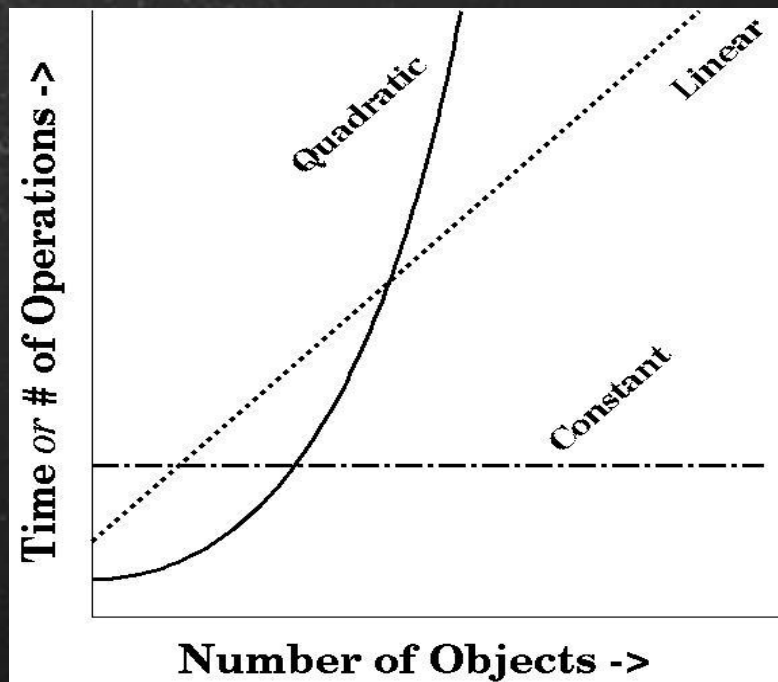


# Algorithmic Complexity



# Algorithmic Complexity

"**Algorithmic Complexity**", also called "**Running Time**" or "**Order of Growth**", refers to the number of steps a program takes as a function of the size of its inputs. In this class, we will assume the function only has one input, which we will say has length  $n$ .



# Algorithmic Complexity

## Notes on Notation:

Algorithmic complexity is usually expressed in 1 of 2 ways. The first is the way used in lecture - "logarithmic", "linear", etc. The other is called **Big-O notation**. This is a more mathematical way of expressing running time, and looks more like a function. For example, a "linear" running time can also be expressed as  $O(n)$ . Similarly, a "logarithmic" running time can be expressed as  $O(\log n)$ .

# Algorithmic Complexity

Here is a list of some common running times:

Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$

We will talk about each briefly.



# Constant-Time Algorithms - $O(1)$

A **constant-time algorithm** is one that takes the same amount of time, regardless of its input. Here are some examples:

- Given two numbers\*, report the sum
- Given a list, report the first element
- Given a list of numbers\*, report the result of adding the first element to itself 1,000,000 times

Why is the last example still constant time?

\*Here, we are referring to numbers of a set maximum size (i.e. 32-bit numbers, 64-bit numbers, etc.)

# Logarithmic-Time Algorithm - $O(\log n)$

A **logarithmic-time algorithm** is one that requires a number of steps proportional to the  $\log(n)$ . In most cases, we use 2 as the base of the log, but it doesn't matter which base because we ignore constants. Because we use the base 2, we can rephrase this in the following way: *every time the **size of the input doubles**, our algorithm performs **one more step***. Examples:

- Binary search
- Searching a tree data structure (we'll see what this is later)

# Linear-Time Algorithms - $O(n)$

A **linear-time algorithm** is one that takes a number of steps directly proportional to the size of the input. In other words, if the size of the **input doubles**, the number of **steps doubles**. Examples:

- Given a list of words, say each item of a list
- Given a list of numbers, add each pair of numbers together (item 1 + item 2, item 3 + item 4, etc.)
- Given a list of numbers, multiply every 3rd number by 2

Again, why is the last algorithm still linear?

# Quadratic-Time Algorithms - $O(n^2)$

A **quadratic-time algorithm** is one that takes a number of steps proportional to  $n^2$ . That is, if the size of the **input doubles**, the number of **steps quadruples**. A typical pattern of quadratic-time algorithms is performing a linear-time operation on each item of the input ( $n$  steps per item \*  $n$  items =  $n^2$  steps). Examples:

- Compare each item of a list against all the other items in the list
- Fill in a  $n$ -by- $n$  game board



# Cubic-Time Algorithms - $O(n^3)$

A **cubic-time algorithm** is one that takes a number of steps proportional to  $n^3$ . In other words, if the **input doubles**, the number of **steps is multiplied by 8**. Similarly to the quadratic case, this could be the result of applying an  $n^2$  algorithm to  $n$  items, or applying a linear algorithm to  $n^2$  items. Examples:

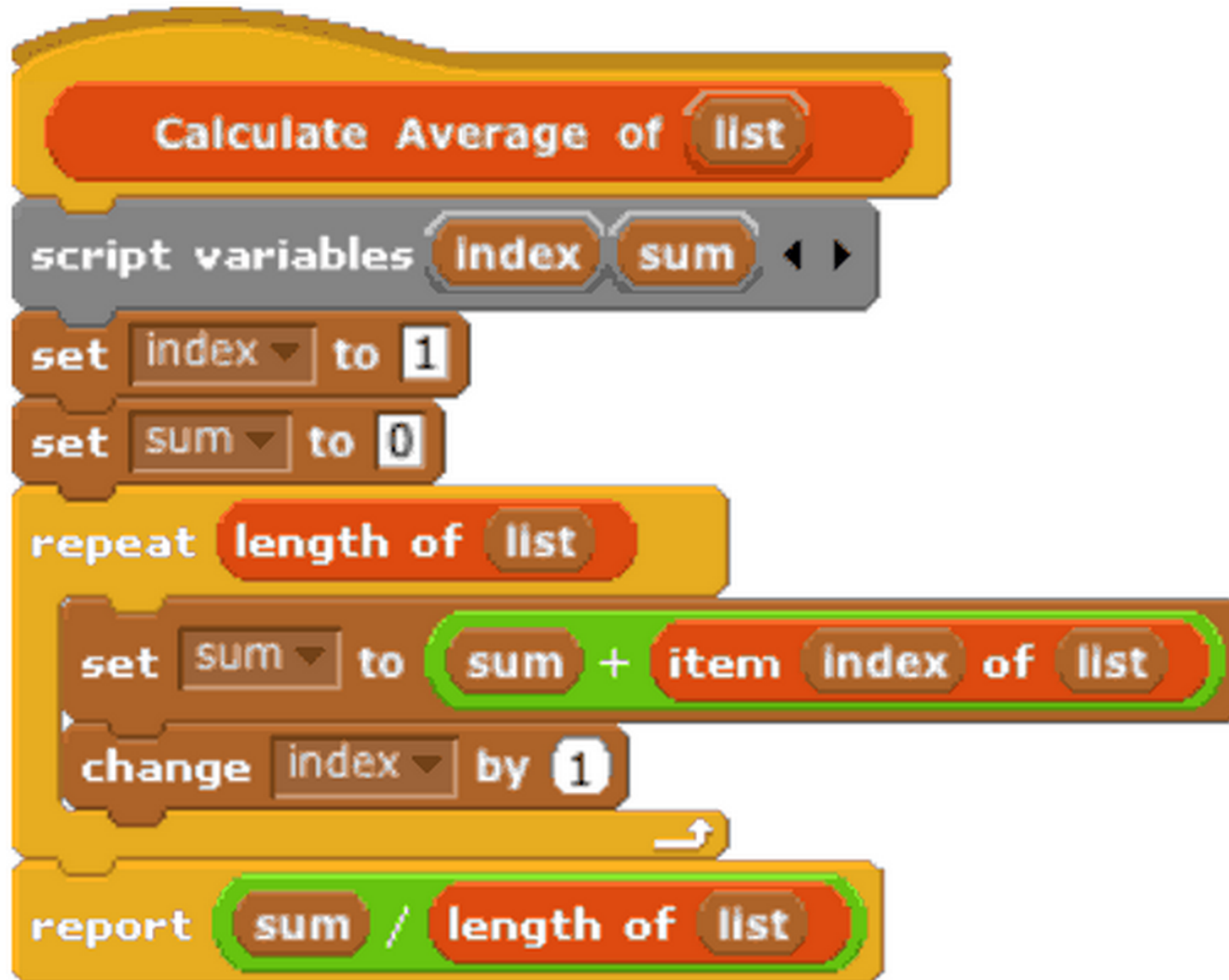
- Fill in a 3D board (or environment)
- For each object in a list, construct an  $n$ -by- $n$  bitmap drawing of the object

# Exponential-Time Algorithms - $O(2^n)$

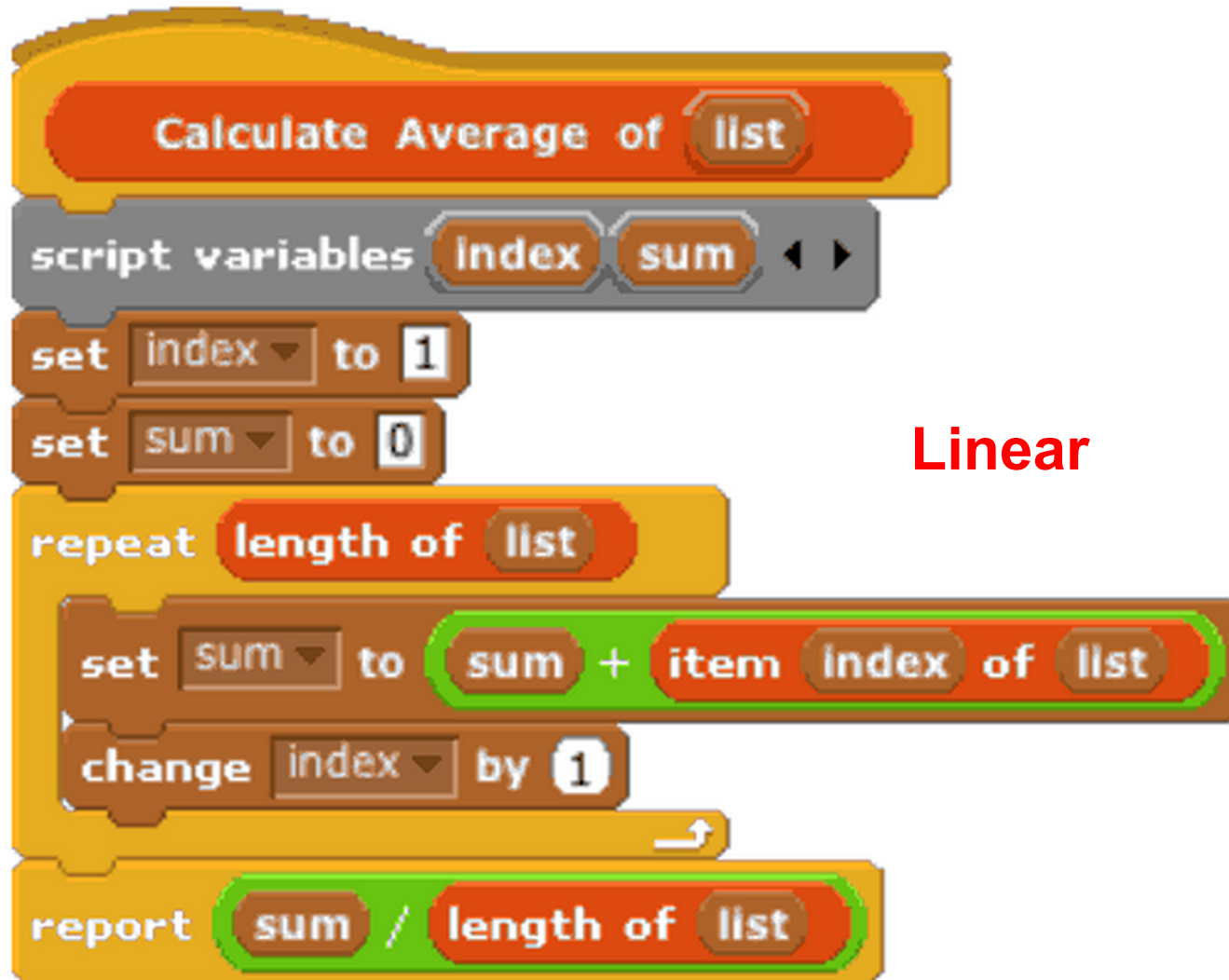
An **exponential-time algorithm** is one that takes time proportional to  $2^n$ . In other words, if the size of the **input increases by one**, the number of **steps doubles**. Note that logarithms and exponents are inverses of each other. Algorithms in this category are often considered too slow to be practical, especially if the input is typically large. Examples:

- Given a number  $n$ , generate a list of every  $n$ -bit binary number

# What is the runtime?

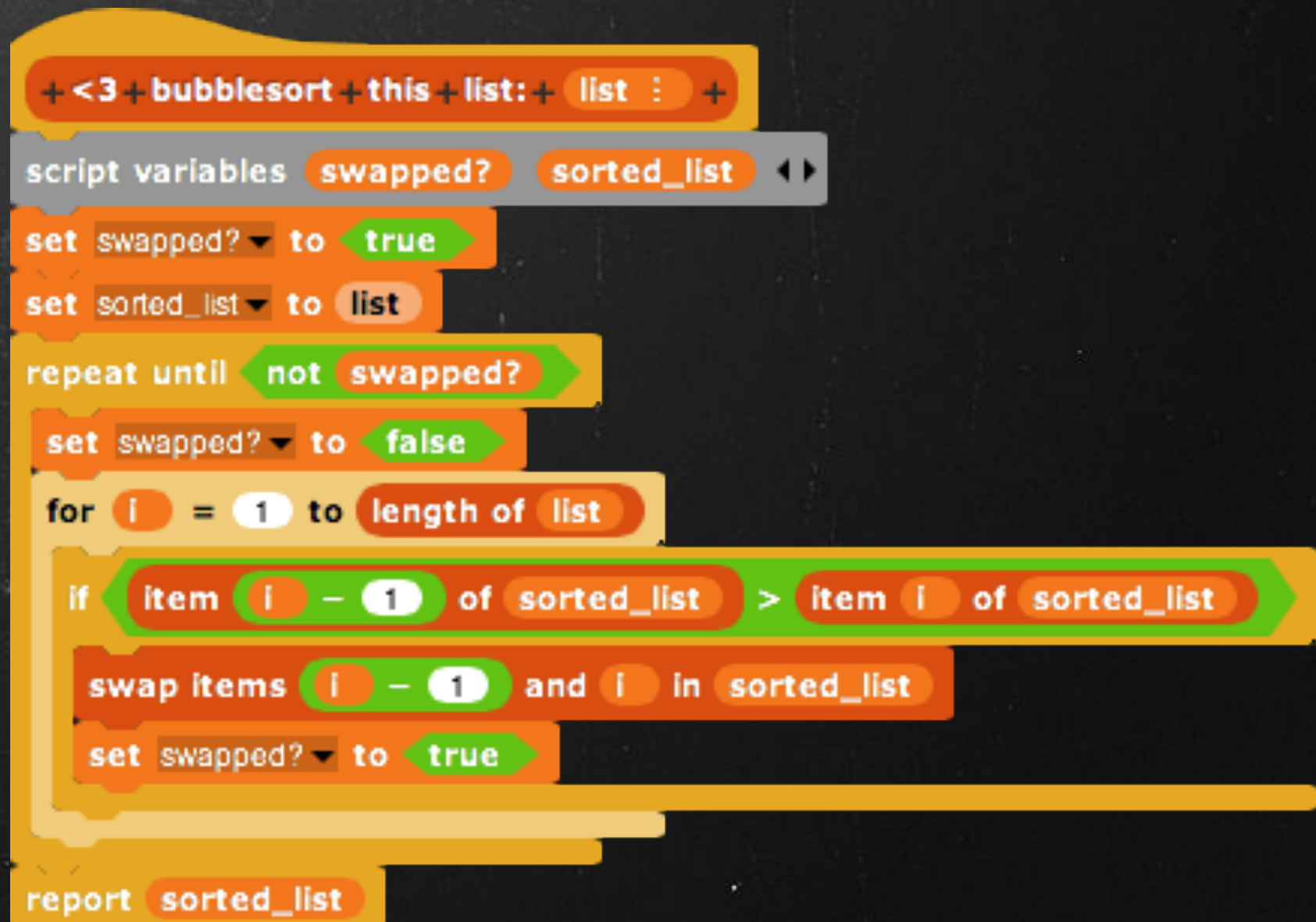


# What is the runtime?





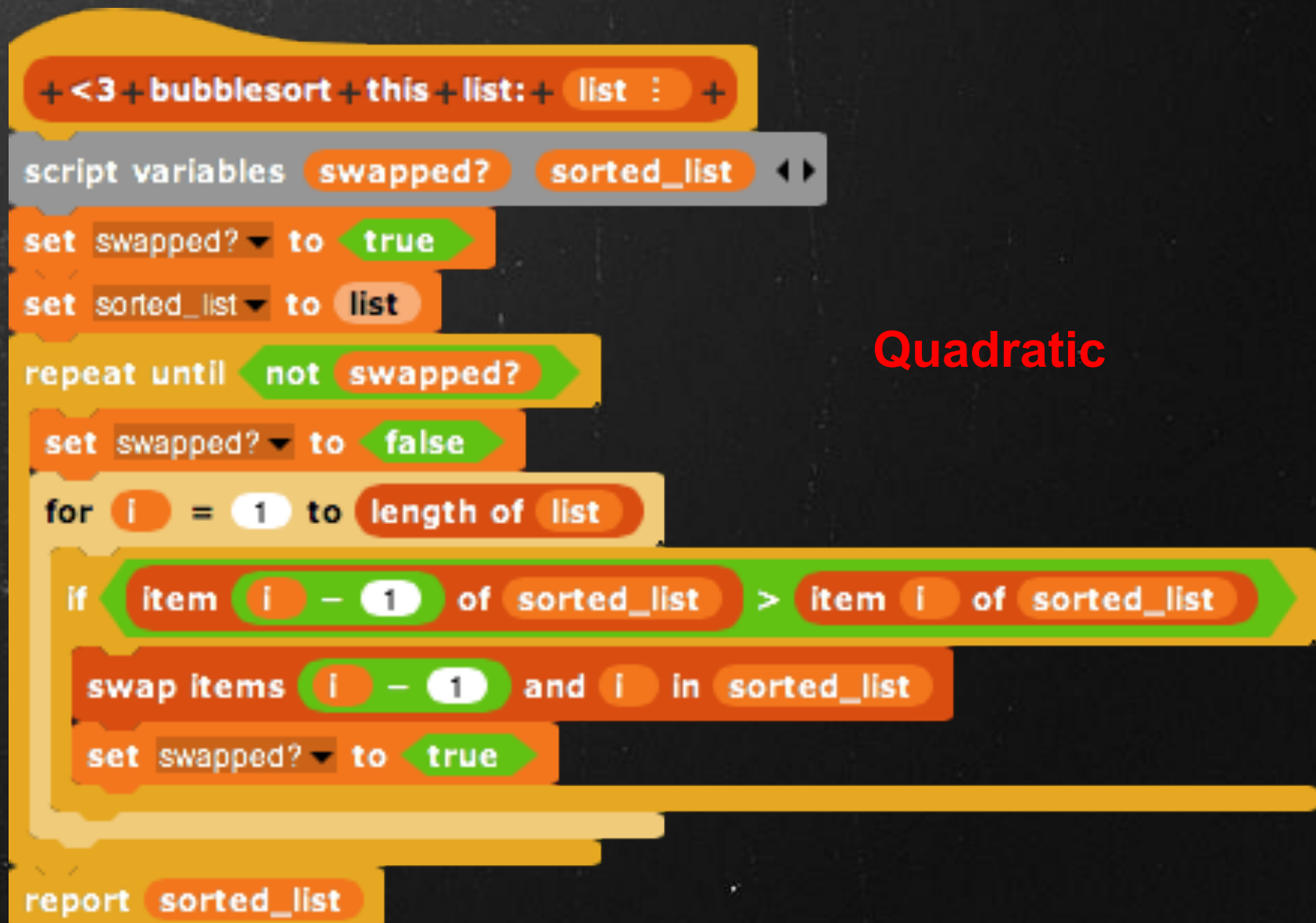
# What is the runtime?



```
+<3+ bubblesort + this + list: + list : +
script variables swapped? sorted_list <>
set swapped? to true
set sorted_list to list
repeat until not swapped?
  set swapped? to false
  for i = 1 to length of list
    if item i - 1 of sorted_list > item i of sorted_list
      swap items i - 1 and i in sorted_list
      set swapped? to true
  end
end
report sorted_list
```

The image shows a Scratch script for a bubble sort algorithm. The script is written in a block-based language and is contained within a script area. It starts with a comment line: `+<3+ bubblesort + this + list: + list : +`. The script then declares two variables: `swapped?` and `sorted_list`. It sets `swapped?` to `true` and `sorted_list` to `list`. The main loop is a `repeat until not swapped?` loop. Inside this loop, it sets `swapped?` to `false` and enters a `for i = 1 to length of list` loop. Inside the `for` loop, it checks if `item i - 1 of sorted_list > item i of sorted_list`. If true, it swaps the items at indices `i - 1` and `i` in `sorted_list` and sets `swapped?` to `true`. After the `for` loop, it ends the `repeat` loop and reports the value of `sorted_list`.

# What is the runtime?



```
+<3+ bubblesort + this + list: + list : +
script variables swapped? sorted_list <>
set swapped? to true
set sorted_list to list
repeat until not swapped?
  set swapped? to false
  for i = 1 to length of list
    if item i - 1 of sorted_list > item i of sorted_list
      swap items i - 1 and i in sorted_list
      set swapped? to true
  report sorted_list
```

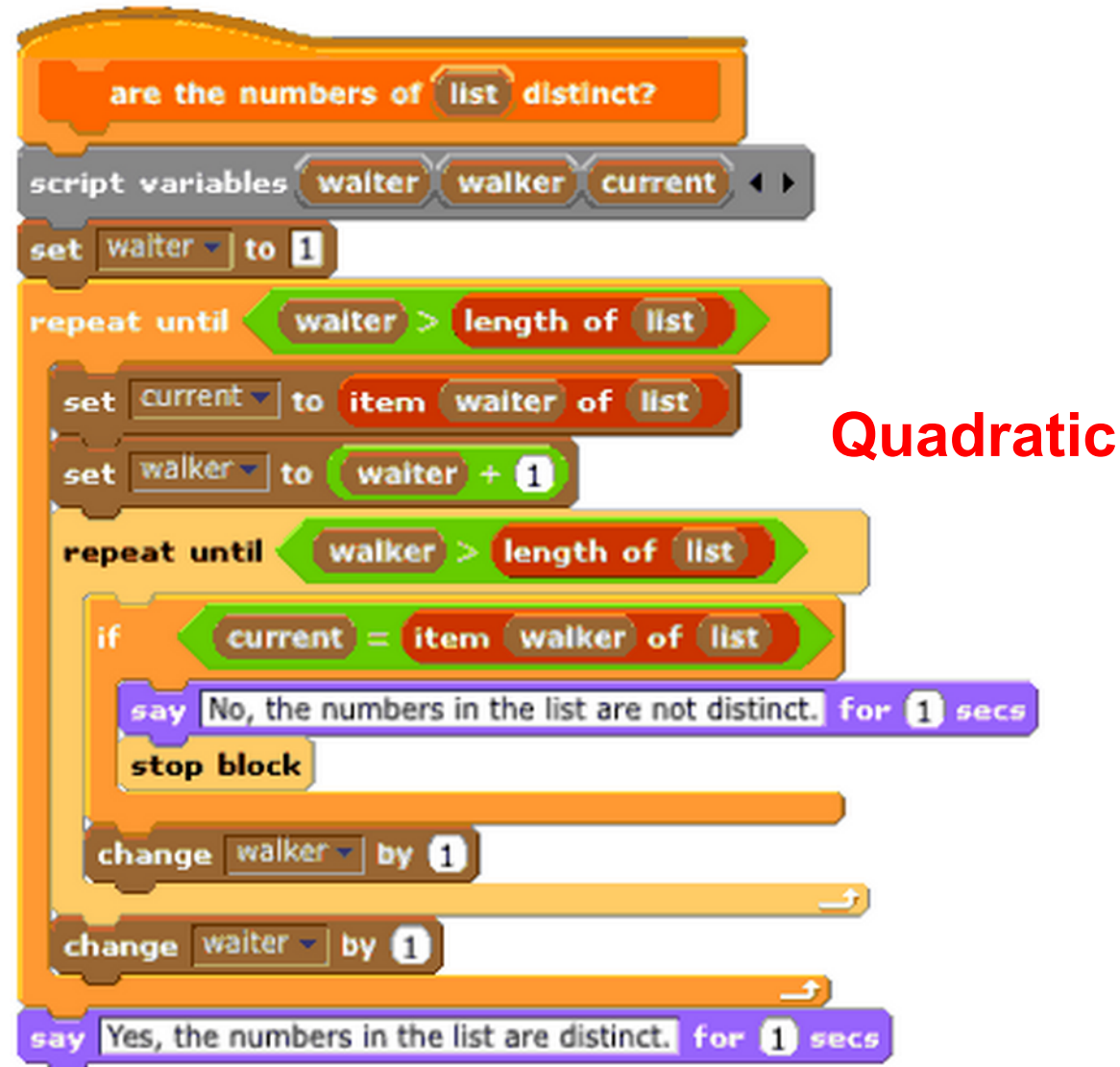
The image shows a Scratch script for a bubble sort algorithm. The script starts with a comment '+<3+ bubblesort + this + list: + list : +'. It then declares two variables: 'swapped?' and 'sorted\_list'. The 'swapped?' variable is set to 'true' and 'sorted\_list' is set to 'list'. A 'repeat until' loop is used with the condition 'not swapped?'. Inside this loop, 'swapped?' is set to 'false'. A 'for' loop iterates from 'i = 1' to 'length of list'. Inside the 'for' loop, an 'if' statement checks if 'item i - 1 of sorted\_list' is greater than 'item i of sorted\_list'. If true, it swaps the items and sets 'swapped?' to 'true'. After the 'for' loop, the script reports 'sorted\_list'.

Quadratic

# What is the runtime?



# What is the runtime?

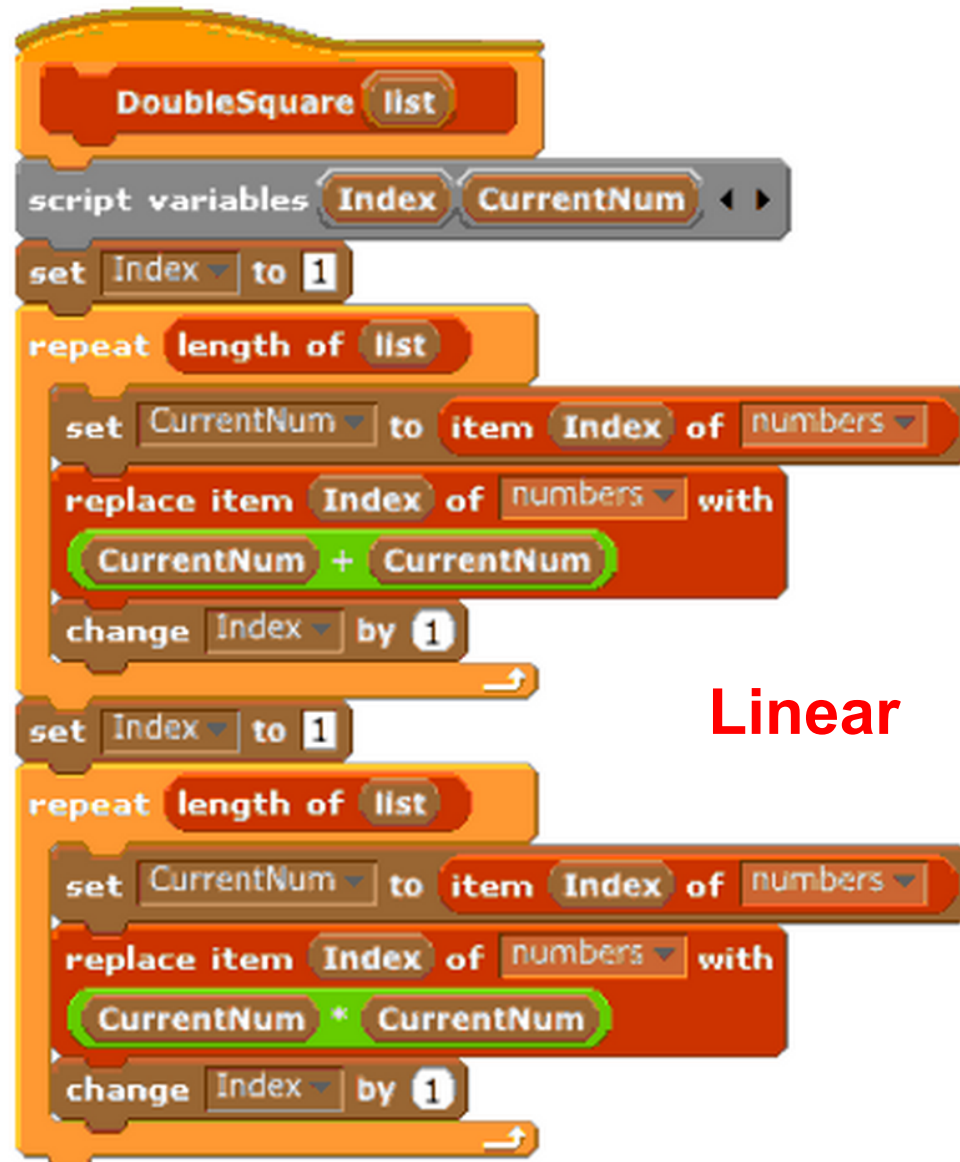




# What is the runtime?



# What is the runtime?

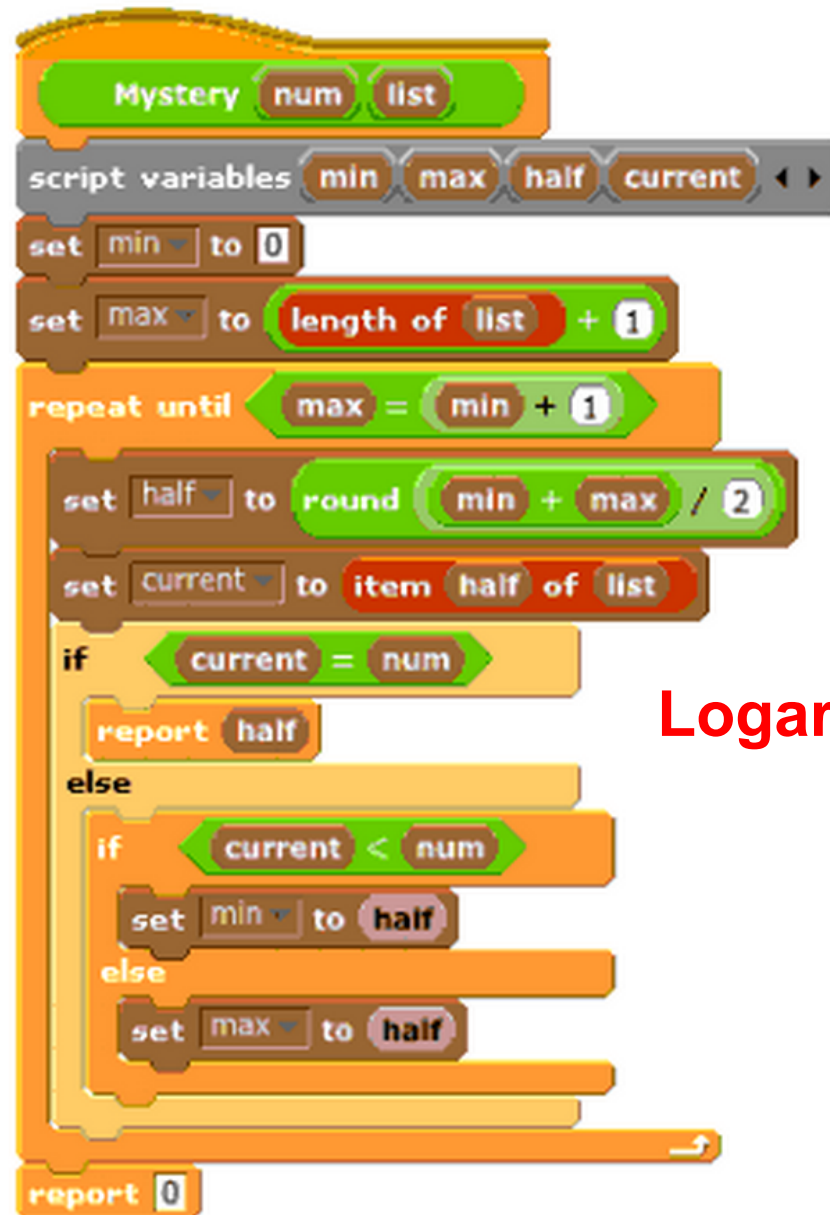


Linear

# What is the runtime?



# What is the runtime?

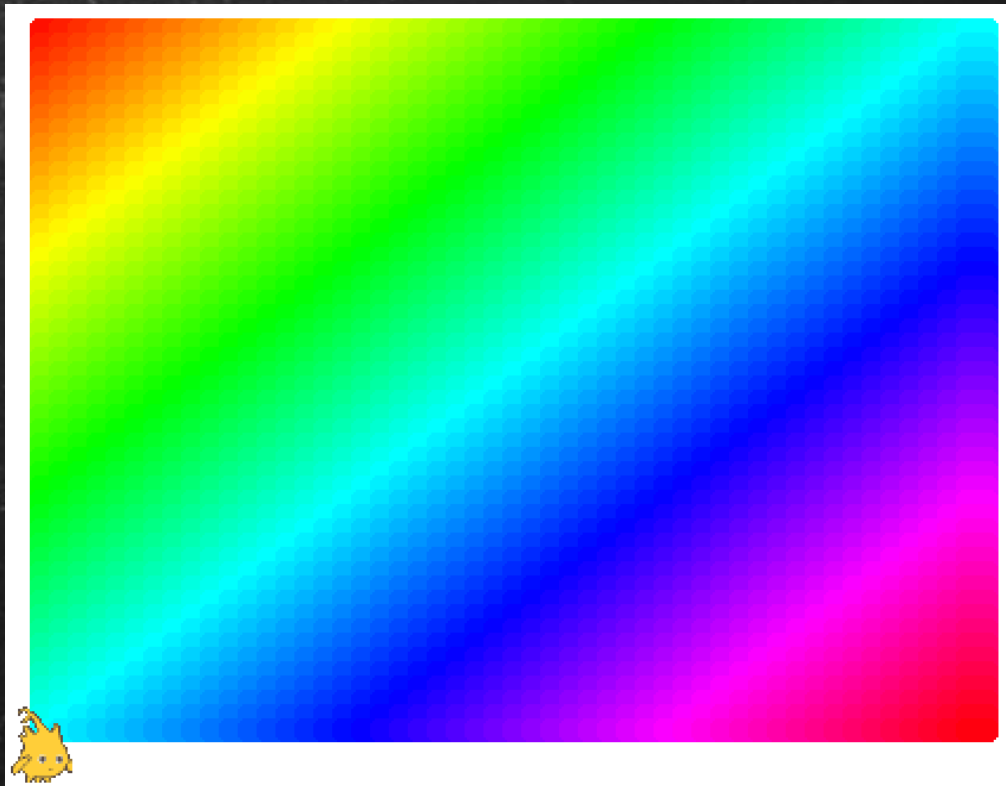


Logarithmic



# What is the runtime?

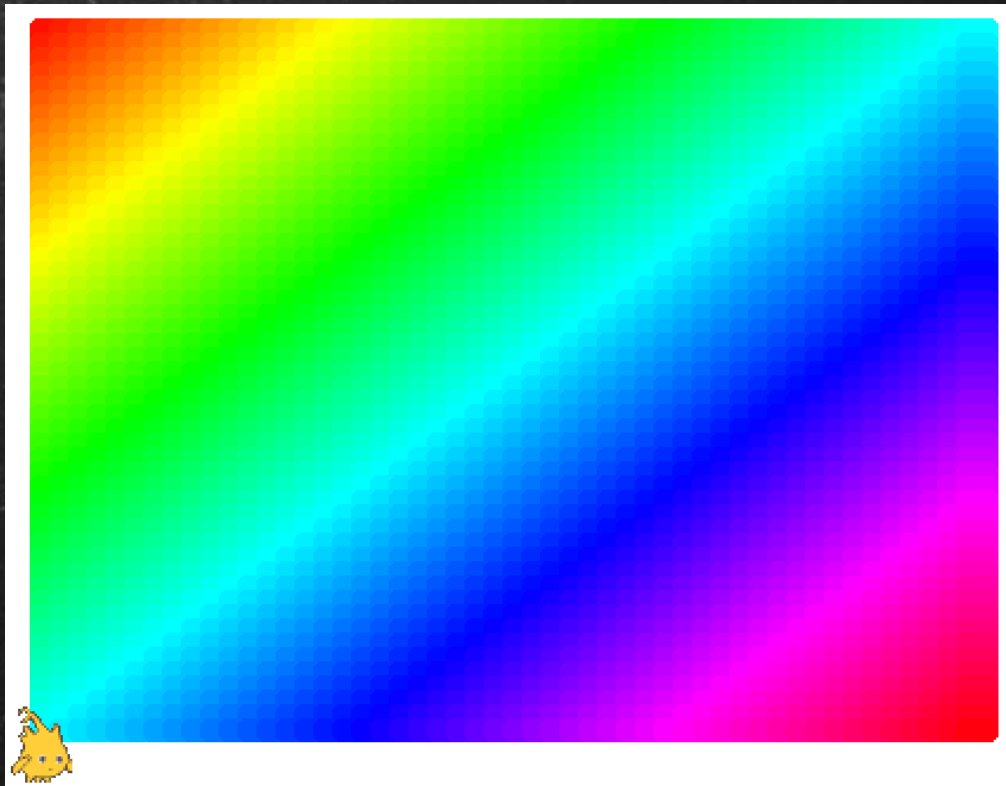
Take a look at the code to the right. What is it doing? What is its running time? Hint: it drew the picture below.



```
do cooler stuff with list
clear
set pen size to 10
set size to 25 %
script variables i j
go to x: -200 y: 150
pen down
set j to 1
repeat length of list
  set i to 1
  repeat length of list
    set pen color to item i of list + item j of list
    move round 400 / length of list steps
    change i by 1
  pen up
  change y by -1 * round 300 / length of list
  set x to -200
  pen down
  change j by 1
pen up
```

# What is the runtime?

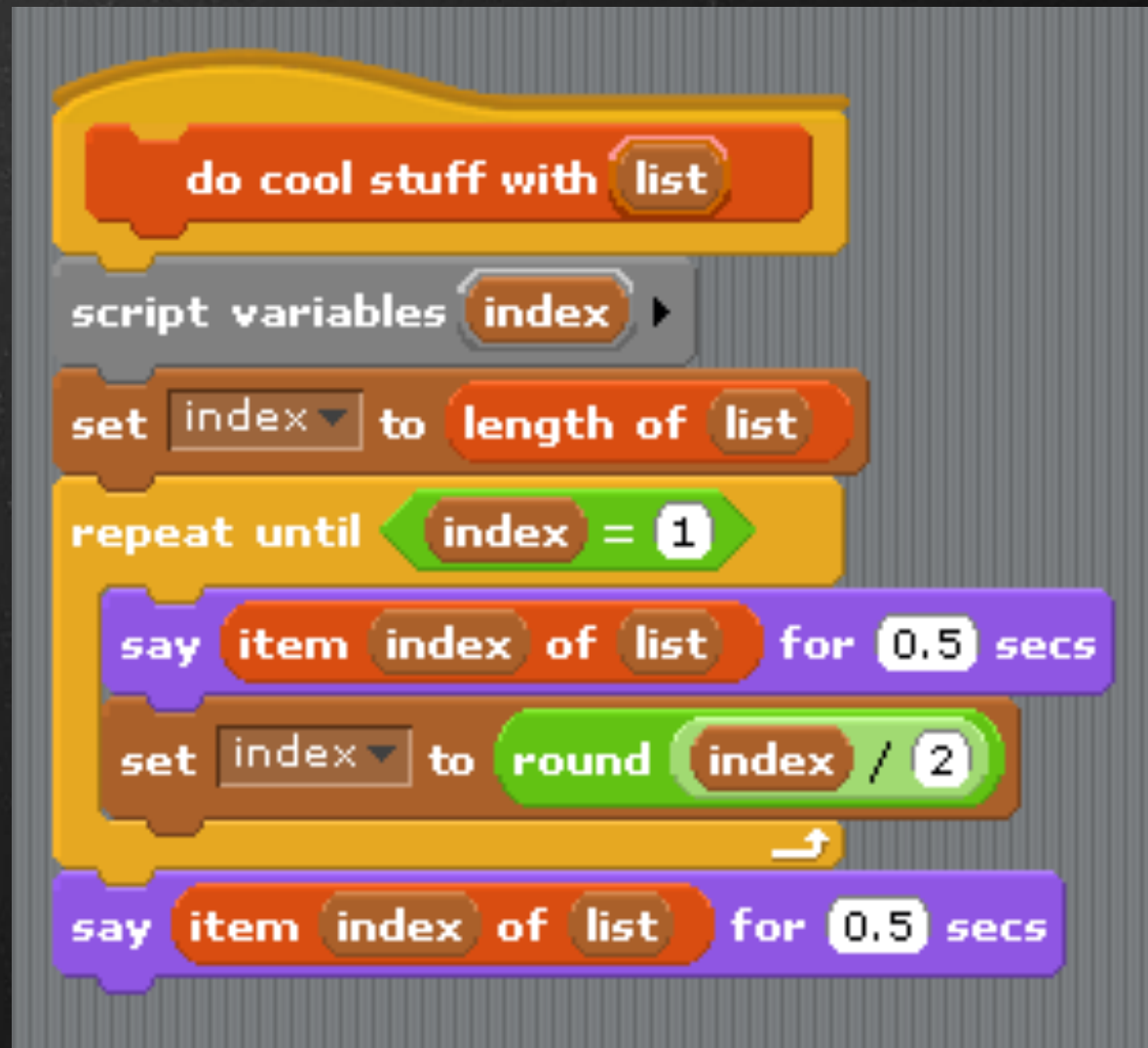
Take a look at the code to the right. What is it doing? What is its running time? Hint: it drew the picture below.



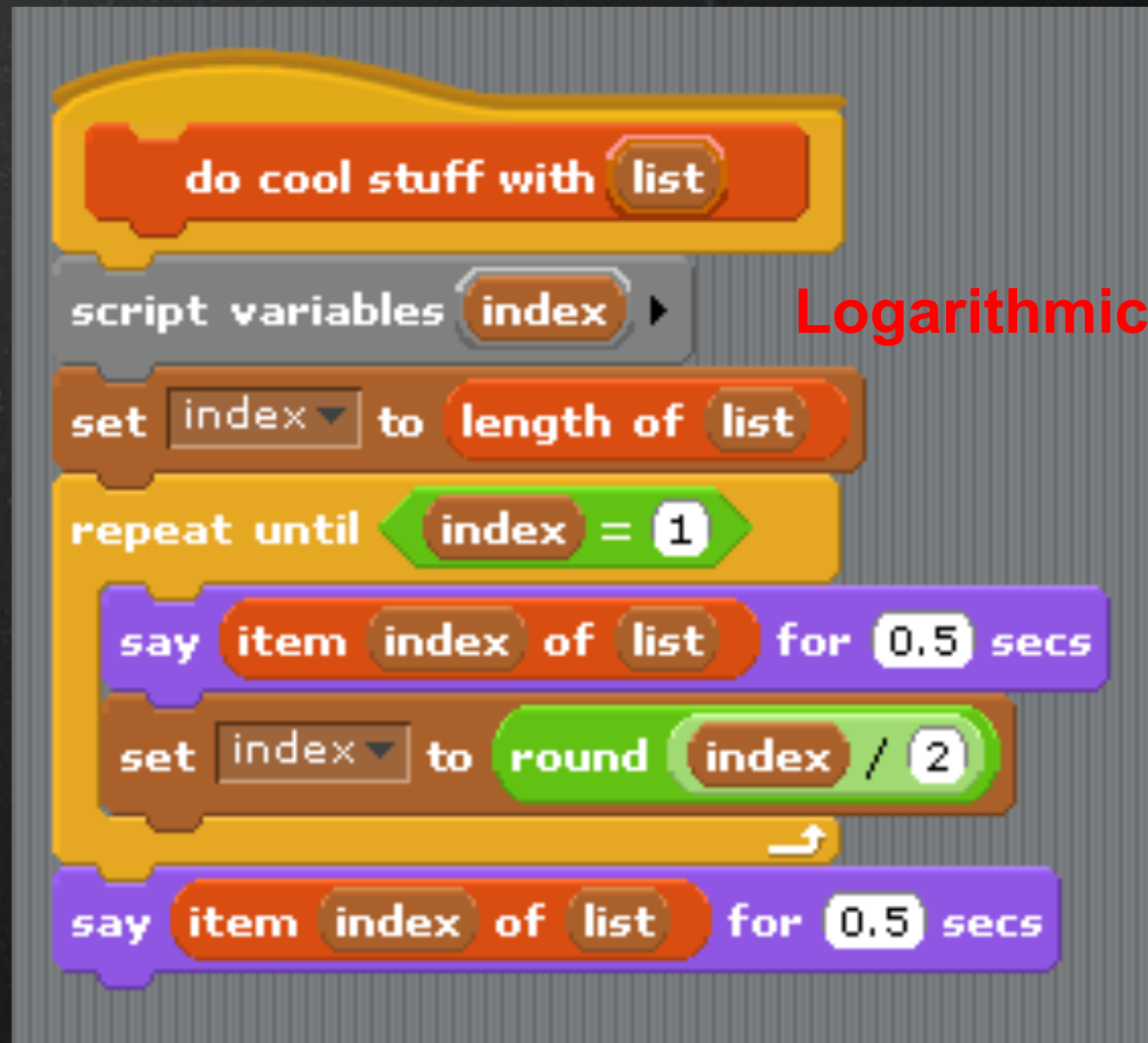
**Quadratic**

```
do cooler stuff with list
clear
set pen size to 10
set size to 25 %
script variables i j
go to x: -200 y: 150
pen down
set j to 1
repeat length of list
  set i to 1
  repeat length of list
    set pen color to item i of list + item j of list
    move round 400 / length of list steps
    change i by 1
  pen up
  change y by -1 * round 300 / length of list
  set x to -200
  pen down
  change j by 1
pen up
```

# What is the runtime?



# What is the runtime?





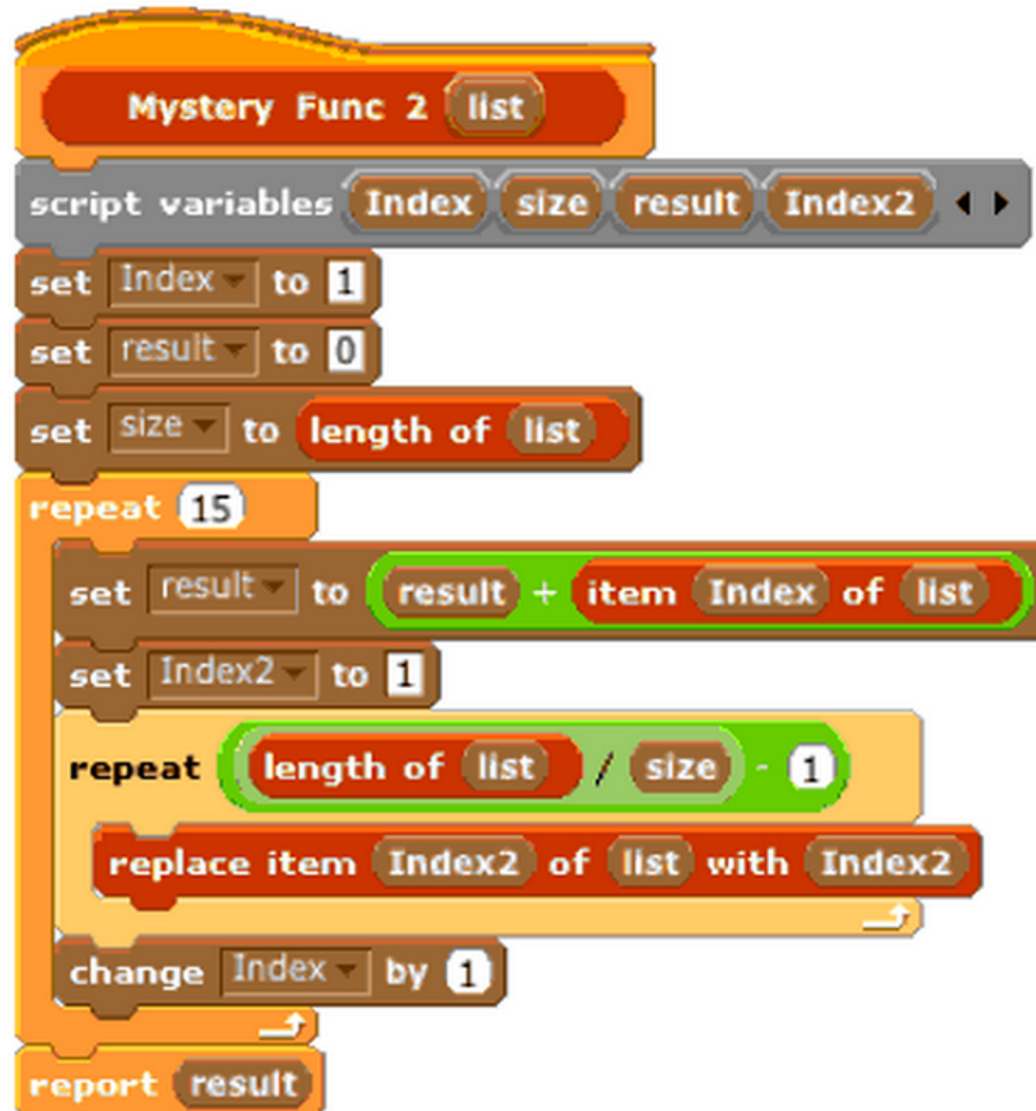
# What is the runtime?



# What is the runtime?



# What is the run-time?



# What is the run-time?

