Big-Oh notation

there is an upper bound to how much time accessing a memory location will take when running a program. The formal statement
of an upper bound is called Big-Oh notation.

The Big-Oh refers to the Greek letter Omicron which is typically used when talking about upper bounds.

In terms of the memory of a computer, we wanted to know how our program would perform if we have a very large list of elements.

- Let's represent the size of the list by a variable called n.
- Let the average access time for accessing an element of a list of size n be given by $f(n)$.

Now we can state the following.

$$O(g(n)) = \{f(n) \quad | \quad \exists\, d > 0,\ n_0 > 0 \in \mathbb{Z}^+ \ni 0 \leq f(n) \leq d * g(n), \forall\ n \geq n_0\}$$

The class of functions designated by $O(g(n))$ consists of all functions f where exists a d greater than 0 and exists an n0 (a positive integer), such 0 is less than or equal to $f(n)$ and $f(n)$ is less than or equal to d times $g(n)$, for all n greater than or equal to n_0.

If f is an element of $O(g(n))$, we say that $f(n)$ is $O(g(n))$. The function g is called an asymptotic upper bound for f in this case.

> 💡 $O(g(n))$ consists of the set of all functions $f(n)$, that have an upper bound of $d * g(n)$, as n approaches ∞.

Because the time to access an element does not depend on n, we can pick $g(n) = 1$. So we say that the average time to access an element in a list of size n is $O(1)$.

If we assume it never takes longer than 100 µs to access an element of a list in Python, then a good choice for d would be 100.

According to the definition above then it must be the case that $f(n)$ is less than or equal to 100 once n gets big enough.

The choice of $g(n) = 1$ is arbitrary in computing the complexity of accessing an element of a list.

We could have chosen $g(n) = 2$. If $g(n) = 2$ were chosen, d might be chosen to be 50 instead of 100. But, since we are only concerned with the overall
growth in the function g, the choice of 1 or 2 is irrelevant and the simplest function is chosen, in this case $O(1)$.

In English, when an operation or program is $O(1)$, we say it is a constant time operation or program. This means the operation does not depend on the size of n.

Most operations that a computer can perform are $O(1)$. Adding, multiplying or comparing two numbers together is a $O(1)$ operation, since the elapsed times does not depend on the value or size of the two values being operated.

When computing complexity, any arithmetic calculation or comparison can be considered a constant time operation.

> 💡 Storing or retrieving a value from the memory are also $O(1)$ operations. This is valid when talking about RAM memory. Storing/retrieving data from a hard drive is a different computation that depends on the disk type.

This idea of computational complexity is especially important when the complexity of a piece of code depends on n.

The PyList Append Operation

Inefficient Append

```
class PyList:
    def __init__( self ) -> None:
        self.items = []

    # Its own mutator append method... maybe not the most efficient
    def append( self , new_item ):
        self.items = self.items + [ new_item ]
```

This code appends new item to its items list as follows:

1. The new item is made into a one-element list by putting '[' and ']' around it. So we create a new element in memory that points to *new_item*'s value.
2. *PyList*'s *items* list is concatenated with the new one-element list using the + operator. The concatenation result is stored in a new memory space.
3. The assignment of the concatenation result to *self.items* updates the *PyList* items object. So it now refers to a new list

> 💡 How does this append method perform as the size of the *PyList* grows?

The first time the append method is called, there are 0 elements in the *self.items* list and 1 element in the *[new_item]* list. So the append method must access 1 element of a list to form the concatenated list, which will have a total of 1 element in it.

The second time the append method is called, there is 1 element in the *self.items* list and 1 element in the *[new_item]* list. So the append method must access 2 elements to form the new list.

The third time the append method is called, a total of 3 elements must be accessed to form the concatenated list.

> 💡 So on, in order to make it up to the *n-th* append operation, there will have to be *n* elements copied to form the new list.

If we want to calculate the amount of time it takes to append *n* elements to the *PyList* we would have to:

- Add up all the list accesses and multiply them by the amount of time it takes to access a list element
- Add the time it takes to store a list element.

So, we can state that the amount of time to append n elements can be written as:

$$f(n) = \sum_{i=1}^{n} i = 1 + 2 + ... + n$$

Remember that retrieve and store operations are O(1) operations, so we asume their execution time to be 1.

We can rewrite the summation expression as:

$$f(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$

💡 It is seen that for a given *n*, the highest power of n in the $f(n)$ expression is n^2, so we conclude that the current PyList append method exhibits $O(n^2)$ complexity, which is not a desired performance.

💡 In terms of *big-Oh* notation we say that the append method is $O(n^2)$.

💡 You typically want to stay away from writing code that has this kind of computational complexity associated with it unless you are absolutely sure it will never be called on large data sizes.

If we take another look at our PyList append method we might be able to make it more efficient if we didn't have to access each element of the first list when concatenating the two lists.

The use of the + operator is what causes Python to access each element of that first list. When + is used a new list is created with space for one more element. Then all the elements from the old list must be copied to the new list and the new element is added at the end of this list.

Using the append method on lists changes the code to use a mutator method to alter the list by adding just one more element.

It turns out that adding one more element to an already existing list is very efficient in Python.

💡 Appending an item to a *list* using the *append* command is a $O(1)$ operation. This means that to append n items to a PyList we have gone from $O(n^2)$ to $O(n)$ complexity by using the built-in append operation instead of the initially declared append function.

Efficient Append

```
class PyList:
    def __init__( self ) -> None:
        self.items = []

    # Its own mutator append method... maybe not the most efficient
    def append( self , new_item ):
        self.items.append( new_item )
```

Commonly Occurring Computational Complexities

The algorithms we will study will be of one of the complexities of $O(1)$, $O(log\ n)$, $O(n\ log\ n)$, $O(n^2)$, or $O(c^n)$. Most algorithms have one of these complexities corresponding to some factor of n.

Constant values added or multiplied to the terms in a formula for measuring the time needed to complete a computation do not affect the overall complexity of that operation.

Computational complexity is only affected by the highest power term of the equation.

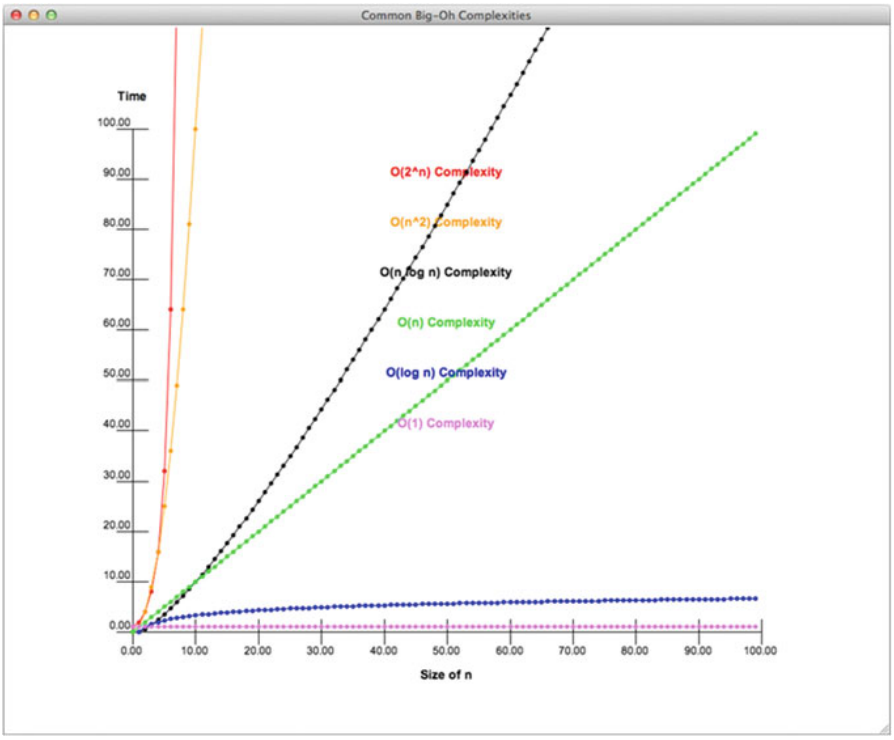
An algorithm that has exponential complexity (i.e. $O(c^n)$) or n-squared complexity (i.e. $O(n^2)$) complexity will not perform very well except for very small values of n.

You will encounter sorting algorithms that are $O(n^2)$ and then you'll learn that we can do better and achieve $O(n\ log\ n)$ complexity.

You'll see search algorithms that are $O(n)$ and then learn how to achieve $O(log\ n)$complexity

A technique called hashing searches in O(1) time.

Common Big-Oh complexities



When concerning ourselves with algorithm efficiency there are two issues that must be considered.

- The amount of time an algorithm takes to run
- The amount of space (RAM memory) an algorithm uses while running.

The time we are concerned with is a measure of how the number of operations grow as the size of the data grows.

Consider a function $T(n)$ the running time of an algorithm where n is the size of the data given to the algorithm. We want to study how $T(n)$ increases as $n \to \infty$.

Big-Oh Asymptotic Upper Bound

Given

$$O(g(n)) = \{f(n) \quad | \quad \exists\, d > 0,\ n_0 > 0 \in \mathbb{Z}^+ \ni 0 \leq f(n) \leq d * g(n), \forall\ n \geq n_0\}$$

We write that

$$f(n)\ is\ O(g(n)) \Leftrightarrow f \in O(g(n))$$

> 💡 And we say that f is big-oh g of n. The definition of Big-Oh says that we can find an upper bound for the time it will take for an algorithm to run.

Saying that the algorithm runs in $O(n^2)$ is accurate even if the algorithm runs in time proportional to n because Big-Oh notation only describes an upper bound. If we truly want to say what the algorithm's running time is proportional to, then we need a little more power.

Asymptotic Lower Bound

> 💡 Omega notation serves as a way to describe a lower bound of a function. There is some n_0 where $T(n)$ dominates $g(n)$ as $n \to \infty$. In that case, we can write that the algorithm is $\Omega(g(n))$.

$$\Omega(g(n)) = \{f(n) \quad | \quad \exists\ c > 0,\ n_0 > 0 \in \mathbb{Z}^+ \ni 0 \leq c * g(n) \leq f(n), \forall\ n \geq n_0\}$$

With both a lower bound and and upper bound definition, we now have the notation to define an asymptotically tight bound. This is called Theta notation.

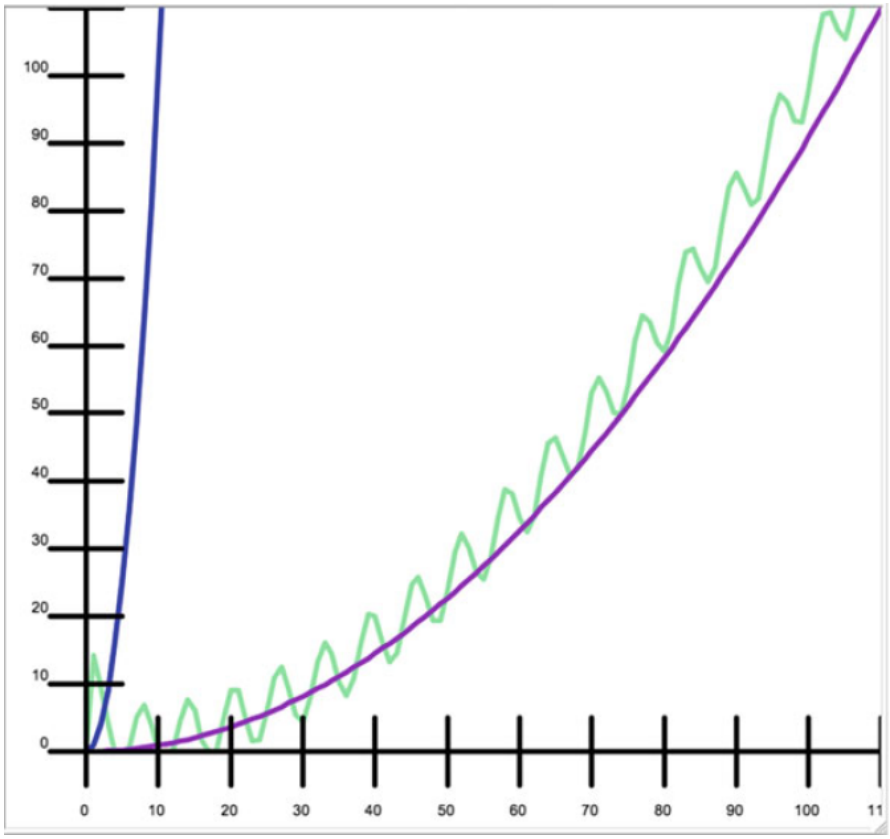
Theta Asymptotic Tight Bound

If we can find such a function g, then we can declare that $\Theta(g(n))$ is an asymptotically tight bound for $T(n)$,

$$\Theta(g(n)) = \{f(n) \quad | \quad \exists\ c > 0,\ n_0 > 0 \in \mathbb{Z}^+ \ni 0 \leq c * g(n) \leq f(n), \forall\ n \geq n_0\}$$

For the observed behavior of an algorithm:

- The upper bound blue line is $g_1(n) = n^2$
- The lower bound purple line is $g_2(n) = n^2/110$



If we let $c = 1$ and $d = 1/110$, we have the asymptotically tight bound of $T(n)$ at $\Theta(n^2)$.

Now, instead of saying that n^2 is an upper bound on the algorithm's behavior, we can proclaim that the algorithm truly runs in time proportional to n^2. The behavior is bounded above and below by n^2 functions, proving the claim that the algorithm is an n^2 algorithm.

Amortized complexity

Sometimes it is not possible to find a tight upper bound on an algorithm.

Most operations may be bounded by some function $c * g(n)$ but every once in a while there may be an operation that takes longer.

In these cases it may be helpful to employ something called Amortized Complexity.

The key behind amortization methods is to get an upper bound as tight as we can for the worst case running time of any sequence of n operations on a data structure, which usually starts out empty.

> 💡 By dividing the total running time by n, we get the average or amortized running time of each operation in the sequence.

Consider the PyList append operation. The efficient version of the PyList append method simply calls the Python append operation on its list.

Python is implemented in C. It turns out that while Python supports an append operation for lists, lists are implemented as arrays in C and it is not possible in C to modify an array's size once created, so technically there is not a native append in C for arrays.

Pretend for a moment that Python lists, like C arrays, did not support the append method on lists and that the only way to create a list was to write something like `[None]*n` where `n` was a fixed value.

Writing `[None]*n` creates a fixed size list of `n` elements each referencing the value `None`. This is the way C and C++ arrays are allocated.

In our example, since we are pretending that Python does not support append we must implement our PyList append method differently.

We can't use the append method and earlier in this chapter we saw that that adding on item at a time with the `+` operator was a bad idea.

Our PyList append operation, when it runs out of space in the fixed size list, will double the size of the list copying all items from the old list to the new list

```
class PyList:
    # size = 1 is the initial number of locations for the list object.
    # num_items tracks how many elements are currently stored since self.items may have empty locations
```

```
    def __init__( self , size = 1 ) -> None:
        self.items = [None] * size
        self.num_items = 0

    def append( self , new_item ):
        # If there is no free space in the items list for the new item
        if self.num_items == len( self.items ):
            # Make the list bigger by allocating a new list twice the current size
            # Copy all the elements over to the new list
            new_list = [ None ]*( self.num_items*2 )

            for index in range( len( self.items ) ):
                new_list[ index ] = self.items[ index ]

            self.items = new_list

        self.items[ self.num_items ] = new_item
        self.num_items += 1
```

Using this new PyList append method, a sequence of n append operations on a PyList object, starting with an empty list, takes $O(n)$ time meaning that individual operations must not take longer than $O(1)$time.

Whenever the list runs out of space a new list is allocated and all the old elements are copied to the new list.

Copying n elements from one list to another takes longer than $O(1)$ time.

Understanding how append could exhibit $O(1)$ complexity relies on computing the amortized complexity of the append operation.

Technically, when the list size is doubled the complexity of append is $O(n)$. But how often does that happen? The answer is *not that often*.

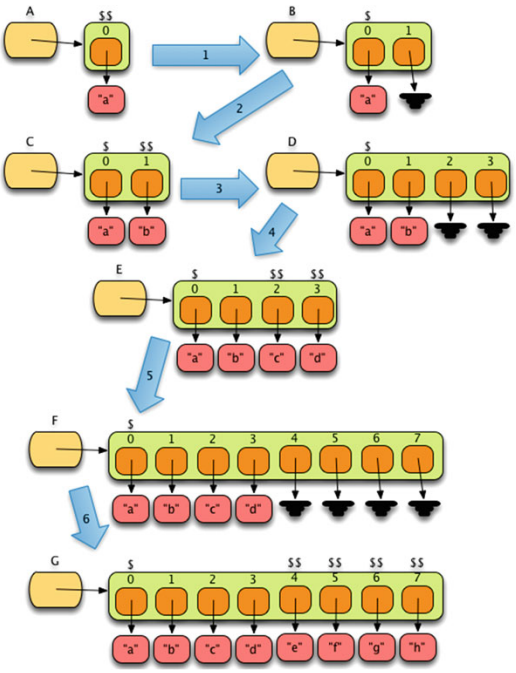
Proof of Append Complexity

The proof that the append method has O(1) complexity uses what is called the accounting method to find the amortized complexity of append.

The accounting method stores up cyber dollars to pay for expensive operations later.

The idea is that there must be enough cyber dollars to pay for any operation that is more expensive than the desired complexity.

Consider a sequence of n append operations on an initially empty list.



Appending the first element to the list is done in O(1) time since the initially empty list has space for the first item, so the allocation operation is done straightforwardly.

However, according to the accounting method, we will assume that the cost of doing the append operation requires an additional two cyber dollars.

This is still $O(1)$ complexity.

Each time we run out of space we'll double the number of slots in the fixed size list.

Allocating a fixed size list is a $O(1)$ operation regardless of the list size. The extra work comes when copying the elements from the old list to the new list.

The first time we need to double the size is when the second append is called. There are two cyber dollars stored up at this point in time.

One of them is needed when copying the one element stored in the old list to the new fixed size list capable of holding two elements.

Transition one shows the two stored cyber dollars and the result after copying to the new list when moving from step A to step B.

When append is called on version B of the list the result is version C. At this point 3 cyber dollars are

stored to be used when doubling the list size from 2 to 4.

When appending the next item, form C to D, we declare a new list of size 4. The first two spaces are filled with the old contents of the list. Two of the three stored cyber dollars are used while copying these values to the new list.

When the list of size four fills, two additional append operations have occurred, storing five cyber dollars.

Four of these cyber dollars are used in the copy from step E to step F.

What if we didn't double the size of the list each time. If we increased the list size by one half its previous size each time, we could still make this argument work if we stored four cyber dollars for each append operation.

As long as the size of the list grows proportionally to its current size each time it is expanded this argument still works to prove that appending to a list is a $O(1)$ operation when lists must be allocated with a fixed size.

Calculating the Time Complexity for the following Algorithms

Example 1: Simple loop

```
x = 0

for i in range( n ):
  x += i
```

The running time of a loop is the multiple of, statements inside the loop by the number of iterations.

Each cycle takes a c amount of time

$$Total\ time = c * n = O(n)$$

Example 2: Nested loops

```
x = 0

for i in range( n ):
  for j in range( m ):
    x += i + j
```

The total running time of this algorithm is the product of the sizes of all the loops.

$$Total\ time = c * n * m$$

$$As\ n\ and\ m \to \infty : n = m \qquad then \qquad Total\ time = c * n^2 \to O(n^2)$$

Example 3: Consecutive loops

```
x = 0

# Nested
for i in range( n ):
  for j in range( m ):
    x += i + j

# Not nested
for k in range( o ):
  x += k
```

The total running time of this algorithm is the product of the sizes of all the loops.

$$Total\ time = c_1 * n * m + c_2 * o$$

$$As\ n,\ m,\ o \to \infty : n = m = o \qquad then \qquad Total\ time = c_1 * n^2 + c_2 * n \to O(n^2)$$

Example 4: Logarithmic complexity

```
i = 1

while( i <= n ):
  i = i * 2
```

If we observe the value of `i`, it is doubling on every iteration `i=1,2,4,8..` , and the original problem is cut by a fraction $(1/2)$ on each iteration, the complexity of this kind is of log time.

If $n = 1$ it takes k=1 iteration… it covers 1 value

If $n = 2$ it takes k=2 iterations… it covers 1 value

If $n = [3, 4]$ it takes k=3 iterations… it covers 2 values

If $n = [5, 6, 7, 8]$ it takes k=4 iterations… it covers 4 values

If $n = [9, 10, 11, 12, 13, 14, 15, 16]$ it takes k=5 iterations … it covers 8 values

If $n = [17, ..., 32]$ it takes k=6 iterations… it covers 16 values

Note that the amount of values in n is 2^{k-3}. So we can establish that, the worst case scenario for each iteration is

$$n = 2^{k-3}$$

We can rewrite the equation to find k as a function of n

$$log_2(n) = log_2(2^{k-3})$$

$$log_2(n) = k - 3$$

$$k = log_2(n) + 3$$

$$as\ n \to \infty : \quad k = log_2(n) \to O(log_2(n))$$

If we generalize the increase of `i` we can finally state that this kind of loop is $O(log(n))$