**Complexity**Written by Jonathan Sande & Kelvin Lau

Will it scale?

This age-old question is always asked during the design phase of software development and comes in several flavors. From an architectural standpoint, **scalability** is how easy it is to make changes to your app. From a database standpoint, scalability is about how long it takes to save or retrieve data in the database.

For algorithms, scalability refers to how the algorithm performs in terms of execution time and memory usage as the input size increases.

When you’re working with a small amount of data, an expensive algorithm may still feel fast. However, as the amount of data increases, an expensive algorithm becomes crippling. So how bad can it get? Understanding how to quantify this is an important skill for you to know.

In this chapter, you’ll take a look at the **Big O notation** for the different levels of scalability in two dimensions: execution time and memory usage.

**Time Complexity**

For small amounts of data, even the most expensive algorithm can seem fast due to the speed of modern hardware. However, as data increases, the cost of an expensive algorithm becomes increasingly apparent. **Time complexity** is a measure of the time required to run an algorithm as the input size increases. In this section, you’ll go through the most common time complexities and learn how to identify them.

**Constant Time**

A constant-time algorithm has the same running time regardless of the size of the input. Consider the following:

void checkFirst(List<String> names) {

if (names.isNotEmpty) {

print(names.first);

} else {

print('no names');

}

}

The number of items in names has no effect on the running time of this function. Whether the input has 10 items or 10 million items, this function only prints the first element of the list. Here’s a visualization of this time complexity in a plot of time versus data size:

DataTimeO(1) Constant time

As input data increases, the amount of time the algorithm takes does not change.

For brevity, programmers use a way of writing known as **Big O notation** to represent various magnitudes of time complexity. The Big O notation for constant time is **O(1)**.

**Linear Time**

Consider the following snippet of code:

void printNames(List<String> names) {

for (final name in names) {

print(name);

}

}

This function prints out all the names in a list. As the input list increases in size, the number of iterations that the for loop makes increases by the same amount.

This behavior is known as **linear time** complexity:

DataTimeO(n) Linear time

Linear time complexity is usually the easiest to understand. As the amount of data increases, the running time increases by the same amount. That’s why you have the straight linear graph illustrated above. The Big O notation for linear time is **O(n)**.

**Note**: What about a function that has two loops over all the data and calls six O(1) methods? Is it O(2n + 6)?

Time complexity only gives a high-level shape of the performance, so loops that happen a fixed number of times are not part of the calculation. All constants are dropped in the final Big O notation. In other words, O(2n + 6) is equal to O(n).

Although not a central concern of this book, optimizing for absolute efficiency can be important. Companies put millions of dollars of R&D into reducing the slope of those constants that Big O notation ignores. For example, a GPU-optimized version of an algorithm might run 100x faster than the naive CPU version while remaining O(n). Although this book will largely ignore that kind of optimization, speedups like this matter.

**Quadratic Time**

More commonly referred to as **n squared**, this time complexity refers to an algorithm that takes time proportional to the square of the input size. Consider the following code:

void printMoreNames(List<String> names) {

for (final \_ in names) {

for (final name in names) {

print(name);

}

}

}

This time, the function prints out all the names in the list for every name in the list. If you have a list with ten pieces of data, it will print the full list of ten names ten times. That’s 100 print statements.

If you increase the input size by one, it will print the full list of eleven names eleven times, resulting in 121 print statements. Unlike the previous function, which operates in linear time, the **n** squared algorithm can quickly run out of control as the data size increases.

Here’s a graph illustrating this behavior:

DataTimeO(n )2Quadratic time

As the size of the input data increases, the amount of time it takes for the algorithm to run increases drastically. Thus, **n** squared algorithms don’t perform well at scale.

The Big O notation for quadratic time is **O(n²)**.

**Note**: No matter how inefficiently a linear time O(n) algorithm is written, for a sufficiently large **n**, the linear time algorithm will execute faster than a super optimized quadratic algorithm. Always. Every time.

**Logarithmic Time**

So far, you’ve learned about the linear and quadratic time complexities where each element of the input is inspected at least once. However, there are scenarios in which only a subset of the input needs to be inspected, leading to a faster runtime.

Algorithms that belong to this category of time complexity can leverage some shortcuts by making some assumptions about the input data.

For instance, if you had a **sorted** list of integers, what’s the quickest way to find if a particular value exists? A naive solution would be to inspect the list from start to finish before reaching a conclusion. The following is an example of this:

const numbers = [1, 3, 56, 66, 68, 80, 99, 105, 450];

bool naiveContains(int value, List<int> list) {

for (final element in list) {

if (element == value) return true;

}

return false;

}

Since you’re inspecting each of the elements once, that would be an O(n) algorithm.

**Note**: You might be thinking, “Hey, if the value that I’m searching for is at the beginning of the list, then the algorithm can exit early. Isn’t that O(1) or at least better than O(n)?”

Big O notation always tells you the **worst-case scenario**. While it’s possible that the algorithm above could finish immediately, it’s also possible that you would have to check every element. While you might think that looking at the worst case is a pessimistic way to view the world, it’s also very helpful because you know it can’t get any worse than that. And once you know the worst case, you can try to improve the algorithm.

Linear time is fairly good, but you can do better. Since the input list is sorted, there’s an optimization you can make.

If you were checking whether the number 451 existed in the list, the naive algorithm would have to iterate from beginning to end, making a total of nine inspections for the nine values in the list. However, since the list is sorted, you can, right off the bat, drop half of the comparisons necessary by checking the middle value:

bool betterNaiveContains(int value, List<int> list) {

if (list.isEmpty) return false;

final middleIndex = list.length ~/ 2;

if (value > list[middleIndex]) {

for (var i = middleIndex; i < list.length; i++) {

if (list[i] == value) return true;

}

} else {

for (var i = middleIndex; i >= 0; i--) {

if (list[i] == value) return true;

}

}

return false;

}

The algorithm first checks the middle value to see how it compares with the desired value. If the middle value is bigger than the desired value, the algorithm won’t bother looking at the values on the right half of the list; since the list is sorted, values to the right of the middle value can only get bigger. In the other case, if the middle value is smaller than the desired value, the algorithm won’t look at the left side of the list. This optimization cuts the number of comparisons by half.

What if you could perform this optimization repeatedly throughout this function? You’ll learn how to do that in Chapter 12, “Binary Search”.

An algorithm that can repeatedly drop half of the required comparisons will have **logarithmic time complexity**. Here’s a graph depicting how a logarithmic time algorithm behaves:

DataTimeO(log n) Logarithmic time

As input data increases, the time it takes to execute the algorithm increases at a slow rate. If you look closely, you may notice that the graph seems to exhibit asymptotic behavior. This can be explained by considering the impact of halving the number of comparisons you need to do.

When you have an input size of 100, halving the comparisons means you save 50 comparisons. If input size is 100,000, halving the comparisons means you save 50,000 comparisons. The more data you have, the more the halving effect scales. Thus, you can see that the graph appears to approach horizontal.

Algorithms in this category are few but extremely powerful in situations that allow for it. The Big O notation for logarithmic time complexity is **O(log n)**.

**Note**: Is it log base 2, log base 10, or the natural log?

In the above example, log base 2 applies. However, since Big O notation only concerns itself with the shape of the performance, the actual base doesn’t matter.

# [how to calculate binary search complexity](https://stackoverflow.com/questions/8185079/how-to-calculate-binary-search-complexity)

Here a more mathematical way of seeing it, though not really complicated. IMO much clearer as informal ones:

The question is, how many times can you divide N by 2 until you have 1? This is essentially saying, do a binary search (half the elements) until you found it. In a formula this would be this:

1 = N / 2x

multiply by 2x:

2x = N

now do the log2:

log2(2x)    = log2 N  
x \* log2(2) = log2 N  
x \* 1         = log2 N

this means you can divide log N times until you have everything divided. Which means you have to divide log N ("do the binary search step") until you found your element.

**Quasilinear Time**

Another common time complexity you’ll encounter is **quasilinear time**. Quasilinear time algorithms perform worse than linear time but dramatically better than quadratic time. You can think of **quasi**-linear as “kind of” like linear time for large data sets. An example of a quasilinear time algorithm is Dart’s sort method.

**Note**: At the time of this writing, the List.sort algorithm in Dart internally uses the quasilinear Dual-Pivot Quicksort algorithm for large lists. However, for lists below a size threshold of 32, it uses an insertion sort.

The Big-O notation for quasilinear time complexity is **O(n log n)**, which is a multiplication of linear and logarithmic time. Here’s the graph:

DataTimeO(n log n)Quasilinear time

Quasilinear time complexity nears a linear slope at higher values. This makes it more resilient to large data sets.

**Comparing Time Complexities**

Take a look at how the time complexities compare to each other for large data sets:

DataTimeO(n log n) O(1) O(n) O(log n) Five major time complexities

Remembering how these curves relate to each other will help you compare the efficiency of various algorithms.

**Other Time Complexities**

The five time complexities you’ve encountered so far are the ones you’ll deal with in this book. Other time complexities do exist but are far less common and tackle more complex problems that are not covered in this book. These time complexities include:

* **O(nᵏ)**: polynomial time.
* **O(2ⁿ)**: exponential time.
* **O(n!)**: factorial time.

And there are many more.

It’s important to note that time complexity is a high-level overview of performance, and it doesn’t judge the speed of the algorithm beyond the general ranking scheme. This means that two algorithms can have the same time complexity, but one may still be much faster than the other. For small data sets, time complexity may not be an accurate measure of actual speed.

For instance, quadratic algorithms such as insertion sort can be faster than quasilinear algorithms such as mergesort if the data set is small. This is because insertion sort doesn’t need to allocate extra memory to perform the algorithm, while mergesort does. For small data sets, the memory allocation can be expensive relative to the number of elements the algorithm needs to touch.

**Improving Algorithm Performance**

Suppose you wrote the following code that finds the sum of numbers from 1 to **n**.

int sumFromOneTo(int n) {

var sum = 0;

for (var i = 1; i <= n; i++) {

sum += i;

}

return sum;

}

sumFromOneTo(10000);

The code loops 10000 times and returns 50005000. It’s O(n) and will take a moment to run as it counts through the loop and prints results.

If you’re curious about how long it takes to run on your machine, you can measure it like so:

final start = DateTime.now();

final sum = sumFromOneTo(10000);

final end = DateTime.now();

final time = end.difference(start);

print(sum);

print(time);

Try increasing the input value to see how that affects the computation time.

Now try the following implementation:

int betterSumFromOneTo(int n) {

return n \* (n + 1) ~/ 2;

}

This version of the function uses a trick that Fredrick Gauss noticed in elementary school. Namely, you can compute the sum using simple arithmetic. This final version of the algorithm is O(1) and tough to beat. A constant time algorithm is always preferred. If you ran betterSumFromOneTo in a loop, you only end up with linear time. The previous O(n) version was just one outer loop away from slow, quadratic time.

**Space Complexity**

The time complexity of an algorithm can help predict scalability, but it isn’t the only metric. **Space complexity** is a measure of the memory required for an algorithm to run.

Consider the following code:

int multiply(int a, int b) {

return a \* b;

}

To perform this simple algorithm, Dart needs to allocate space for the two input parameters, a and b, as well as space for the return value. The actual size that Dart allocates internally depends on the implementation details and where the code is running, but whatever the case it’s still a fixed amount of space. Even for very large input values, the return value will just overflow; it won’t take more space. That means the space complexity for this algorithm is **constant**, and so the Big O notation is O(1).

However, now take a look at this example:

List<String> fillList(int length) {

return List.filled(length, 'a');

}

This algorithm creates a list filled with the string 'a'. The larger length is, the longer the list will be and thus the more space will be required to store the list in memory. Since the space increases proportionally with the input value, the space complexity of this algorithm is **linear** and the Big O notation is O(n).

With one small change you could make that algorithm have **quadratic** space complexity:

List<String> stuffList(int length) {

return List.filled(length, 'a' \* length);

}

Not only do larger values for length make the list longer, they also increase the size of the string in each element of the list. Specifying 5 for length would create a list of length 5 whose elements are 'aaaaa'. As with quadratic time complexity, the Big O notation for quadratic space complexity is O(n²).

**Other Notations**

So far, you’ve evaluated algorithms using Big O notation, which tells the worst case runtime. This is by far the most common measurement that programmers evaluate with. However, other notations exist as well:

* **Big Omega notation** is used to measure the best-case runtime for an algorithm. This isn’t as useful as Big O because getting the best case is often untenable.
* **Big Theta notation** is used to measure the runtime for an algorithm that is between the best- and worse-case scenarios.

**Key Points**

* **Time complexity** is a measure of the time required to run an algorithm as the input size increases.
* You should know about constant time, logarithmic time, linear time, quadratic time and quasilinear time and be able to order them by cost.
* **Space complexity** is a measure of the memory required for the algorithm to run.
* **Big O notation** is used to represent the general form of time and space complexity.
* Time and space complexity are high-level measures of scalability; they do not measure the actual speed of the algorithm itself.
* For small data sets, time complexity is usually irrelevant. A quasilinear algorithm can be slower than a quadratic algorithm.