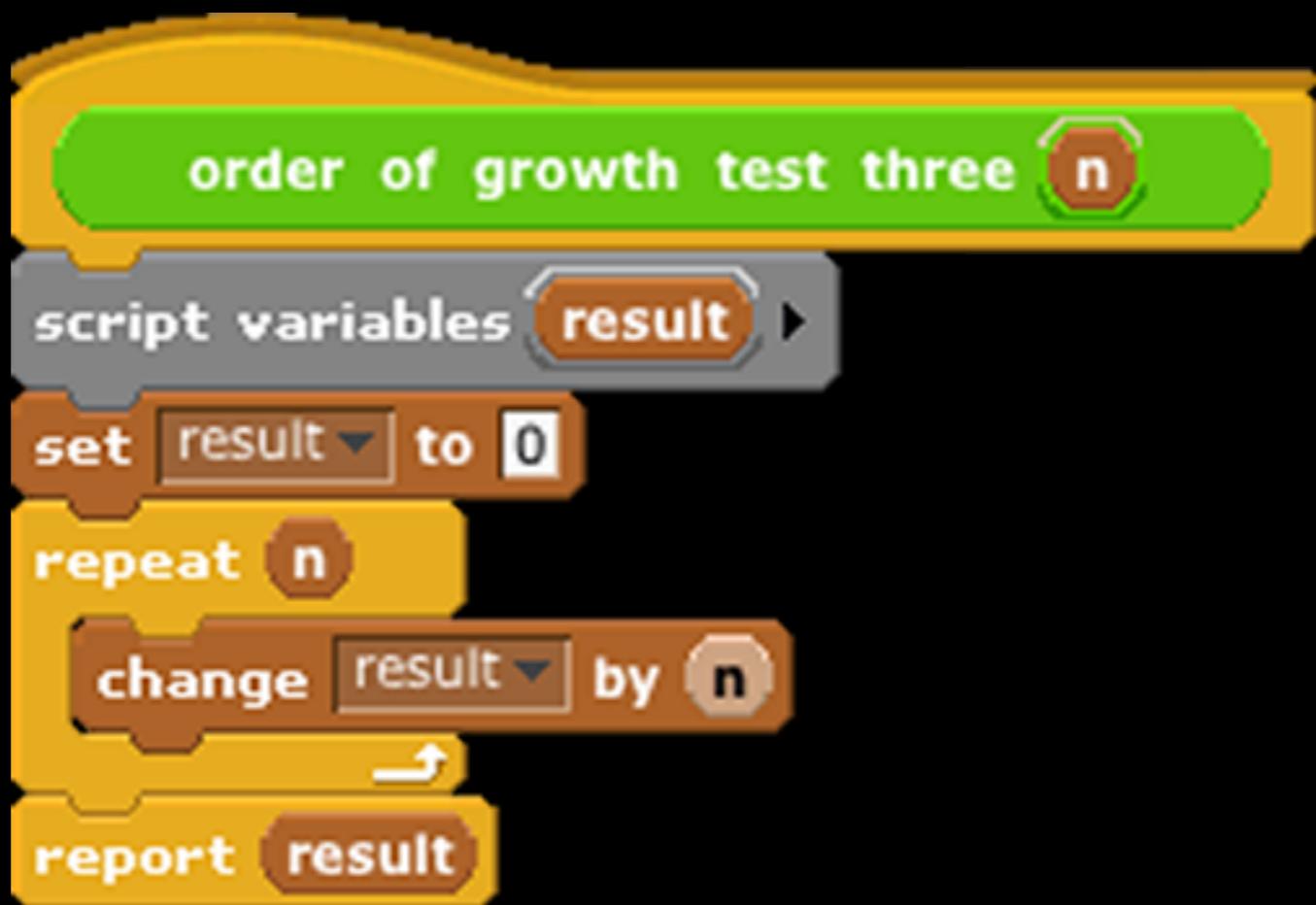
# Algorithms & Complexity



Still Constant:  $O(c)$

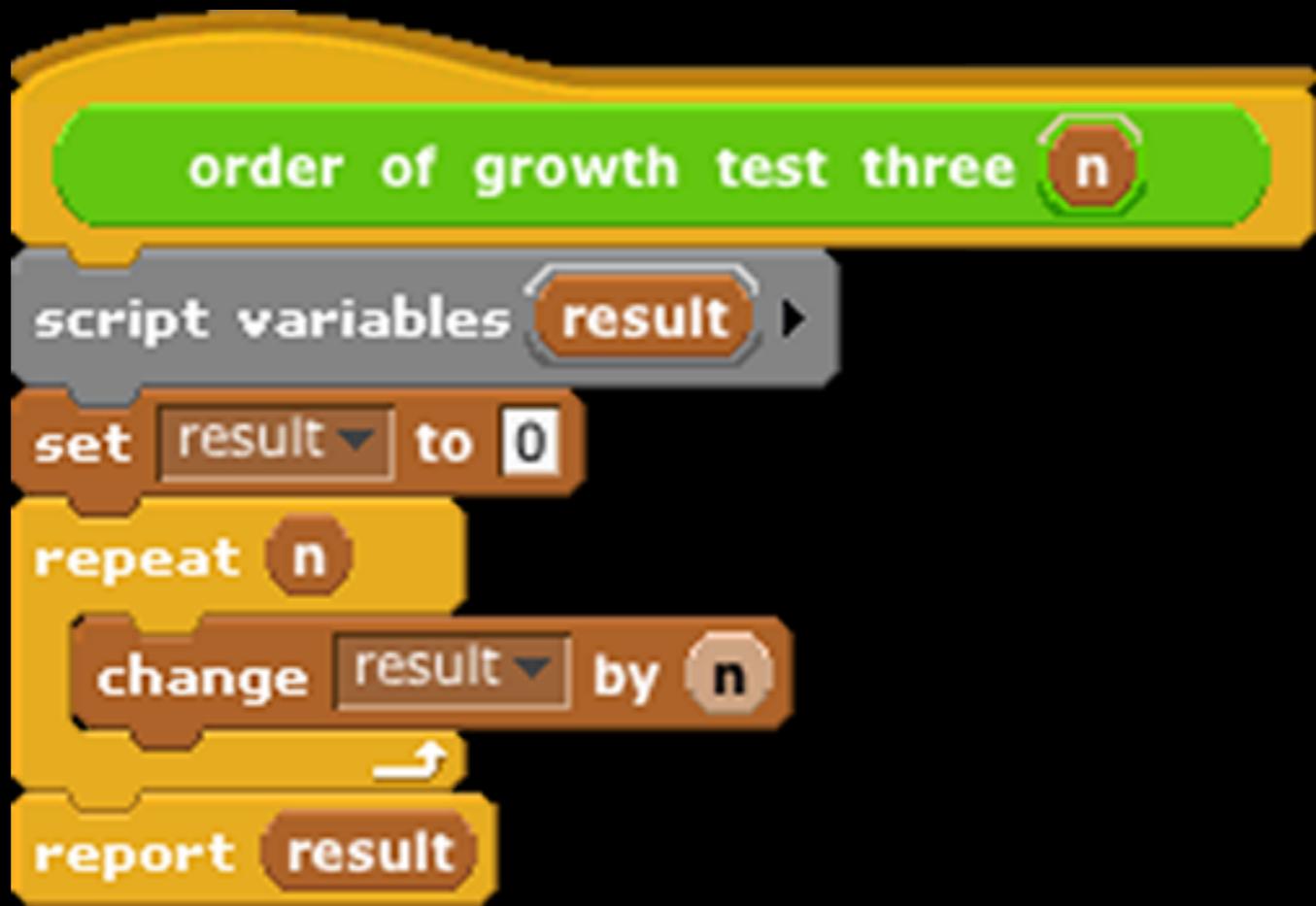
Reason: No matter what the list is,  
the loop *always* repeats 1000 times.

# Algorithms & Complexity



Order of growth?

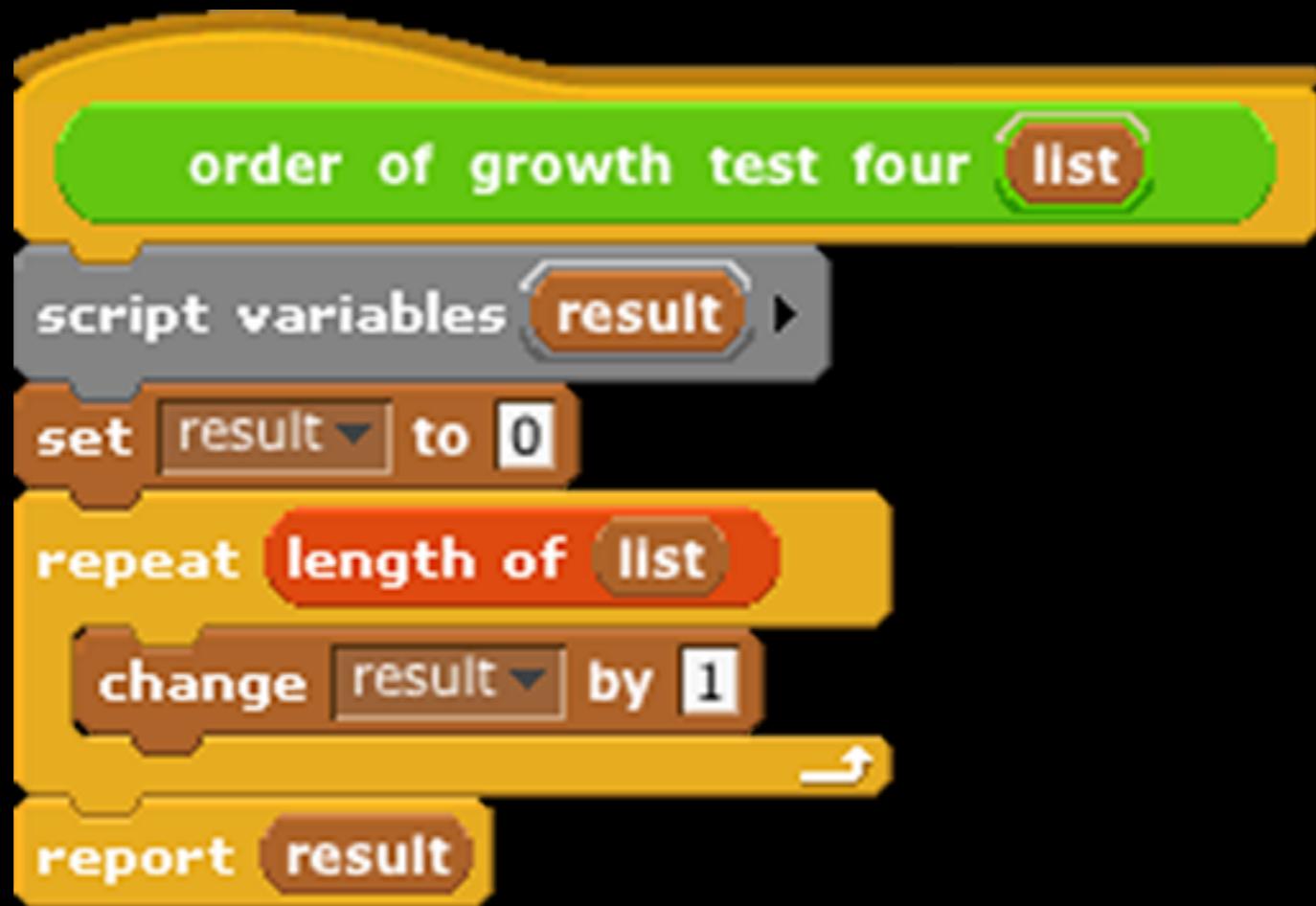
# Algorithms & Complexity



Linear:  $O(n)$

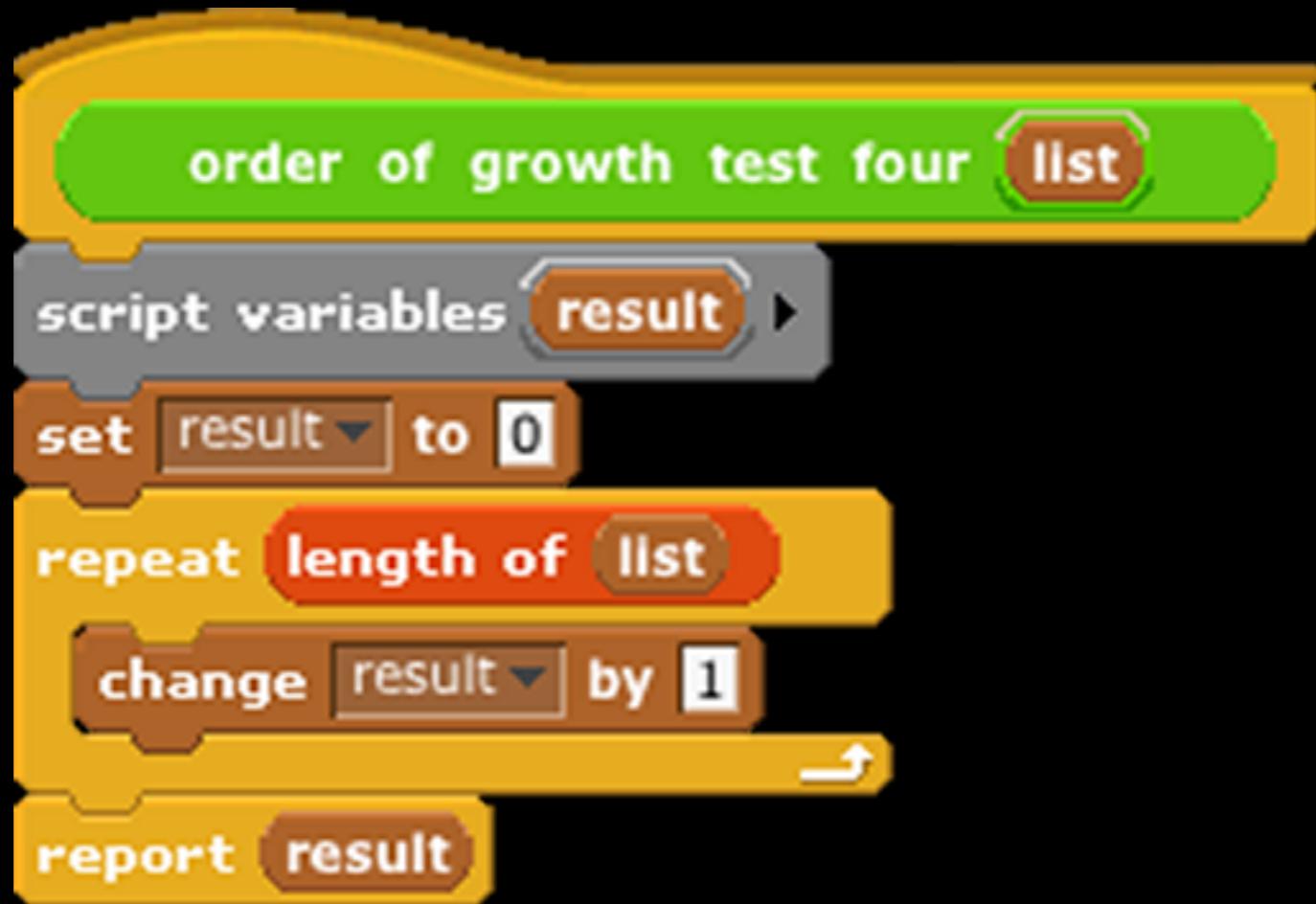
Reason: Number of operations  
proportional to the input size ( $n$ )

# Algorithms & Complexity



Order of growth?

# Algorithms & Complexity



Still Linear:  $O(n)$

Reason: Number of operations  
proportional to the input size (length)

# Algorithms & Complexity



Order of growth?

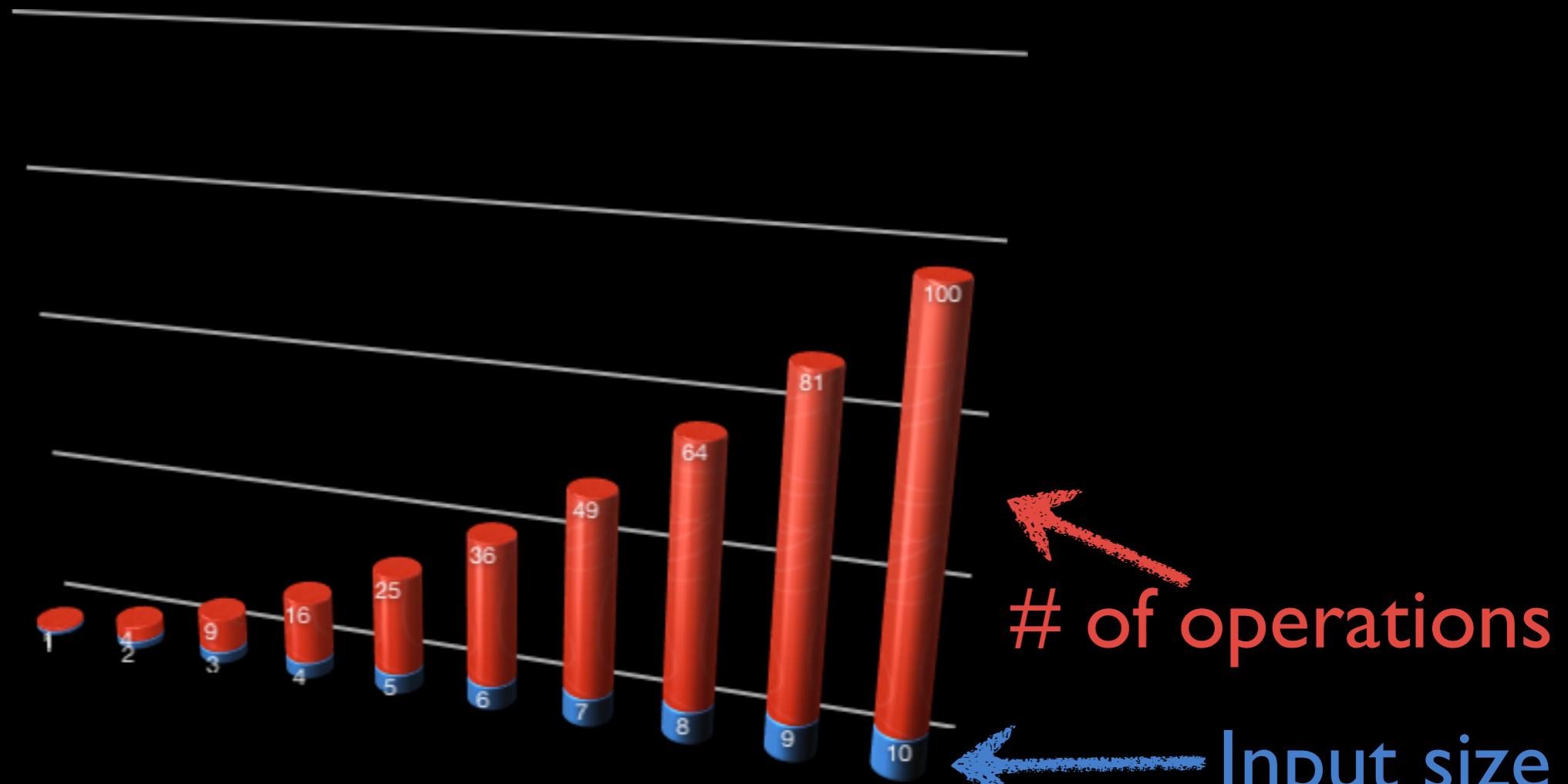
# Algorithms & Complexity



Quadratic:  $O(n^2)$

Reason: Number of operations proportional to the square of the size of the input data ( $n$ )

# Algorithms & Complexity



Quadratic:  $O(n^2)$

# Algorithms & Complexity



Order of growth?

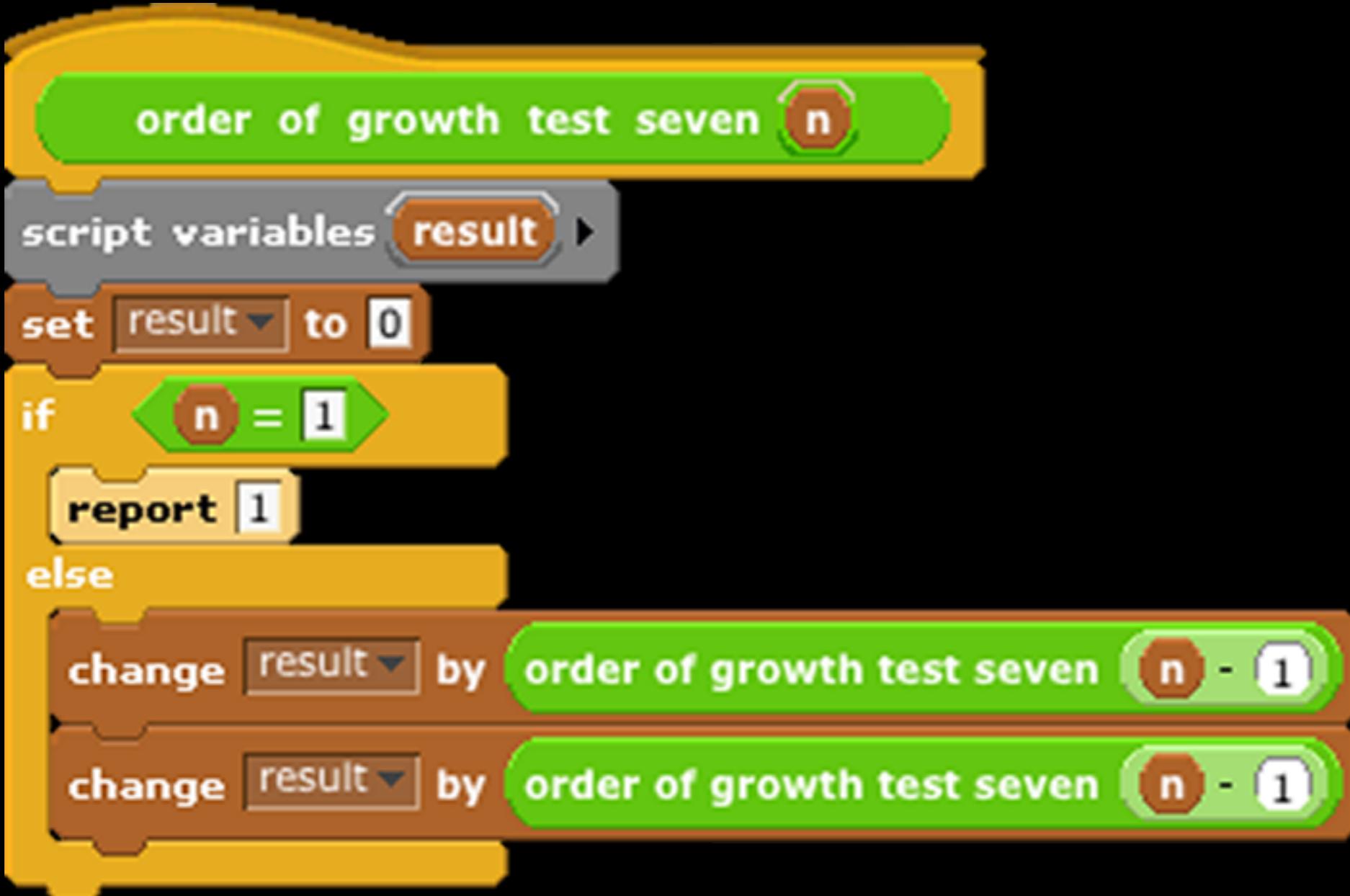
# Algorithms & Complexity



Still Quadratic:  $O(n^2)$

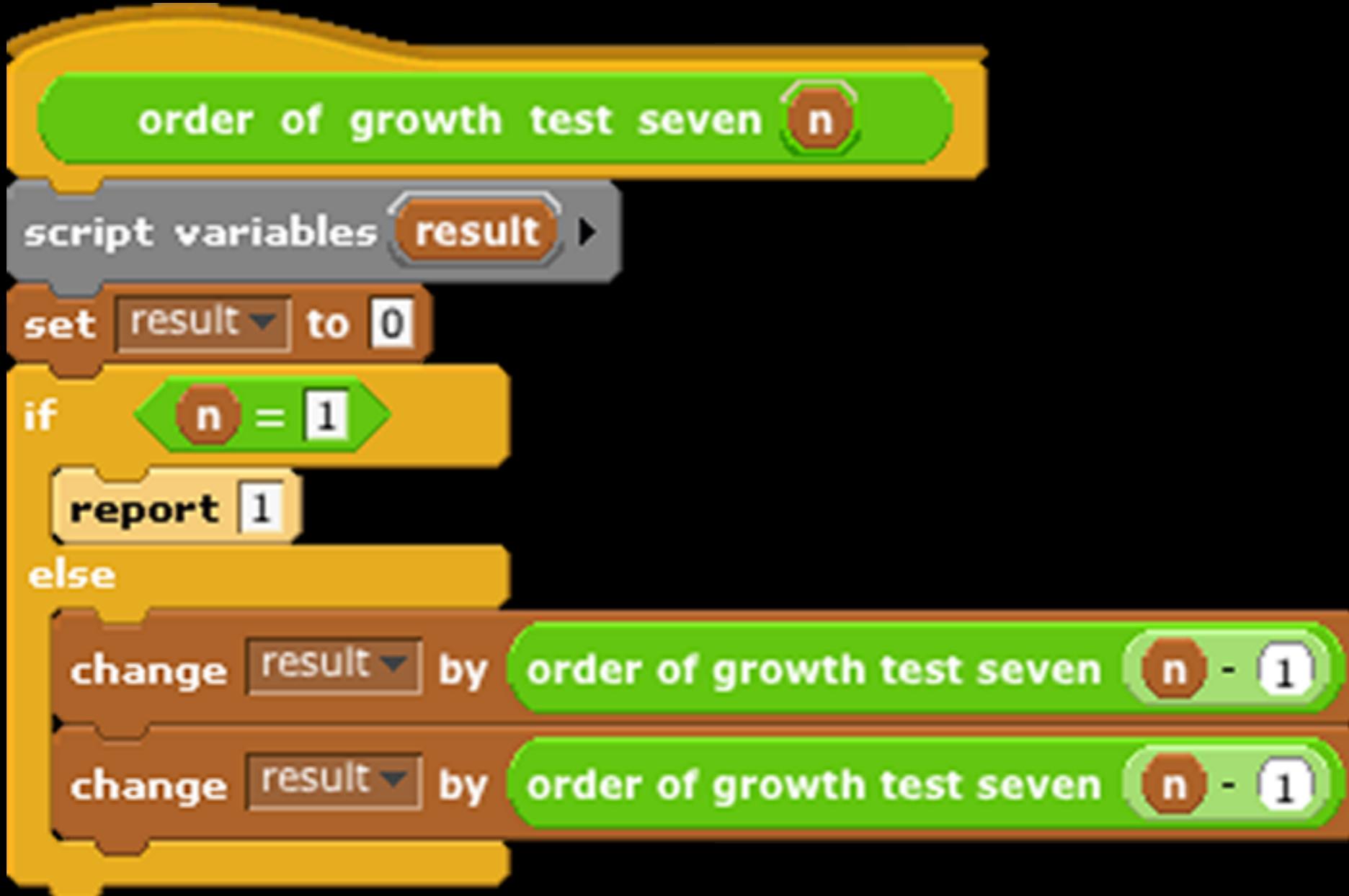
Reason: Number of operations proportional to the square of the size of the input data (length)

# Algorithms & Complexity



Order of growth?

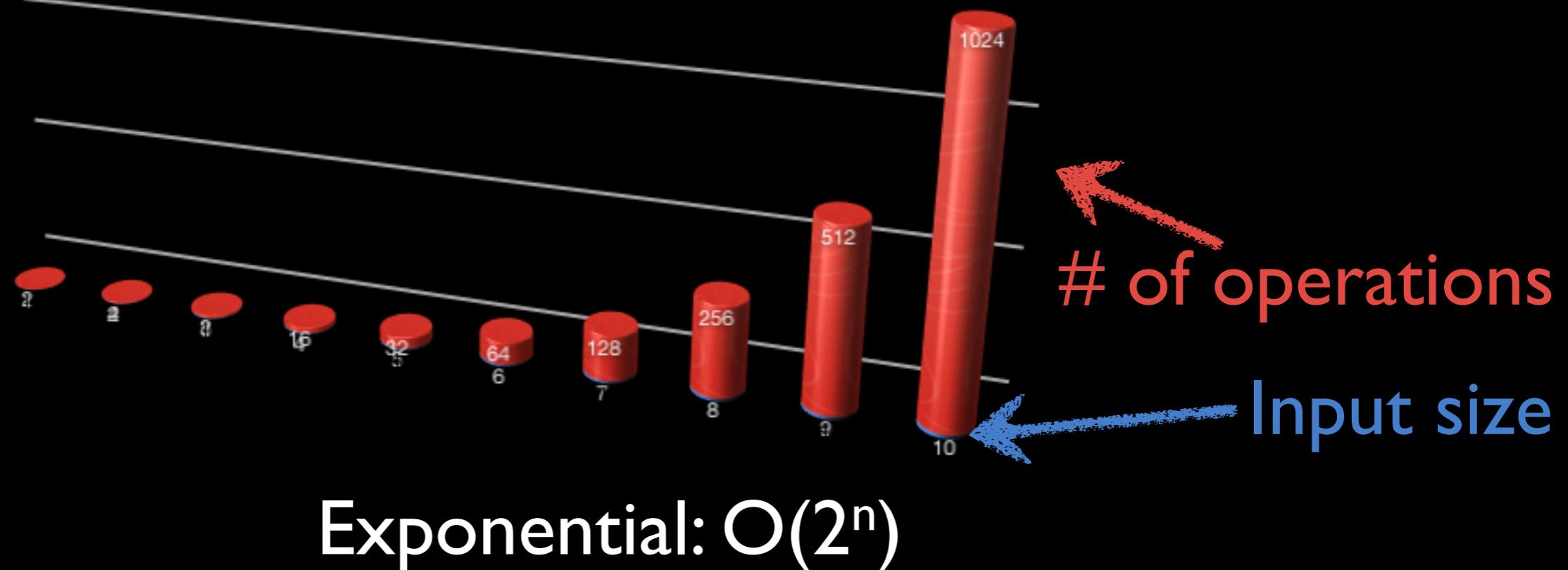
# Algorithms & Complexity



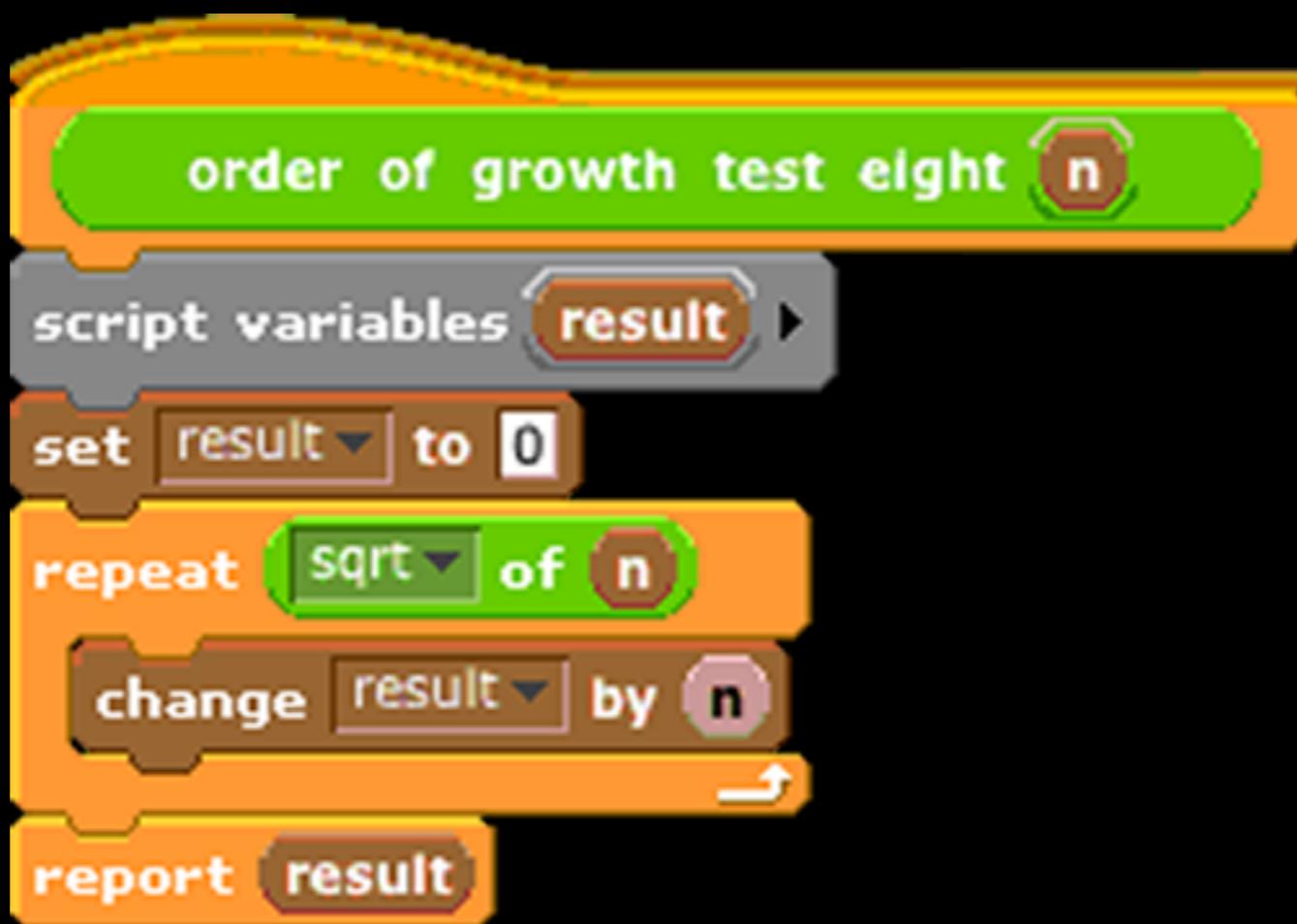
Exponential:  $O(c^n)$

Reason: the recursive call is run twice for each value of n.

# Algorithms & Complexity

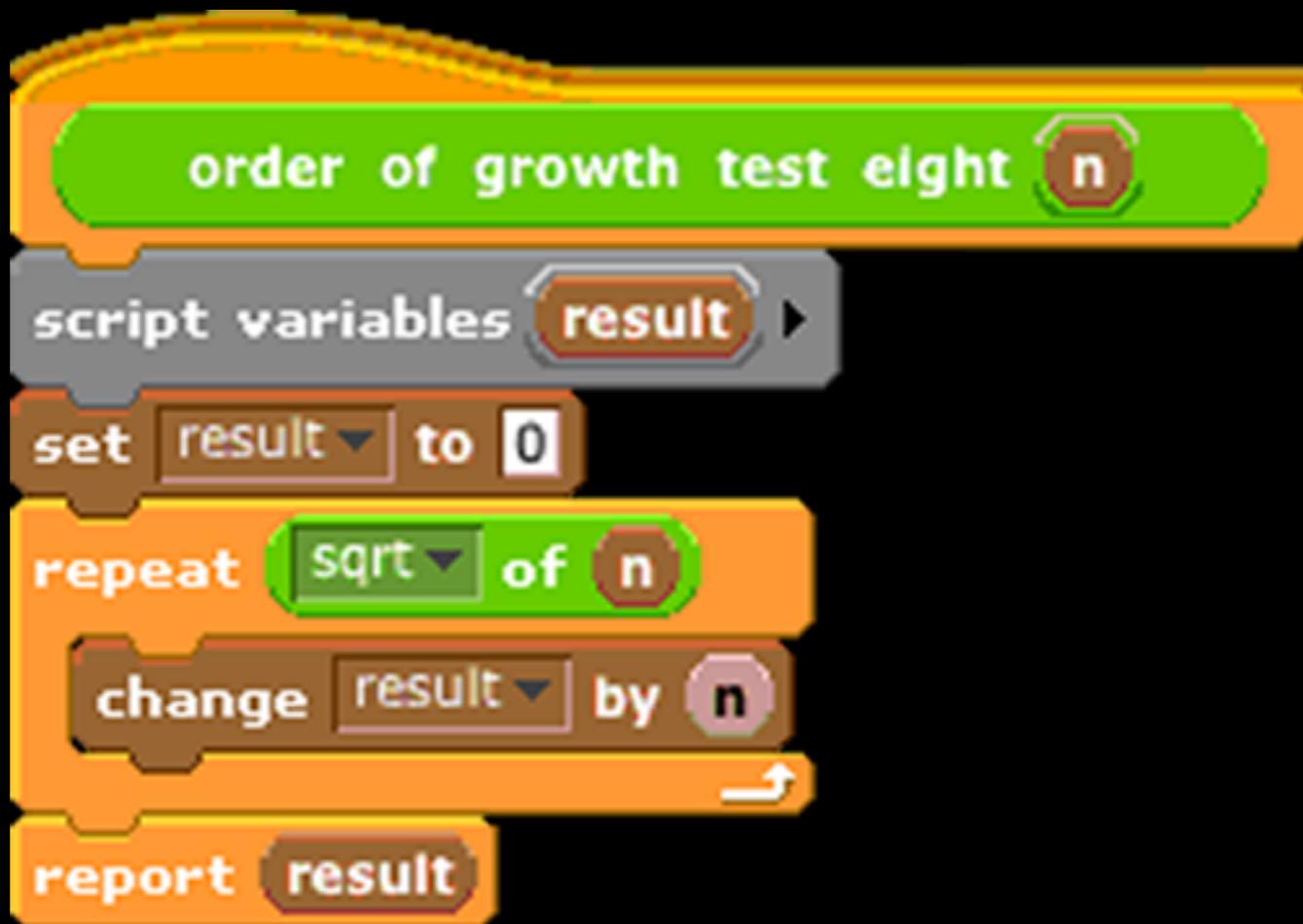


# Algorithms & Complexity



Order of growth?

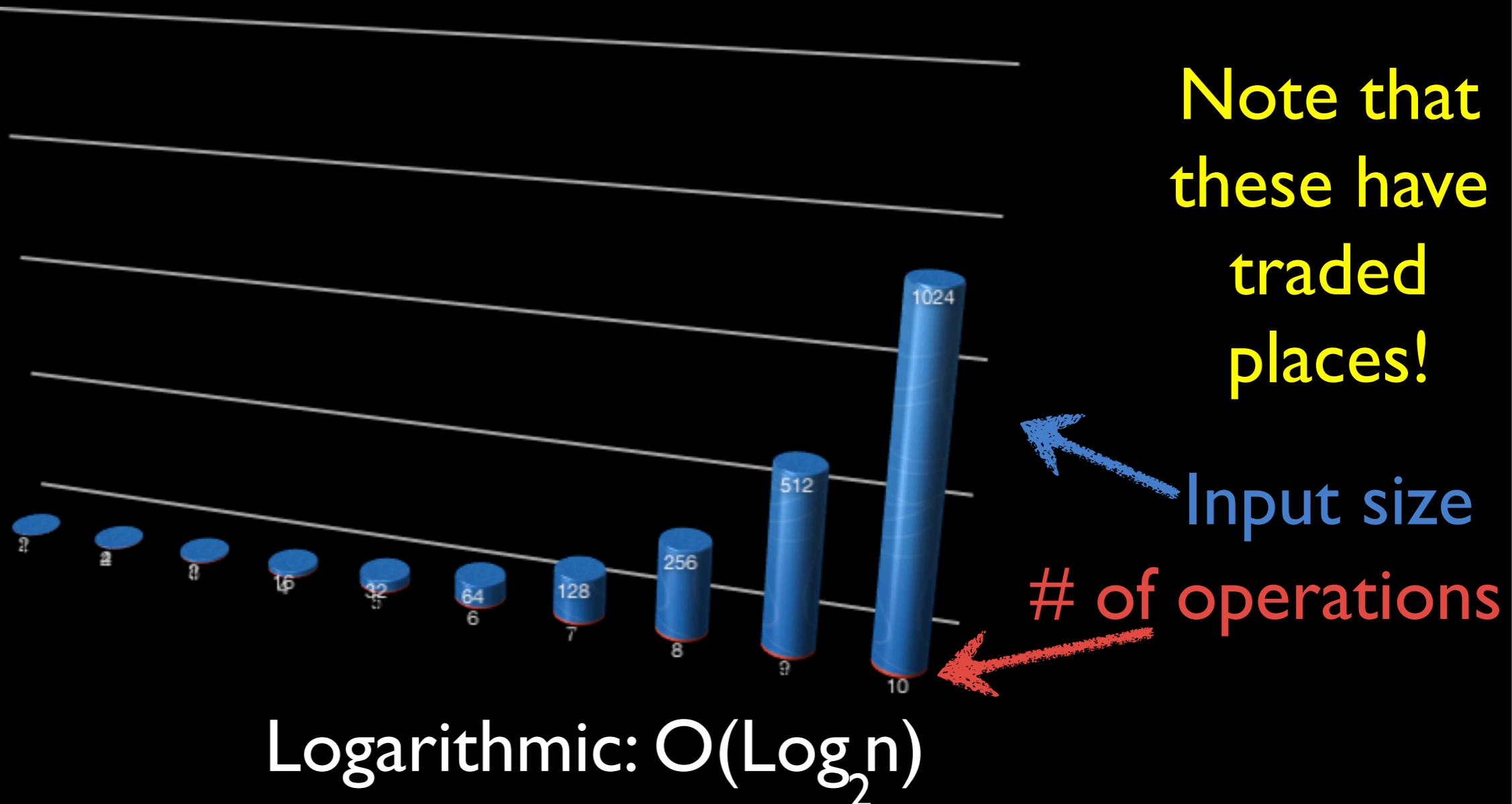
# Algorithms & Complexity



Logarithmic:  $O(\log n)$

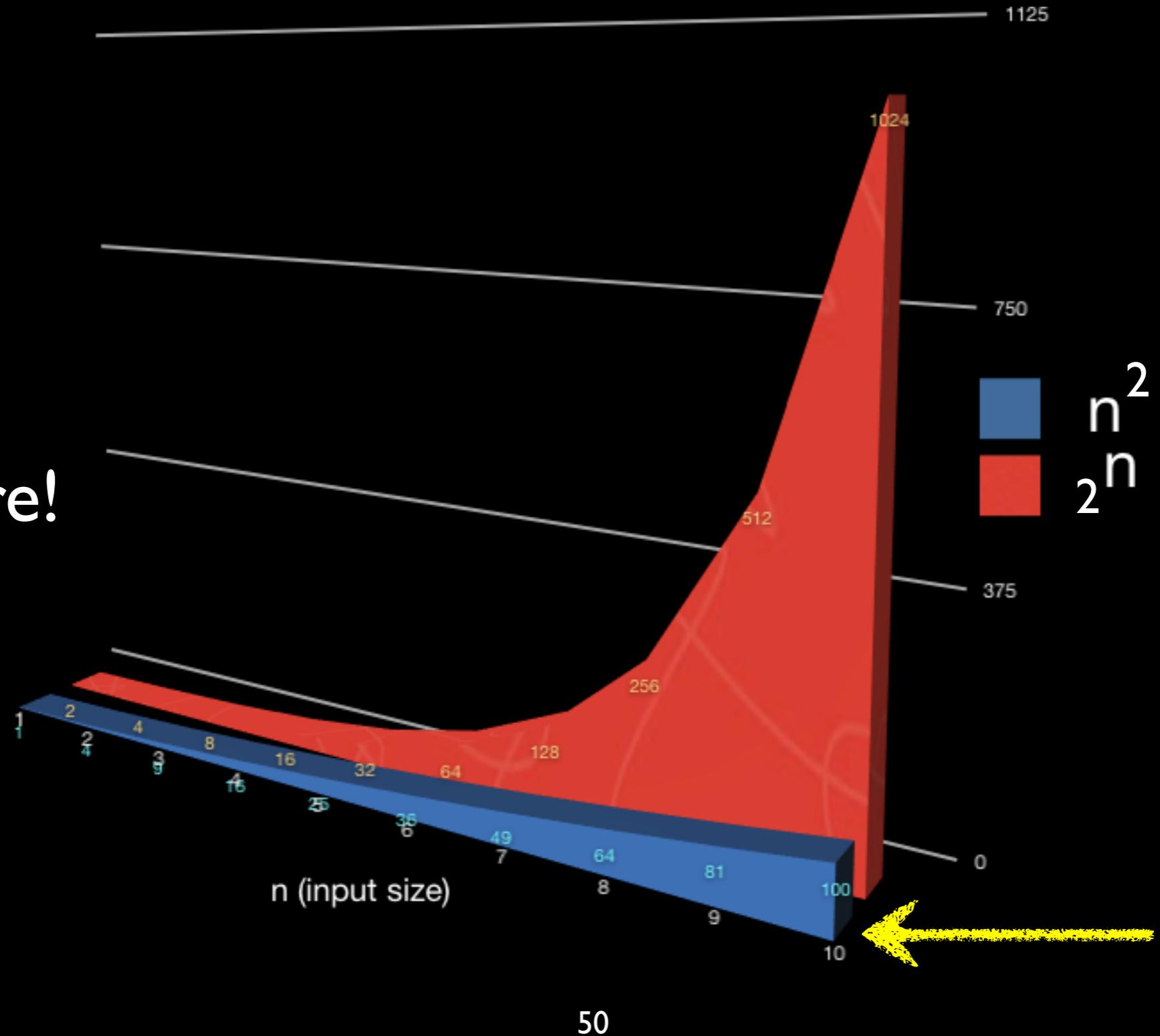
Reason: the number of times the loop runs grows far more slowly (square root) than “n”

# Algorithms & Complexity



# Algorithms & Complexity

Compare!

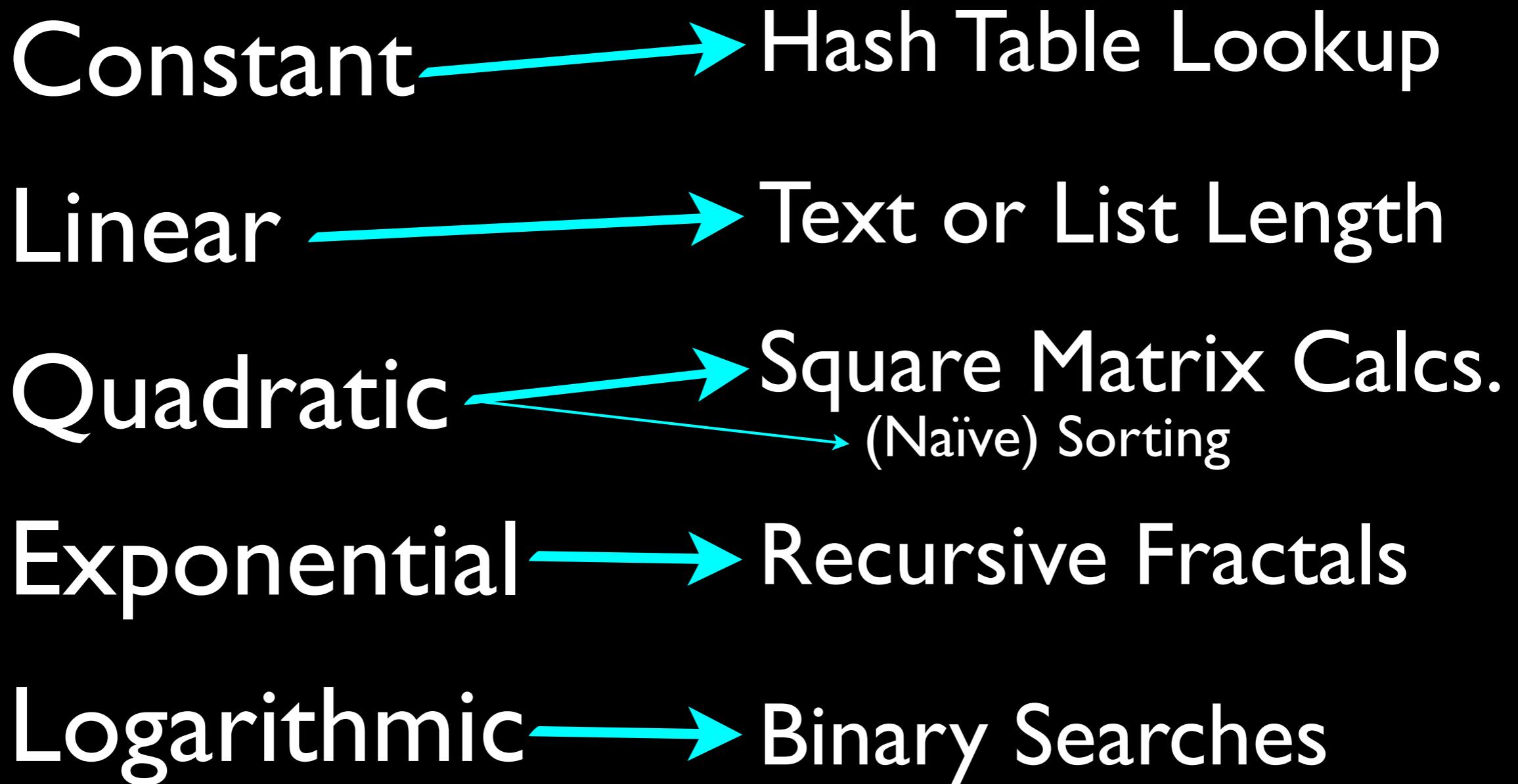


Note that linear growth doesn't even register on this graph!

# Algorithms & Complexity

Consider the number of times  
the calculation *repeats*, rather  
than specific inputs.

# Algorithms & Complexity Examples



# Concurrency

```
when I receive first increment by 1  
set index to pick random 1 to 10  
replace item index of resultingList with  
item index of resultingList + 1
```

```
when I receive second increment by 1  
set index to pick random 1 to 10  
replace item index of resultingList with  
item index of resultingList + 1
```

```
when green flag clicked  
delete all of resultingList  
set index to 1  
repeat 10  
add 0 to resultingList  
repeat 100  
broadcast first increment by 1  
broadcast second increment by 1  
set index to 1  
set result to 0  
repeat 10  
set result to item index of resultingList + result  
change index by 1  
say result for 5 secs
```

List adds up  
to 200?

# Concurrency

```
when I receive first increment by 1
set index to pick random 1 to 10
replace item index of resultingList with
item index of resultingList + 1
```

```
when I receive second increment by 1
set index to pick random 1 to 10
replace item index of resultingList with
item index of resultingList + 1
```

```
when green flag clicked
delete all of resultingList
set index to 1
repeat (10)
  add 0 to resultingList
repeat (100)
  broadcast first increment by 1
  broadcast second increment by 1
  set index to 1
  set result to 0
  repeat (10)
    set result to item index of resultingList + result
    change index by 1
  say result for 5 secs
```

List adds up  
to 200?

No! Why?



# Concurrency

These might choose the same number at the same time!

```
when I receive first increment by 1
set index to pick random 1 to 10
repeat (10)
  replace item (index) of resultingList with
    item (index) of resultingList + 1
```

```
when I receive second increment by 1
set index to pick random 1 to 10
repeat (10)
  replace item (index) of resultingList with
    item (index) of resultingList + 1
```

```
when green flag clicked
delete all of resultingList
set index to 1
repeat (10)
  add 0 to resultingList
```

```
repeat (100)
  broadcast first increment by 1
  broadcast second increment by 1
```

```
set index to 1
set result to 0
```

```
repeat (10)
  set result to item (index) of resultingList + result
  change index by 1
  say result for 5 secs
```

List adds up to 200?

No! Why?

# Concurrency

```
when I receive first increment by 1  
set index to pick random 1 to 10  
replace item index of resultingList with  
item index of resultingList + 1
```

```
when I receive second increment by 1  
set index to pick random 1 to 10  
replace item index of resultingList with  
item index of resultingList + 1
```

```
when green flag clicked  
delete all of resultingList  
set index to 1  
repeat 10  
add 0 to resultingList  
repeat 100  
broadcast first increment by 1  
broadcast second increment by 1  
set index to 1  
set result to 0  
repeat 10  
set result to item index of resultingList + result  
change index by 1  
say result for 5 secs
```

List adds up  
to 200?

Anything else?

# Concurrency

No “wait” here means these might access the same indexes (or not access them at all), by interrupting a broadcast script that is already running!

```
when I receive first increment by 1
set index to pick random 1 to 10
replace item index of resultingList with
item index of resultingList + 1
```

```
when I receive second increment by 1
set index to pick random 1 to 10
replace item index of resultingList with
item index of resultingList + 1
```

```
when green flag clicked
delete all of resultingList
set index to 1
repeat (10)
  add 0 to resultingList
repeat (100)
  broadcast first increment by 1
  broadcast second increment by 1
  set index to 1
  set result to 0
  repeat (10)
    set result to item index of resultingList + result
    change index by 1
    say result for 5 secs
```

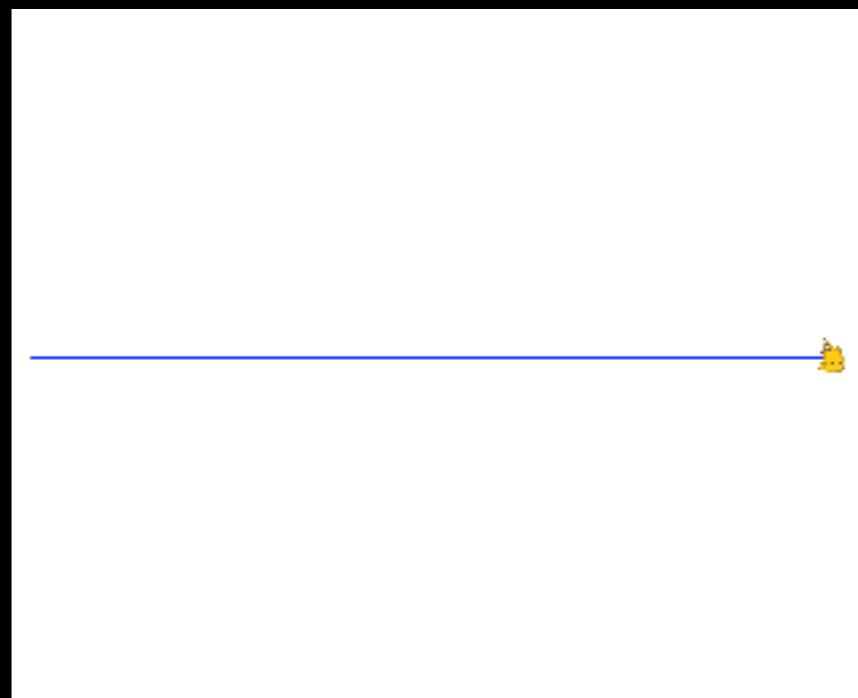
List adds up to 200?

No! Why?

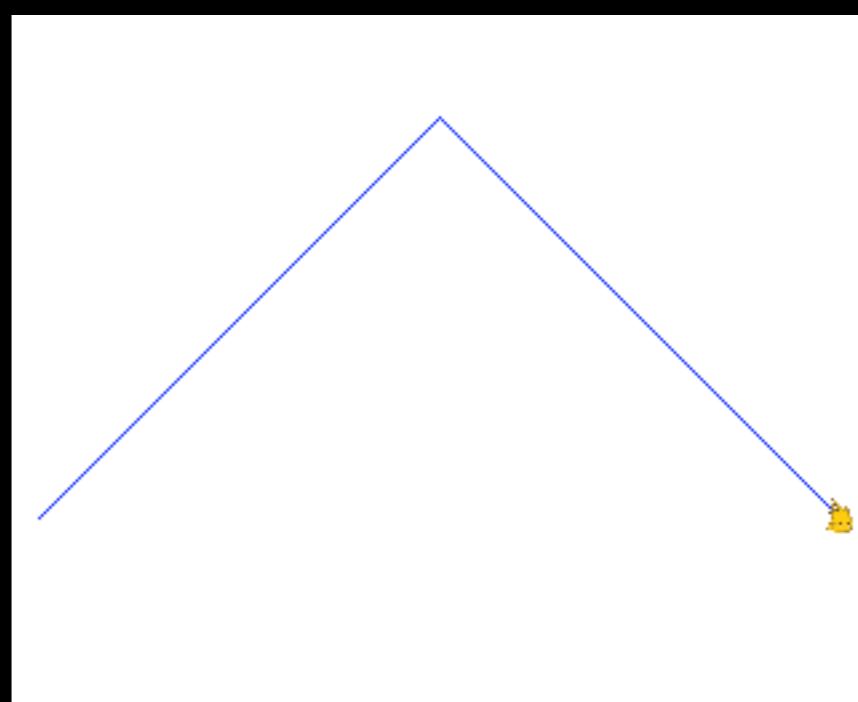


# Recursion

```
fractal [length =300] [n]
if [n = 0]
  pen down
  move [length] steps
  pen up
else
  move [length] steps
  turn [135] degrees
  fractal [length / sqrt of 2] [n - 1]
  turn [90] degrees
  fractal [length / sqrt of 2] [n - 1]
  turn [135] degrees
  move [length] steps
```



n=0



n=1

# Recursion

?

n=2

?

n=4

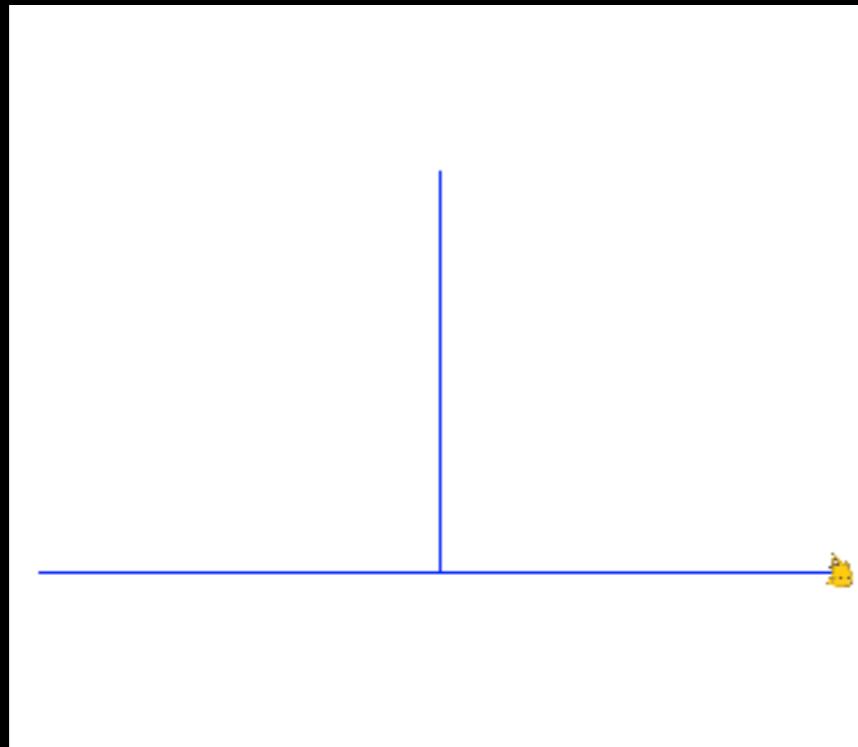
?

n=3

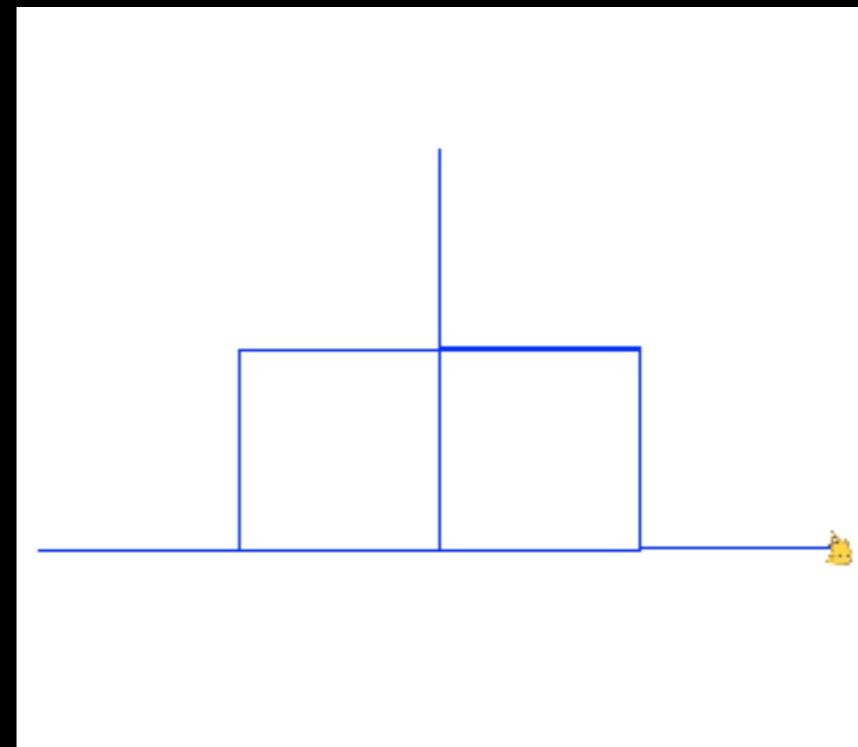
?

n= $\infty$

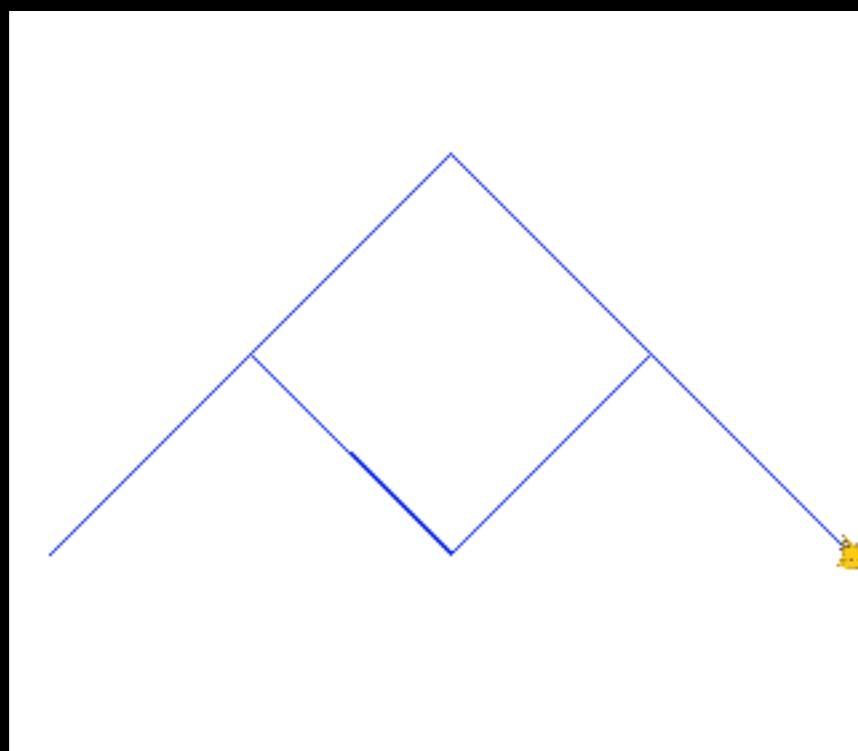
# Recursion



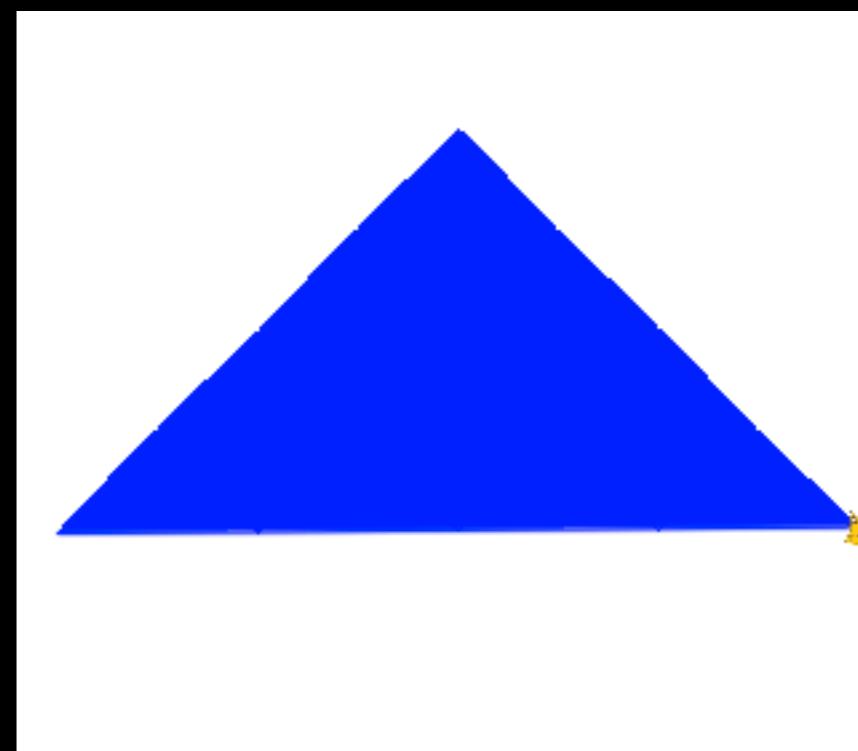
$n=2$



$n=4$



$n=3$



$n=\infty$

# Recursion

An algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task.

- Recursive solutions involve two major parts:
  1. **Base case(s)**, in which the problem is simple enough to be solved directly,
  2. **Recursive case(s)**. A recursive case has three components:
    1. **Divide the problem** into one or more simpler or smaller parts of the problems,
    2. **Invoke the function** (recursively) on each part, and
    3. **Combine the solutions** of the parts into a solution for the problem.
- Depending on the problem, any of these may be trivial or complex.

<http://inst.eecs/~cs31/sp09/lectures/L06/2009SpCS3L06.html>



# Data Structure : Hash Tables

Can we use “MapReduce” to  
build a hash table?

(we'll come back to that...)

# Lambdas & HOFs

- What is a Lambda?
- What is a Higher-Order Function (HOF)?

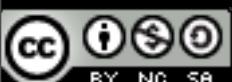
# Lambdas & HOFs

- A “First Class” Procedure
- “Function as Data”

# Lambdas & HOFs

- What is a Higher-Order Function (HOF)?
  - In mathematics and computer science, **higher-order functions, functional forms, or functionals** are functions which do *at least one* of the following:
    - **take one or more functions as an input**
    - **output a function**

[http://en.wikipedia.org/wiki/Higher-order\\_function](http://en.wikipedia.org/wiki/Higher-order_function)



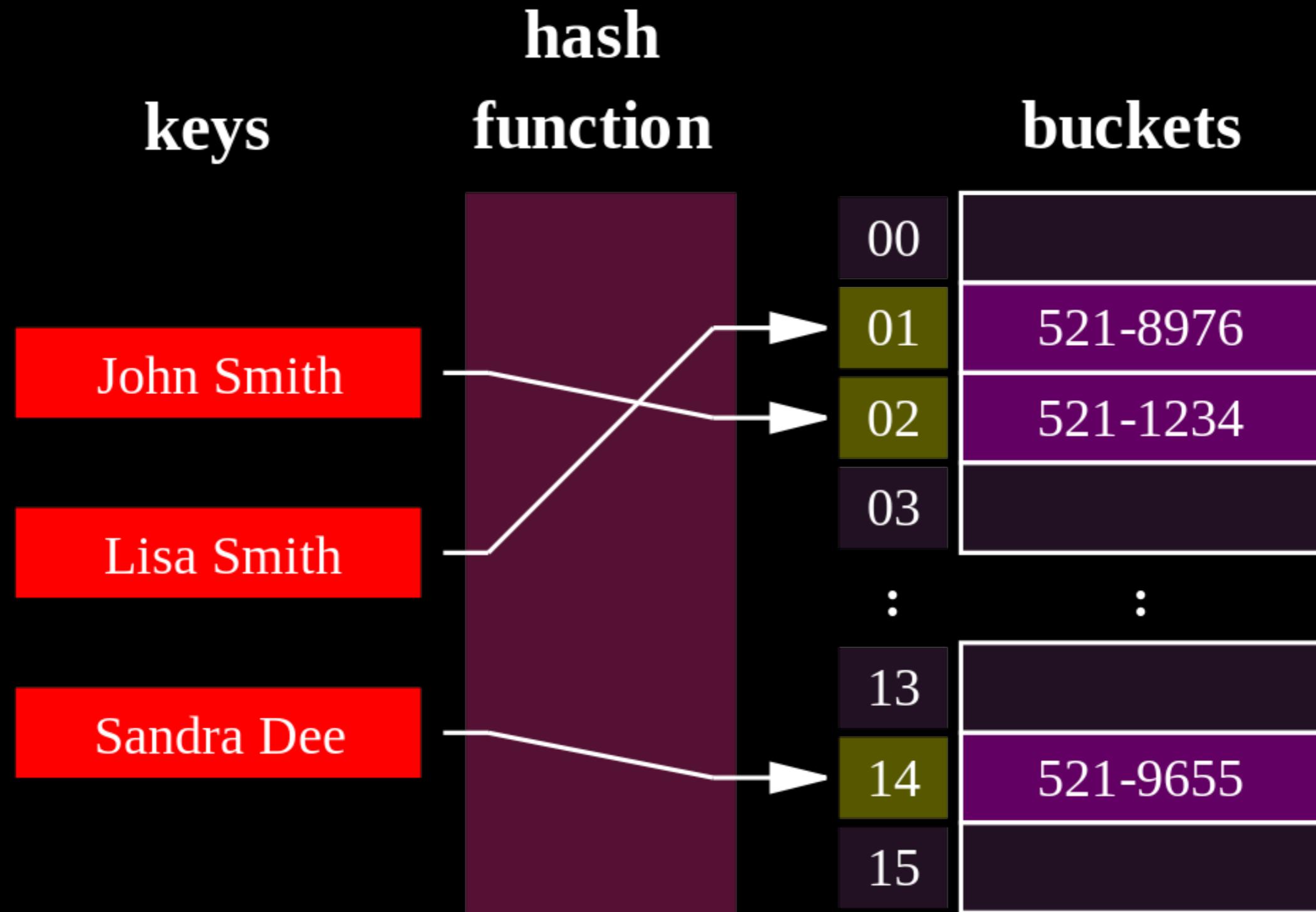
# Lambdas & HOFs

- Useful HOFs (you can build your own!)
  - map Reporter over List
    - Report a new list, every element  $E$  of  $List$  becoming  $\text{Reporter}(E)$
  - keep items such that Predicate from List
    - Report a new list, keeping only elements  $E$  of  $List$  if  $\text{Predicate}(E)$
  - combine with Reporter over List
    - Combine all the elements of  $List$  with  $\text{Reporter}(E)$
    - This is also known as “reduce”
- Acronym example
  - keep → map → combine

<http://inst.eecs.berkeley.edu/~cs10/sp11/lec/17/src/2011-03-30-CS10-L17-DG-HOF-l.pptx>



# Data Structure : Hash Tables



# Data Structure : Hash Tables

- Determine size of hash table.
- Hashing function (algorithm) to generate keys (index) from values (items).
- Modulo key (index) by hash table size.
- Store key and value in hash table at the resulting index.
- Find key in table using above algorithms (reading value instead of storing it).

# Data Structure : HOF/Hash Tables

Can we use “MapReduce”  
to build a hash table?

# Data Structure : HOF/Hash Tables

Yes!

- Try to program one on your own... it's *great* practice with HOFs, Lists, etc.!
- I will post the solution to Piazzza...