2 Complexity Written by Márton Braun

How well will it scale?

This question is always asked sooner or later in the software development cycle and comes in several flavors.

From an architectural perspective, scalability refers to how flexible your app is as your features are increasing. From a database perspective, scalability is about the capability of a database to handle an increasing amount of data and users. For a web server, being scalable can mean that it can serve a high number of users accessing it at the same time. Regardless of what the question actually means, you need to study it and come up with a response as soon as possible. This way, you can avoid big problems down the line.

For algorithms, scalability refers to how the algorithm performs in terms of execution time and memory usage as the input size increases. With a small amount of data, any algorithm may still feel fast. However, as the amount of data increases, an expensive algorithm can become crippling.

So how bad can it get? Estimating this is an important skill for you to know.

In this chapter, you'll learn about the Big O notation for the different levels of scalability in two dimensions:

- Execution time.
- Memory usage.

Time complexity

With 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.

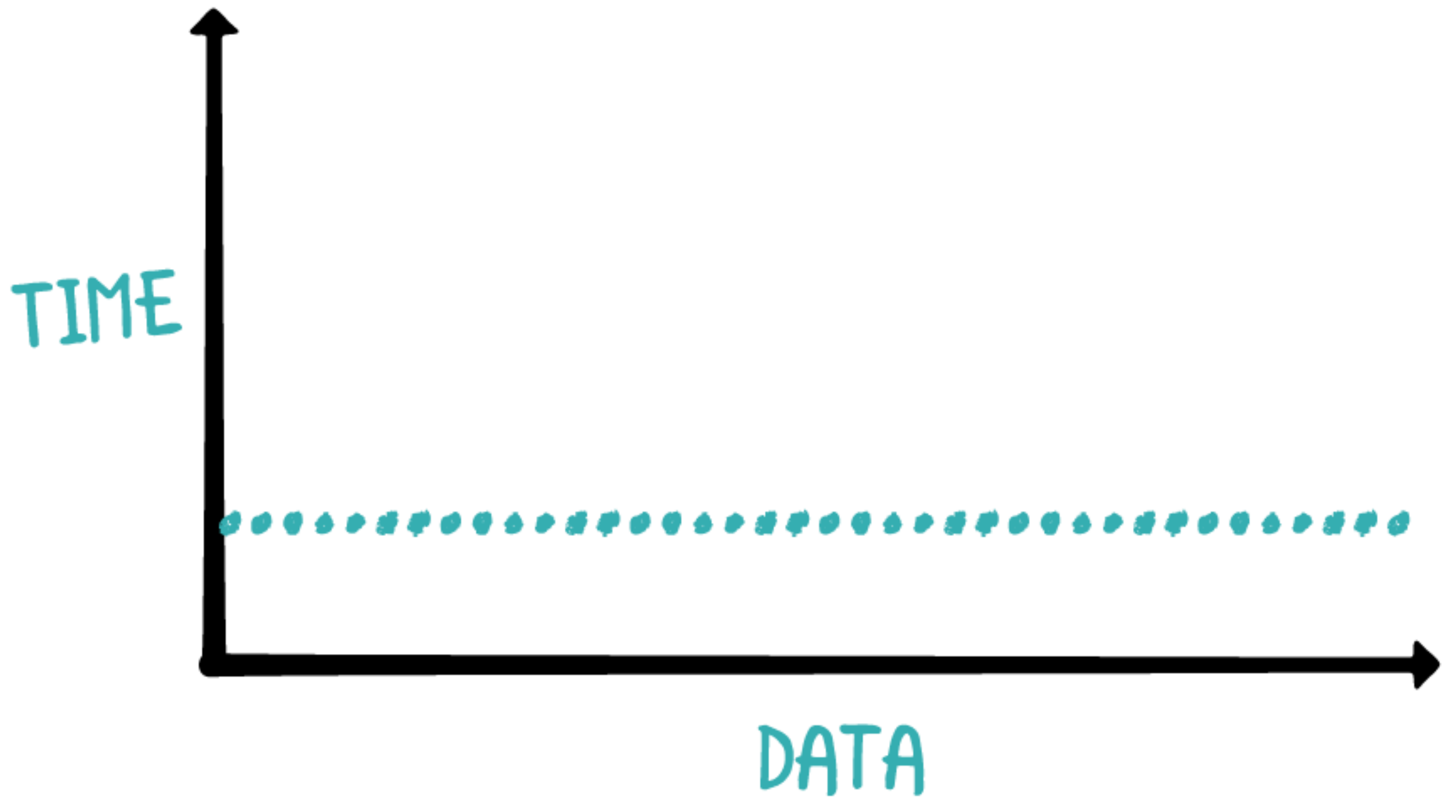
Constant time

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

```
fun checkFirst(names: List<String>) {
  if (names.firstOrNull() != null) {
    println(names.first())
  } else {
    println("no names")
  }
}
```

The size of `names` does not affect the running time of this function. Whether `names` has 10 items or 10 million items, this function only checks the first element of the list.

Here's a visualization of this time complexity in a plot between time versus data size:



Constant time

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

For brevity, programmers use a notation known as **Big O notation** to represent various magnitudes of time complexity. The Big O notation for constant time is *O*(*1*). It's one unit of time, regardless of the input. This time doesn't need to be small, though. The algorithm can still be slow, but it's equally slow all of the time. :]

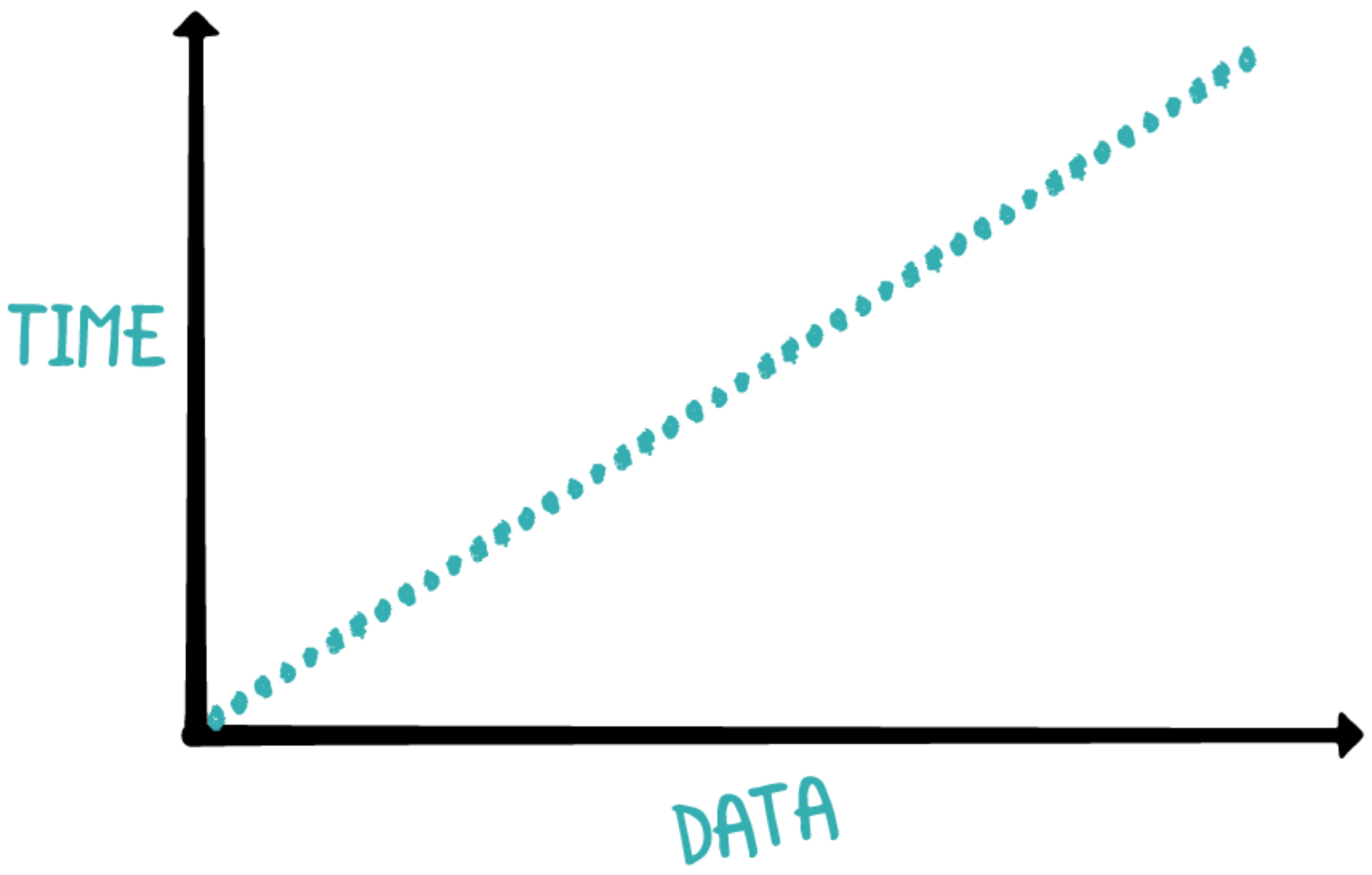
Linear Time

Consider the following snippet of code:

```
fun printNames(names: List<String>) {
  for (name in names) {
    println(name)
  }
}
```

This function prints all the names in a `String` list. As the input list increases in size, the number of iterations is increased by the same amount.

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



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 of the data and a calls six different $O(1)$ methods? Is it $O(2n + 6)$?
>
> Time complexity only gives a high-level shape of the performance. Loops that happen a set number of times are not part of the calculation. You'll need to abstract everything and consider only the most important thing that affects performance. All constants are dropped in the final Big O notation. In other words, $O(2n + 6)$ is surprisingly equal to $O(n)$.

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:

```
fun multiplicationTable(size: Int) {
  for (number in 1..size) {
    print(" | ")
    for (otherNumber in 1..size) {
      print("$number x $otherNumber = ${number * otherNumber} | ")
    }
    println()
  }
}
```

If you call this function using a small number, like 2, you'll get the following output:

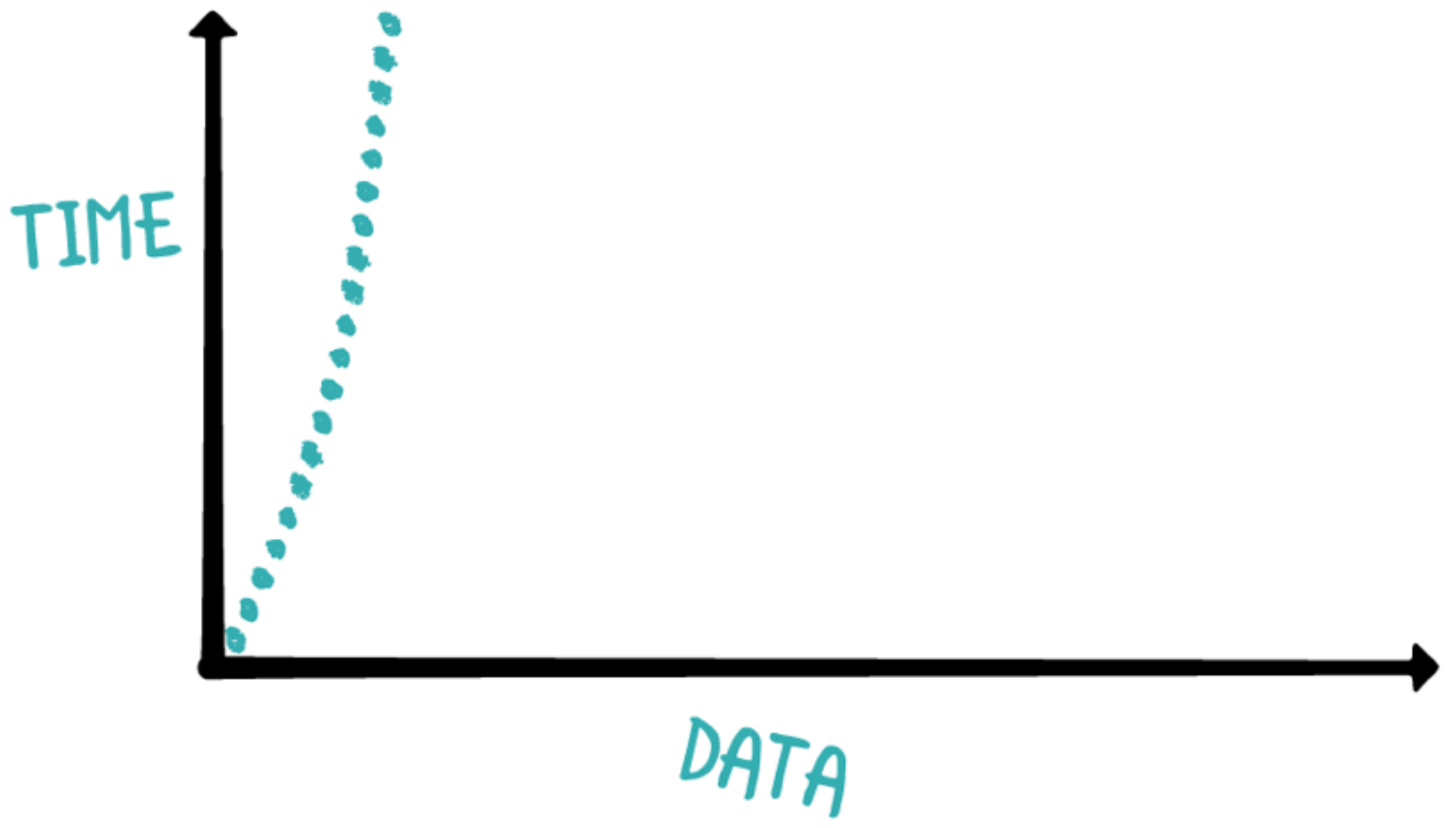
```
 | 1 x 1 = 1 | 1 x 2 = 2 |
 | 2 x 1 = 2 | 2 x 2 = 4 |
```

This time, the function prints all products of the numbers that are less than or equal to the input, starting with 1.

If the input is 10, it'll print the full multiplication board of 10 × 10. That's 100 print statements. If you increase the input size by one, it'll print the product of 11 numbers with 11 numbers, 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. Imagine printing the results for `multiplicationTable(100_000)`!

Here's a graph illustrating this behavior:



Quadratic 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*(*n*^2).

> **Note**: No matter how inefficiently a linear time *O(n)* algorithm is written, for a sufficiently large *n*, the linear time algorithm will always execute faster than a super optimized quadratic algorithm.
>
> Although not a central concern of this book, optimizing for absolute efficiency can be crucial.
>
> 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 100× faster than the naive CPU version while remaining *O*(*n*).

Logarithmic time

So far, you've learned about the linear and quadratic time complexities wherein 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 are ones that can leverage some shortcuts by making some assumptions about the input data. For instance, if you had a **sorted** list of integers, what is the quickest way to find if a particular value exists?

A possible solution would be to inspect the list from start to finish to check every element before reaching a conclusion. Since you're inspecting every element once, that would be a *O*(*n*) algorithm.

Linear time is fairly good, but you can do better. Since the input array is sorted, there's an optimization that you can make. Consider the following code:

```
val numbers = listOf(1, 3, 56, 66, 68, 80, 99, 105, 450)

fun linearContains(value: Int, numbers: List<Int>): Boolean {
  for (element in numbers) {
    if (element == value) {
      return true
    }
  }
  return false
}
```

If you were checking if the number 451 existed in the list, this algorithm would have to iterate from the 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:

```
fun pseudoBinaryContains(value: Int, numbers: List<Int>): Boolean {
  if (numbers.isEmpty()) return false
  val middleIndex = numbers.size / 2

  if (value <= numbers[middleIndex]) {
    for (index in 0..middleIndex) {
      if (numbers[index] == value) {
        return true
      }
    }
  } else {
    for (index in middleIndex until numbers.size) {
      if (numbers[index] == value) {
        return true
      }
    }
  }
  return false
}
```

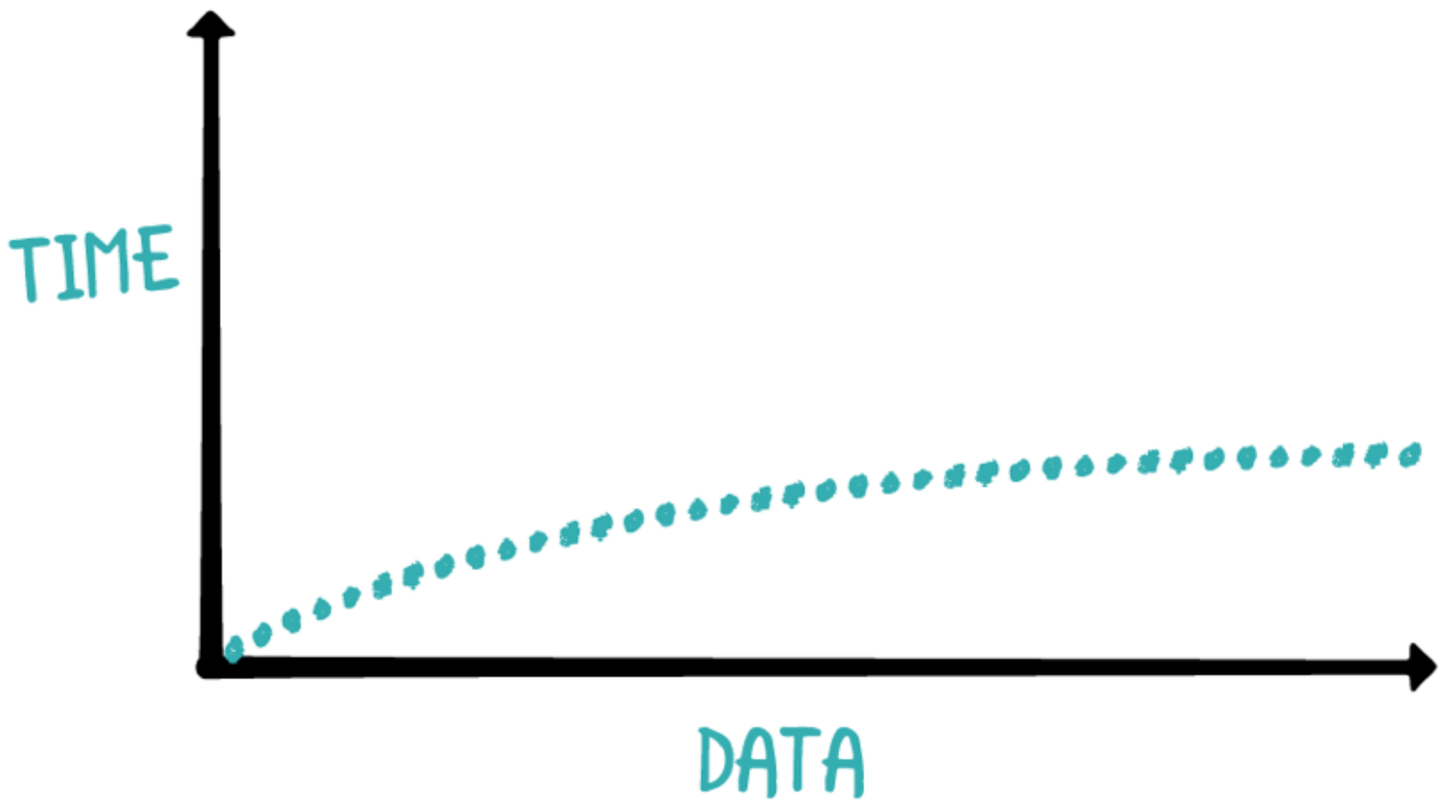
The above function makes a small but meaningful optimization wherein it only checks half of the list to come up with a conclusion.

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 do this optimization repeatedly throughout this method? You'll find out in Chapter 11, "Binary Search".

An algorithm that can repeatedly drop half of the required comparisons will have logarithmic time complexity. Here's a graph illustrating how a logarithmic time algorithm would behave as input data increases:



Logarithmic time

As input data increases, the time it takes to execute the algorithm increases at a slower 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 the input size was 10,000, halving the comparisons means you save 5,000 comparisons.

You'll need just a few more halvings, and your input data will be around 50 again. The more data you have, the more the halving effect scales.

Algorithms in this category are few but are 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. The more input data you can drop after

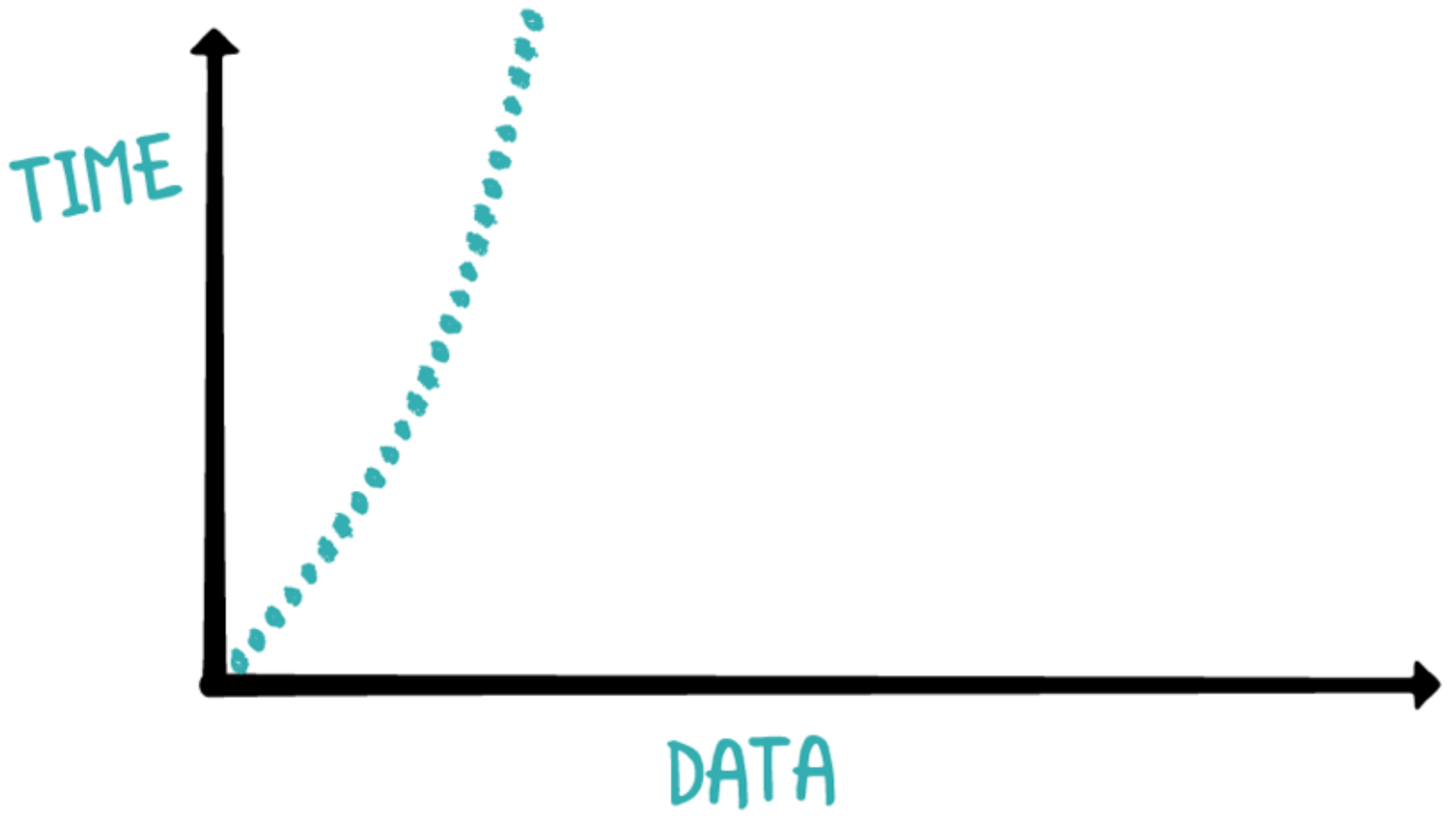
> each pass, the faster the algorithm will be.

Quasilinear time

Another common time complexity you'll encounter is **quasilinear time**. Algorithms in this category perform worse than linear time but dramatically better than quadratic time. They are among the most common algorithms you'll deal with.

An example of a quasilinear time algorithm is Kotlin's `sort` method.

The Big-O notation for quasilinear time complexity is *O*(*n log n*) which is a multiplication of linear and logarithmic time. So quasilinear fits between logarithmic and linear time. It's a magnitude worse than linear time but still better than many of the other complexities that you'll see next. Here's the graph:



Quasilinear time

The quasilinear time complexity shares a similar curve with quadratic time. The key difference is that quasilinear complexity is more resilient to large data sets.

Other time complexities

The five complexities you've encountered are the ones that you'll encounter in this book. Other time complexities do exist, but are far less common and tackle more complex problems that are not discussed in this book. These time complexities include **polynomial time**, **exponential time**, **factorial time** and 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 the insertion sort does not need to allocate extra memory to perform the algorithm, while mergesort needs to allocate multiple arrays.

> **Note**: For small data sets, the memory allocation can be expensive relative to the number of elements the algorithm needs to touch.

Comparing time complexity

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

```
fun sumFromOne(n: Int): Int {
  var result = 0
  for (i in 1..n) {
    result += i
  }
  return result
}
```

If you try to call the function with `sumFromOne(10000)`, the code loops 10,000 times and returns `50005000`. It's *O*(*n*) and will take a moment to run as it counts through the loop and prints results.

This can also be written using `reduce`:

```
fun sumFromOne(n: Int): Int {
  return (1..n).reduce { sum, element -> sum + element }
}
```

The time complexity of the version that uses `reduce` is also O(n) since it essentially performs the same logic. It continuously adds each element to the sum and returns the total sum.

Finally, you can write:

```
fun sumFromOne(n: Int): Int {
  return n * (n + 1) / 2
}
```

This version of the function uses a trick that a famous mathematician, **Fredrick Gauss**, noticed while he was still in elementary school. The sum of a series of numbers starting from *1* up to *n* can be computed using simple arithmetic. This final version of the algorithm is *O*(*1*) is and tough to beat. A constant time algorithm is always preferred over a linear or logarithmic time algorithm since the time it takes to run will not change regardless of how large *n* gets.

Space complexity

The time complexity of an algorithm isn't the only performance metric against which algorithms are ranked. Another important metric is its space complexity, which is a measure of the amount of memory it uses.

Consider the following code:

```
fun printSorted(numbers: List<Int>) {
  val sorted = numbers.sorted()
  for (element in sorted) {
    println(element)
  }
}
```

The above function creates a sorted copy of the list and prints it. To calculate the space complexity, you analyze the amount of memory the function allocates.

Since `numbers.sorted()` produces a new list with the same size of `numbers`, the space complexity of `printSorted` is *O*(*n*). While this function is simple and elegant, there may be some situations in which you want to allocate as little memory as possible.

You could rewrite the above function like this:

```
fun printSorted(numbers: List<Int>) {
  // 1
  if (numbers.isEmpty()) return

  // 2
  var currentCount = 0
  var minValue = Int.MIN_VALUE

  // 3
  for (value in numbers) {
    if (value == minValue) {
      println(value)
      currentCount += 1
    }
  }

  while (currentCount < numbers.size) {
    // 4
    var currentValue = numbers.maxOrNull()!!
    for (value in numbers) {
      if (value < currentValue && value > minValue) {
        currentValue = value
      }
    }

    // 5
    for (value in numbers) {
      if (value == currentValue) {
        println(value)
        currentCount += 1
      }
    }

    // 6
    minValue = currentValue
  }
}
```

Woah, that’s a lot of code for something you’ve previously done in a couple of lines! But this implementation respects space constraints.

The overall goal is to iterate through the array multiple times, printing the next smallest value for each iteration.

Here’s what this algorithm is doing:

1. Check for the case if the list is empty. If it is, there’s nothing to print.
2. `currentCount` keeps track of the number of print statements made. `minValue` stores the last printed value.
3. The algorithm begins by printing all values matching the `minValue` and updates the `currentCount` according to the number of print statements made.
4. Using the `while` loop, the algorithm finds the lowest value bigger than `minValue` and stores it in `currentValue`.
5. The algorithm then prints all values of `currentValue` inside the array

while updating `currentCount`.

6. `minValue` is set to `currentValue`, so the next iteration will try to find the next minimum value.

The above algorithm only allocates memory for a few variables. Since the amount of memory allocated is constant and does not depend on the size of the list, the space complexity is $O(1)$.

This is in contrast with the previous function, which allocates an entire list to create the sorted representation of the source array. The tradeoff here is that you sacrifice time and code readability to use as little memory is possible.

> **Note**: While today's devices have a lot of space to store information, there was a time when everything an algorithm needed to store was bound to just a couple of KB.
>
> Algorithms could be designed to take a longer time to finish but the memory constraint was a physical one, and it could not be broken.
>
> Nowadays, the available memory is huge but so is the data you are handling, so algorithms still need to take space complexity into account.

Key points

- **Time complexity** is a measure of the time required to run an algorithm as the input size increases.
- **Space complexity** is a measure of the resources required for the algorithm to manipulate the input data.
- **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 don't 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 linear algorithm.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.