



"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.



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).

Here is a list of some common running times:

Constant	
----------	--

O(1)

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(n²)

A quadratic-time algorithm is one 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(n³)

A cubic-time algorithm is one that takes a number of steps proportional to n³. 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 n² algorithm to n items, or applying a linear algorithm to n² 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(2ⁿ)

An exponential-time algorithm is one that takes time proportional to 2ⁿ. 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

```
Calculate Average of list
script variables index sum
set index √ to 1
set Sum √ to 0
repeat length of list
                        item (index) of (list)
                 sum)+
 change index -
                length of (list)
        sum /
report
```

```
+<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
      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
```

```
are the numbers of "list" distinct?
script variables [ walter || walker || current ||
set | waiter - to 1
               waiter 🔀 length of (list)
repeat until
  set current v to item waiter of list
  set walker to waiter + 1
                  walker > length of list
  repeat until
            current item (walker) of (list)
      say No, the numbers in the list are not distinct. for 1 secs
      stop block
    change walker by 1
  change | waiter = | by 1
    Yes, the numbers in the list are distinct. for 1 secs
```

```
DoubleSquare list
script variables Index CurrentNum 🕕
set Index v to 1
repeat length of (list)
 set CurrentNum to item Index of numbers ▼
 replace item Index of numbers with
   CurrentNum + (CurrentNum)
 change Index by 1
set Index to 1
repeat length of (list)
 set CurrentNum to item Index of numbers v
 replace item Index of numbers with
   CurrentNum) * (CurrentNum)
 change Index by 1
```

```
Mystery num list
script variables min max half current +
set min to 0
set Max to length of list + 1
            max = (min) + (1)
repeat until
 set half to round (min) + (max) / 2
     current to item half of list
        current = num
   report (half)
 else
         current < num
     set min to half
     set max to half
report 0
```

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
set pen size to 10
set size to 25 %
script variables 🚺 🚺 🕕
go to x: -200 y: 150
pen down
set j v to 1
repeat length of list
 set 💌 to 1
 repeat length of list
   set pen color to litem i of list + item j of list
   move round 400 / length of list
   change 💌 by 1
  pen up
 change y by [-1] * round [300] / length of list
 set x to -200
 pen down
 change j by 1
```

```
do cool stuff with list
script variables (index)
set index▼ to length of list
repeat until (index) = 1
 say item index of list for 0.5 secs
 set index to round index / 2
    item index of list for 0.5 secs
```

```
Are the elements of list list in the list list2
script variables[index]>
set index to 1
repeat (length of (list)
            not
            Is the item (item (index) of (list) in the list (list2)
 if
            (use binary search)
            false
    report
  change index by 1
```

```
Mystery Func 2 (list)
script variables Index I size I result I Index2
set Index to 1
set result to 0
set Size to length of list
repeat 15
  set result to (result) + item Index of list
  set Index2 v to 1
 repeat (length of list) / size
   replace item (Index2) of (list) with (Index2)
  change Index v by 1
```