Big O and Similar Notations

In computer science, choosing the right algorithm for the job can make the difference between having something that takes a few seconds to run versus something that takes days or weeks. As an example, here are three different ways to sum up the integers from 1 to n.

1. Probably the most obvious way would be to loop from 1 to n and use a variable to keep a running total.

```
total = 0
for i in range(1,n+1):
    total += i
```

2. If you know enough math, there is a formula $1+2+\cdots+n = n(n+1)/2$. So the sum could be done in one line, like below:

```
total = n*(n+1)/2
```

3. Here is a bit of a contrived approach using nested loops.

```
total = 0
for i in range(1, n+1):
    for j in range(i):
        total += 1
```

Any of these algorithms would work fine if we are just adding up numbers from 1 to 100. But if we are adding up numbers from 1 to a billion, then the choice of algorithm would really matter. On my system, I tested out adding integers from 1 to a billion. The first algorithm took about 1 second to add up everything. The next algorithm returned the answer almost instantly. I estimate the last algorithm would take around 12 years to finish, based on its progress over the three minutes I ran it.

In the terminology of computer science, the first algorithm is an $O(n)$ algorithm, the second is an $O(1)$ algorithm, and the third is an $O(n^2)$ algorithm.

This notation, called *big O notation*, is used to measure the running time of algorithms. Big O notation doesn't give the exact running time, but rather it's an order of magnitude estimation. Measuring the exact running time of an algorithm isn't practical as so many different things like processor speed, amount of memory, what else is running on the machine, the version of the programming language, etc. can affect the running time. So instead, we use big O notation, which measures how an algorithm's running time grows as some parameter n grows. In the example above, n is the integer we are summing up to, but in other cases it might be the size of an array or list, the number of items in a matrix, etc.

With big O, we usually only care about the dominant or most important term. So something like $O(n^2+n+1)$ is the same as $O(n^2)$, as n and 1 are small in comparison to n^2 as n grows. Also, we usually ignore constants. So $O(3.47n)$ is the same as $O(n)$. And if we have something like $O(2n+4n^3+.6n^2-1)$, we would just write that as $O(n^3)$. Remember that big O is just an order of magnitude estimation, and we don't often care about being exact.

Estimating the running times of algorithms

To estimate the big O running time of an algorithm, here are a couple of rules of thumb:

1. If an algorithm runs in the same amount of time regardless of how large n is, then it is $O(1)$.
2. A loop that runs n times will contributes a factor of n to the big O running time.
3. If loops are nested, multiply their running times.
4. If one loop follows another, add their running times.

5. If the loop variable is increasing in a way such as `i=i*2` or `i=i/3`, instead of changing by a constant amount (like `i += 1` or `i-=2`), then that contributes a factor of $\log n$ to the big O running time.

Here are several examples of how to compute the big O running time of some code segments. All of these involve working with an array, and we will assume n is the length of the array.

1. Here is code that sums up the entries in an array:

```
total = 0
for i in range(len(a)):
    total += a[i]
```

   This code runs in $O(n)$ time. It is a pretty ordinary loop.

2. Here is some code involving nested for loops:

```
for i in range(len(a)):
    for j in range(len(a)):
              print(a[i]+a[j])
```

   These are two pretty ordinary loops, each running for $n =$ `len(a)` steps. They are nested, so their running times multiply, and overall the running time is $O(n^2)$. For each of the n times the outer loop runs, the inner loop has to also run n times, which is where the n^2 comes from.

3. Here are three nested loops:

```
for i in range(len(a)):
    for j in range(len(a)-1):
        for k in range(len(a)-2:
            print(a[i]+a[j]*a[k])
```

   This code runs in $O(n^3)$ time. Technically, since the second and third loops don't run the full length of the array, we could write it as $O(n(n-1)(n-2))$, but remember that we are only interested in the most important term, which is n^3, as $n(n-1)(n-2)$ can be written as n^3 plus a few smaller terms.

4. Here are two loops, one after another:

```
count = count2 = 0
for i in range(len(a)):
    count += 1

for i in range(len(a)):
    count2 += 2
```

   The running time here is $O(n+n) = O(2n)$, which we simplify to $O(n)$ because we ignore constants.

5. Here are a few simple lines of code:

```
w = a[0]
z = a[len(a)-1]
print(w + z)
```

   The running time here is $O(1)$. The key idea is that the running time has no dependence on n. No matter how large that array is, this code will always take the same amount of time to run.

6. Here is some code with a loop:

```
c = 0
stop = 100
for i in range(stop):
    c += a[i]
```

   Despite the loop, this code runs in $O(1)$ time. The loop always runs to 100, regardless of how large the array is. Notice that `len(a)` never appears in the code.

7. Here is a different kind of loop:

```
sum = 0
i = 1
while i < len(a):
    sum += a[i]
    i *= 2
```

This code runs in $O(\log n)$ time. The reason is the that the loop variable is increasing via `i *= 2`. It goes up as 1, 2, 4, 8, 16, 32, 64, 128, ... It takes only 10 steps to get to 1000, 20 steps to get to 1 million, and 30 steps to get to 1 billion.

The number of steps to get to n is gotten by solving $2^x = n$, and the solution to that is $\log_2(n)$ (which we often write just as $\log(n)$).[1]

8. Here is a set of nested loops:

```
n = len(a)
for i in range(n):
    sum = 0
    m = n
    while m > 1:
        sum += m
        m /= 2
```

This code has running time $O(n \log n)$. The outer loop is a pretty ordinary one and contributes the factor of n to the running time. The inner loop is a logarithmic one as the loop variable is cut in half at each stage. Since the loops are nested, we multiply their running times to get $O(n \log n)$.

9. Here is some code with several loops:

```
total = 0
i = 0
while i < len(a):
    i += 1
    total += a[i]

for i in range(len(a)/2):
    for j in range(len(a)-1,-1,-1):
         total += a[i]*a[j];
```

The running time is $O(n^2)$. To get this, start with the fact that the while loop runs in $O(n)$ time. The nested loops' running times multiply to give $O(n \cdot n/2)$. The while loop and the nested loops follow one another, so we add their running times to get $O(n + n \cdot n/2)$. But remember that we only care about the dominant term, and we drop constants, so the end result is $O(n^2)$.

10. Here is a simple loop with a possibly surprising running time:

```
i = 0
while i < len(a):
    if a[i] == 0:
        del(a[i])
    i += 1
```

It looks like just a simple loop with running time $O(n)$, but there is a trick. The `del` method, which deletes items from the list, actually itself has $O(n)$ running time. That function essentially leaves a hole in the list and items have to be shifted left to fill in the hole. So overall, this code has running time $O(n^2)$.

Common running times

The most common running times are probably $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, and $O(2^n)$. These are listed in order of desirability. An $O(1)$ algorithm is usually the most preferable, while an $O(2^n)$ algorithm should be avoided if at all possible.

[1] For logs, there is a formula, called the change-of-base-formula, that says $\log_a(x) = \log_b(x)/\log_b(a)$. In particular, all logs are constant multiples of each other, and with big O, we don't care about constants. So we can just write $O(\log n)$ and say that the running time is logarithmic.

To get a good feel for the functions, it helps to compare them side by side. The table below compares the values of several common functions for varying values of n.

	1	**10**	**100**	**1000**	**10000**
1	1	1	1	1	1
$\log n$	0	3.3	6.6	10.0	13.3
n	1	10	100	1000	10,000
$n \log n$	0	33	664	9966	132,877
n^2	1	100	10,000	1,000,000	100,000,000
2^n	2	1024	1.2×10^{30}	1.1×10^{301}	2.0×10^{3010}

Things to note:

1. The first line remains constant. This is why O(1) algorithms are usually preferable. No matter how large the input gets, the running time stays the same.

2. Logarithmic growth is extremely slow. An O($\log n$) algorithm is often nearly as good as an O(1) algorithm. Each increase by a factor of 10 in n only corresponds to a growth of about 3.3 in the logarithm. Even at the incredibly large value $n = 10^{100}$ (1 followed by 100 zeros), $\log n$ is only about 332.

3. Linear growth (O(n)) is what we are most familiar with from real life. If we double the input size, we double the running time. If we increase the input size by a factor of 10, the running time also increases by a factor of 10. This kind of growth is often manageable. A lot of important algorithms can't be done in O(1) or O($\log n$) time, so it's nice to be able to find a linear algorithm in those cases.

4. The next line, for $n \log n$, is sometimes called loglinear or linarithmic growth. It is worse than O(n), but not all that much worse. The most common occurrence of this kind of growth is in the running times of the best sorting algorithms.

5. With quadratic growth (O(n^2)), things start to go off the rails a bit. We can see already with n =10,000 that n^2 is 100 million. Whereas with linear growth, doubling the input size doubles the running time, with quadratic growth, doubling the input size corresponds to increasing the running time by 4 times (since $4 = 2^2$). Increasing the input size by a factor of 10 corresponds to increasing the running time by $10^2 = 100$ times.

   Some problems by their very nature are O(n^2), like adding up the elements in a 2-dimensional array. But in other cases, with some thought, an O(n^2) algorithm can be replaced with an O(n) algorithm. If it's possible that the n you're dealing with could get large, it is worth the time to try to find a replacement for an O(n^2) algorithm.

6. The last line is exponential growth, O(2^n). Exponential growth gets out of hand extremely quickly. None of the examples we saw earlier had this running time, but there are a number of important practical optimization problems whose best known running times are exponential.

7. Other running times — There are infinitely many other running times, like O(n^3), O(n^4), ..., or O($n^{1.5}$), O($n!$), and O(2^{2^n}). The ones listed above are just the most common.

Some notes about big O notation

Here are a few important notes about big O notation.

1. When we measure running times of algorithms, we can consider the best-case, worst-case, and average-case performance. For instance, let's say we are searching an array element-by-element to find the location of a particular item. The best case would be if the item we are looking for is the first thing in the list. That's an O(1) running time. The worst case is if the element is the last thing, corresponding to a O(n) running time. The average case is that we will need to search around half of the elements before

finding the element, corresponding to a $O(n/2)$ running time, which we would write as $O(n)$, since we ignore constants with big O notation.

Of the three, usually what we will refer to when looking at the big O of an algorithm is the worst case running time. Best case is interesting, but not often all that useful. Average case is probably the most useful, but it is typically harder to find than the worst case, and often it turns out to have the same big O as the worst case.

2. We have mostly ignored constants, referring to values such as $3n$, $4n$, and $5n$ all as $O(n)$. Often this is fine, but in a lot of practical scenarios this won't give the whole story. For instance, consider algorithm A that has an exact running time of $10000n$ seconds and algorithm B that has an exact running time of $.001n^2$ seconds. Then A is $O(n)$ and B is $O(n^2)$. Suppose for whatever problem these algorithms are used to solve that n tends to be around 100. Then algorithm A will take around 1,000,000 seconds to run, while algorithm B will take 10 seconds. So the quadratic algorithm is clearly better here. However, once n gets sufficiently large (in this case around 10 million), then algorithm A begins to be better.

   An $O(n)$ algorithm will always *eventually* have a faster running time than any $O(n^2)$ algorithm, but n might have to be very large before that happens. And that value of n might turn out larger than anything that comes up in real situations. Big O notation is sometimes called the *asymptotic* running time, in that it's sort of what you get as you let n tend toward infinity, like you would to find an asymptote. Often this is a useful measure of running time, but sometimes it isn't.

3. Big O is used for more than just running times. It is also used to measure how much space or memory an algorithm requires. For example, one way to reverse an array is to create a new array, run through the original in reverse order, and fill up the new array. This uses $O(n)$ space, as we need to create a new array of the same size as the original. Another way to reverse an array consists of swapping the first and last elements, the second and second-to-last elements, etc. This requires only one additional variable that is used in the swapping process. So this algorithm uses $O(1)$ space.

4. Big O notation sometimes can have multiple parameters. For instance, if we are searching for a substring of length m inside a string of length n, we might talk about an algorithm that has running time $O(n+m)$.

Big O, Big Omega, and Big Theta

Here is the formal mathematical definition of big O:

> $f(n)$ is $O(g(n))$ if there exist c, N, such that for all $n \geq N$, we have $f(n) \leq cg(n)$.

In this definition, $f(n)$ stands for the running time of a particular algorithm, and $g(n)$ is for what it is big O of. What the math is saying is that once the input size n gets past a certain point N, then the running time of the algorithm won't be any larger than $cg(n)$. In other words, it is eventually less than $cg(n)$. That constant c is needed in order to be able to say that something with running time $n+1$ is $O(n)$ since without it, it would not ever be true that $n+1 < n$. But there are values of c such that $n+1 < cn$ eventually. For instance, $n+1 \leq 2n$ for $n \geq 1$.

Notice the $\leq$ symbol in the definition. This actually tells us something interesting about big O notation. It's actually sort of a less than or equal to type of notation. In particular, an algorithm that has exact running time $3n$ is not only $O(n)$, but it is also technically $O(n^2)$, $O(n^3)$, and even $O(2^n)$. This is because $3n$ is eventually less than n^2, n^3, and 2^n.

There is a related notation big Ω that has the same definition as big O except that the $\leq$ is replaced with $\geq$. Saying $f(n)$ is $O(g(n))$ means that the running time is as good or better than a constant times $g(n)$ eventually. Saying $f(n)$ is $\Omega(g(n))$ means that the running time cannot be any better than a constant times $g(n)$ eventually. For instance, an algorithm with running time n^3 is $\Omega(n)$, $\Omega(n^2)$ and $\Omega(n^3)$.

There is one more notation, which is big Θ notation. If $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$, then it is $\Theta(g(n))$. In particular, big O is like a "less than or equal to," big Ω is like a "greater than or equal to," and big Θ is like "equal to." When most people use big O, what they really mean is big Θ, but it's been traditional to use big O this way,

and we'll do it in these notes as well. There may be a few times when we need to be careful, and in those cases we will use big Θ.

Below are a few examples:

- An algorithm with running time $2n^2 + 1$ is $O(n^2)$, but it is also $O(n^5)$ because $2n^2 + 1$ is eventually smaller than a constant times n^5. When we say something is $O(n^5)$, we are saying that its running time is at least as good as a constant times n^5 and possibly better.
- An algorithm with running time $2n^2 + 1$ is not $O(n)$ since $2n^2 + 1$ is not eventually less than a constant times n.
- An algorithm with running time $2n^2 + 1$ is $\Omega(n^2)$, but it is also $\Omega(n)$ because it is eventually larger than a constant times n. Saying something is $\Omega(n)$ means that its running time is at best as good as a constant times n and possibly worse.
- An algorithm with running time $2n^2 + 1$ is not $\Omega(n^3)$ since $2n^2 + 1$ is not worse than a constant times n^3.
- An algorithm with running time $2n^2 + 1$ is $\Theta(n^2)$. It is not $\Theta(n)$ or $\Theta(n^3)$. Remember that Θ is more of an exact measure of order.