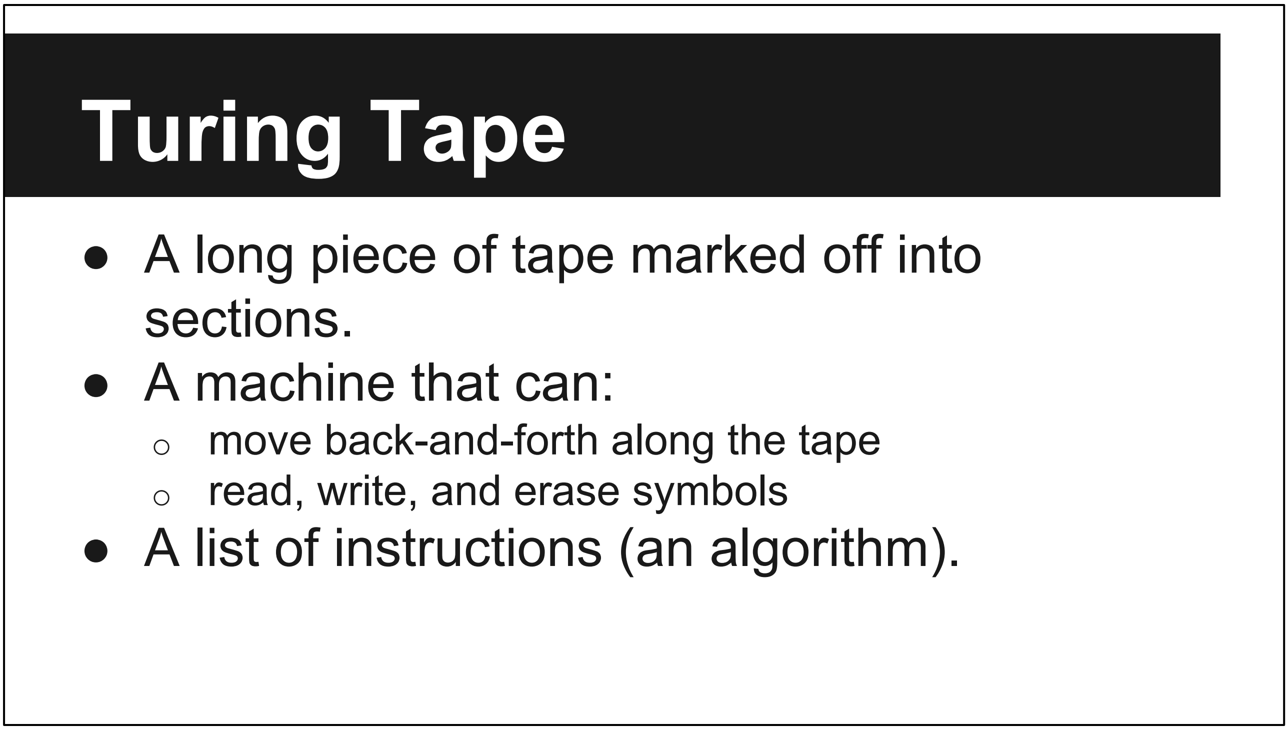
Intro to Computer Hardware and Performance

Prerequisites

- Ruby Data Types

What you’ll learn

- How a computer really works.
- What software performance is.
- How some Ruby code performs.

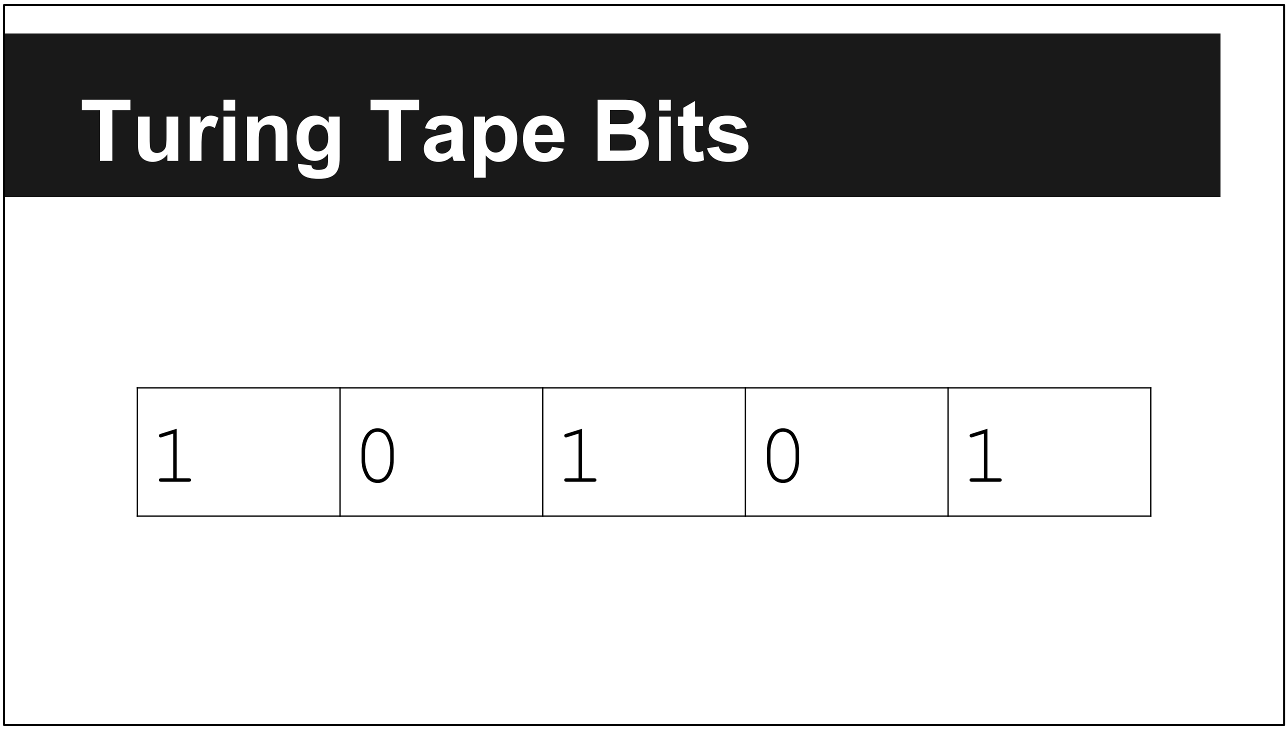


Turing Tape

- A long piece of tape marked off into sections.
- A machine that can:
  - move back-and-forth along the tape
  - read, write, and erase symbols
- A list of instructions (an algorithm).

You’ve already learned how a theoretical computer works. All you need to be a computer is:

- a long piece of tape marked off into sections
- a machine that can
  - move back-and-forth along the tape
  - read, write, and erase symbols
- a list of instructions (an algorithm)

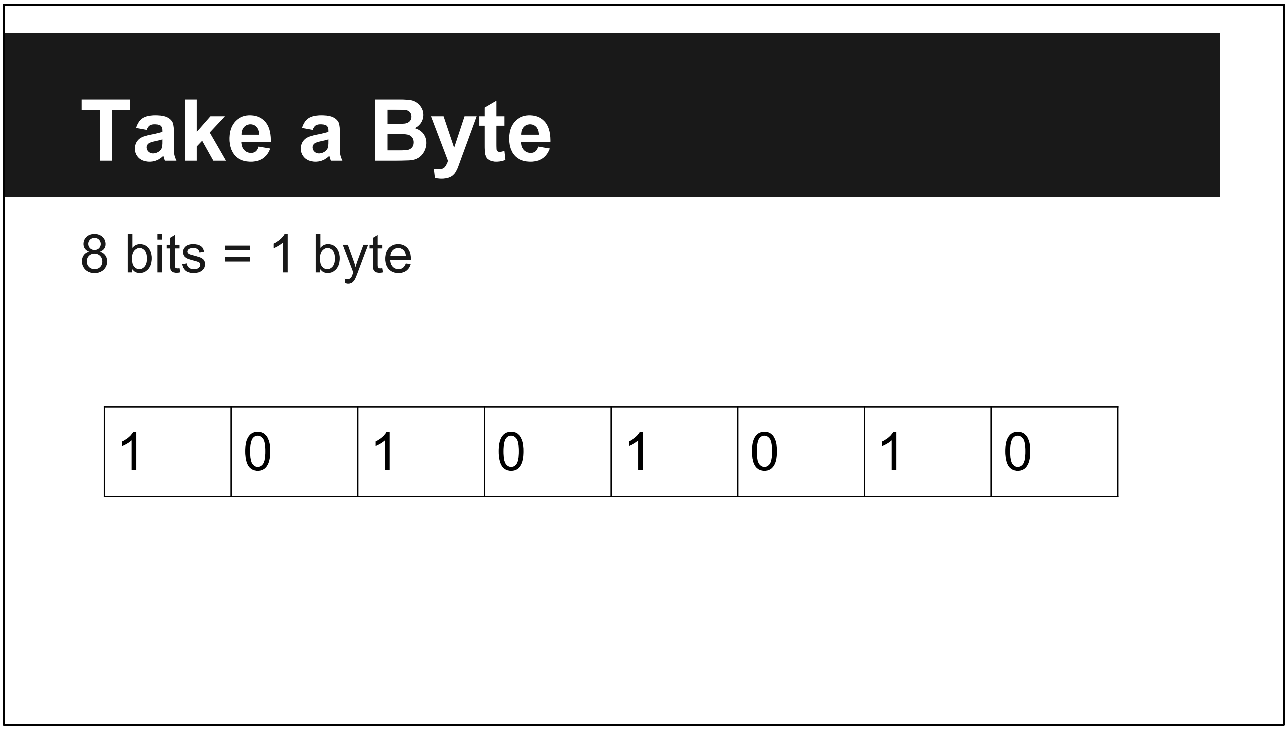


Turing Tape Bits

1	0	1	0	1

Remember also that everything in a computer must be numbers. So, when we talk about software, instead of dots and spaces, we refer to 1’s and 0’s.

A single digit, which can be 1 or 0, is called a bit.



Take a Byte

8 bits = 1 byte

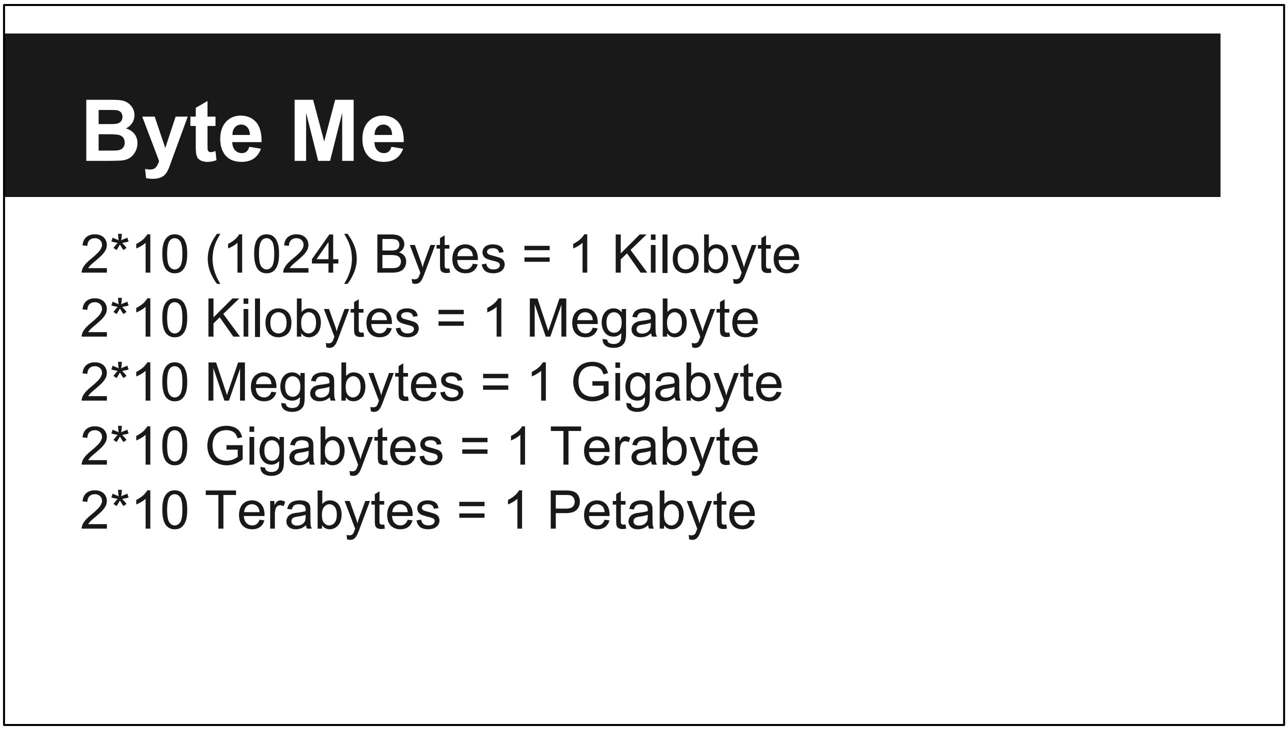
1	0	1	0	1	0	1	0

Bits are counted in chunks of 8. So a byte represents 8-bits.

Why 8? Because a bit can be one of two values, zero or one. So it’s a binary number system.

We’re used to a 10-digit number system, a “decimal” system. So when we count, we naturally prefer to chunk in powers of 10. 10^1 (10), 10^2 (100), and so on.

In a binary world, binary people prefer to chunk in powers of 2. So 2^2 (4), 2^3 (8), 2^4 (16), etc.



Byte Me

2*10 (1024) Bytes = 1 Kilobyte
2*10 Kilobytes = 1 Megabyte
2*10 Megabytes = 1 Gigabyte
2*10 Gigabytes = 1 Terabyte
2*10 Terabytes = 1 Petabyte

2^{10} ends up being the primary way we divy up groups of bytes into orders of magnitude.

So there are 2^{10} bytes in a Kilobyte, for example. And so on, up to Petabyte and beyond.

2^{10} is *almost* 1000, so some people just use 1000 as an estimate. But it's not quite right.

Hardware manufacturers, for example, count by 1000s instead of 1024s. That's why your new computer never has the disk space that's printed on the box.

If you find it necessary to do some byte conversions, Google can help. Just search for "bytes in a Gigabyte" and Google will just give you the answer.

Hard Disk Drive

This a 500GB hard drive, the kind you might find in a laptop. If each bit is a square in the Turing Tape, then this drive is capable of storing nearly 4 trillion little squares. How?

Hard Disk Drive

The Turing tape in a hard drive is very, very thin and stored in a spiral pattern on a metal disk.

Hard Disk Drive

The disk spins very rapidly, and the actuator arm moves across the disk very rapidly.

[video]

If you’ve ever wondered where that clicking sound in your computer comes from, it’s this.

Hard Disk Drive

So, conceptually, it’s a lot like this. But like I said in a previous lecture, the materials are much more complex.

RAM

There are actually two Turing tapes in a computer. The hard disk drive is the longest, but it’s also the slowest.

The faster, but much shorter tape is called RAM, or random-access memory. As you can see, it looks a lot different. No metal disk. No actuator arm. These chips only store 512MB each, much less than the 500GB disk drives I showed you before.

RAM

RAM doesn't work the same as hard disks. They don't have any moving parts, so I don't have a video to show you. RAM actually works more like series of lights (like the kind you'd see around a vanity mirror). On means 1, off means 0.

Because there are no moving parts, RAM is very fast. You just ask the RAM chip "is the 3rd light on" and it'll tell you right away.

If you ask a hard disk the same question, it has to search for that 3rd box. It has to spin the disk into place and move the actuator arm above that section to read the value. This delay is made up of the "seek time" (the time to move the arm into place) and the "latency" (the time it takes to spin the disk into place).

[show this on the whiteboard]

RAM

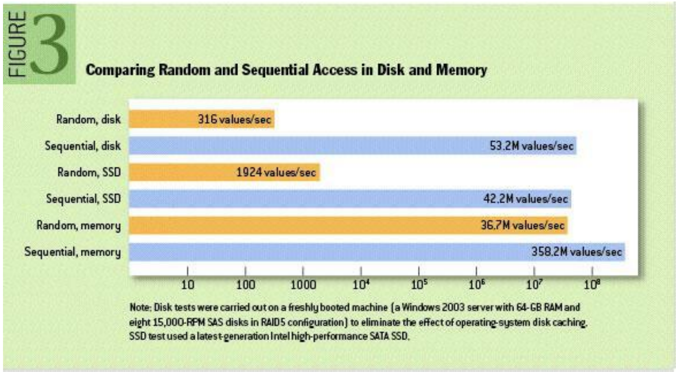


FIGURE 3

Comparing Random and Sequential Access in Disk and Memory

Note: Disk tests were carried out on a freshly booted machine (a Windows 2003 server with 64-GB RAM and eight 15,000-RPM SAS disks in RAID5 configuration) to eliminate the effect of operating-system disk caching. SSD test used a latest-generation Intel high-performance SATA SSD.

How much faster is RAM? According to this chart from 2009, much much faster.

If you want to access randomly placed Turing tape boxes, the hard disk analyzed for this chart is capable of reading 316 “values”/second. A “value” in this case is 4-bytes.

RAM is capable of accessing randomly placed bits at 36.7 *million* values/second, over *100,000 times* faster. They don’t call it random-access memory for nothing!

If you’re lucky enough to be accessing boxes sequentially (in order on the tape rather than randomly separated), then RAM is only 7 times faster.

SSD drives, a fancy new tech that combines some of the speed of RAM with the size of disk drives, helps to bridge the gap a little.

Source: http://queue.acm.org/detail.cfm?id=1563874

RAM

Another way RAM differs from hard disks is that it’s volatile. That means, much like a series of lights, if you cut the power, you lose everything. All the bits flip to 0.

So RAM is good for fast, short term memory. But if you expect to save something for the long term, you eventually need to write it to the disk.

The “Desk” Metaphor

When you combine hard disks and RAM in a computer, you get a machine that works a lot like a Desk.

The top surface is the RAM. It’s small and fast. Any paperwork you put on the table is easy to access. But it’s also volatile. If you leave something up there long enough, eventually something is going to happen to it: you’ll spill your coffee on it, your pets or kids will knock it away, the cleaning lady will throw it out. It’s not safe.

The drawers are the hard disks. There’s lots of room and it can sometimes be hard to find things. But your paperwork is tucked away, safe from your cats and your coffee.

Swapping

Your RAM can get full. When that happens, your computer's operating system will start secretly moving RAM bits to your hard disk. This is called swapping, and it's bad. Your computer will continue running, but as soon as you try to access the bits that have been moved to disk, it'll start running 7 to 100,000 times slower.

If anyone remembers those old cheap Compaq Presario or eLearning computers they used to sell at Walmart, this is why they were so terrible. They started out loaded up with so much pre-installed software that you were swapping from day 1.

Your computer won't complain if you run out of RAM. It'll just start swapping to disk and going slower. But it will complain if you run out of disk space. And, if you have enough disk space, your computer will still complain if it thinks it's swapping too much.

Space

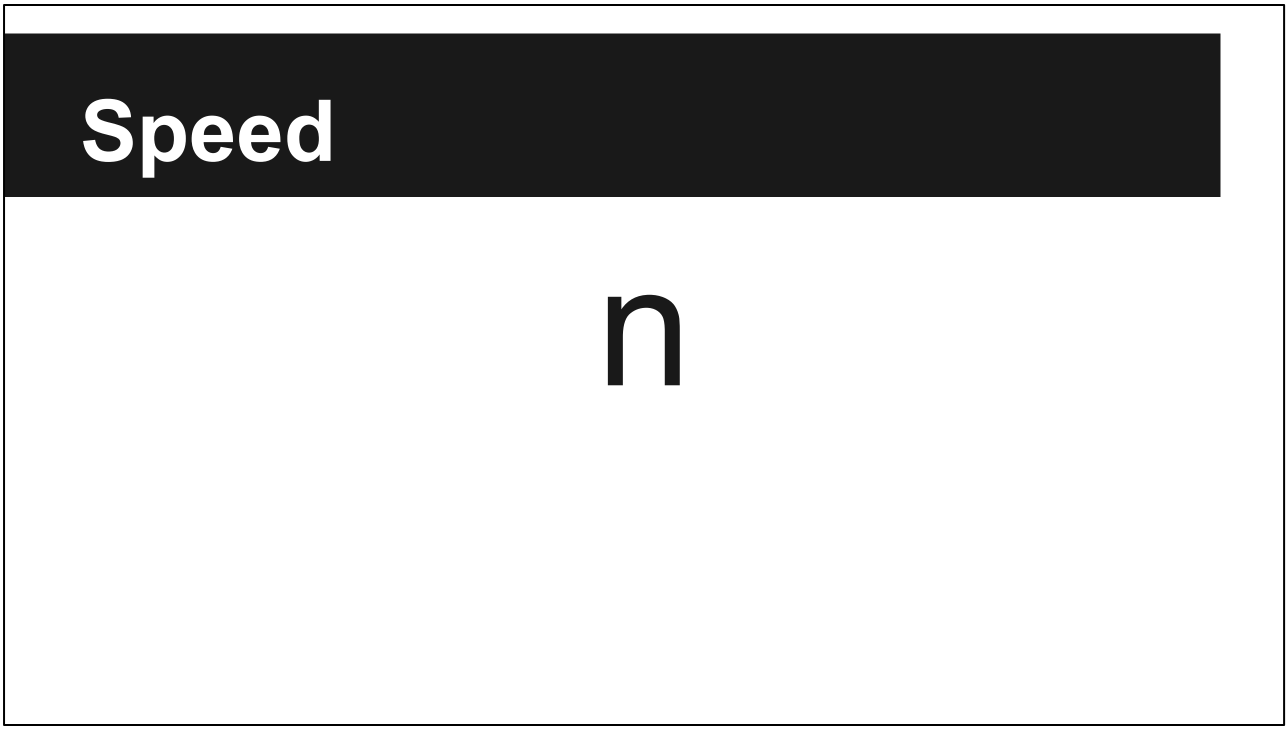
What does this have to do with Ruby?

Ruby, when it's running, takes up some RAM. The bigger your program, the more RAM you'll need. So when evaluating the performance of a Ruby application, one dimension of that performance is space. How many bits, how many Turing tape squares, do you need? In general, the less space you use, the better.

Some of that space is the number of lines of code.

Outside of Ruby itself, most of the space taken up by your program is the size of your data structures. It's the number of items in your Arrays and the number of pairs in your Hashes.

Size is related to speed. Bigger programs run slower. For example, it takes a lot of time to sort through a 1 million item Array.



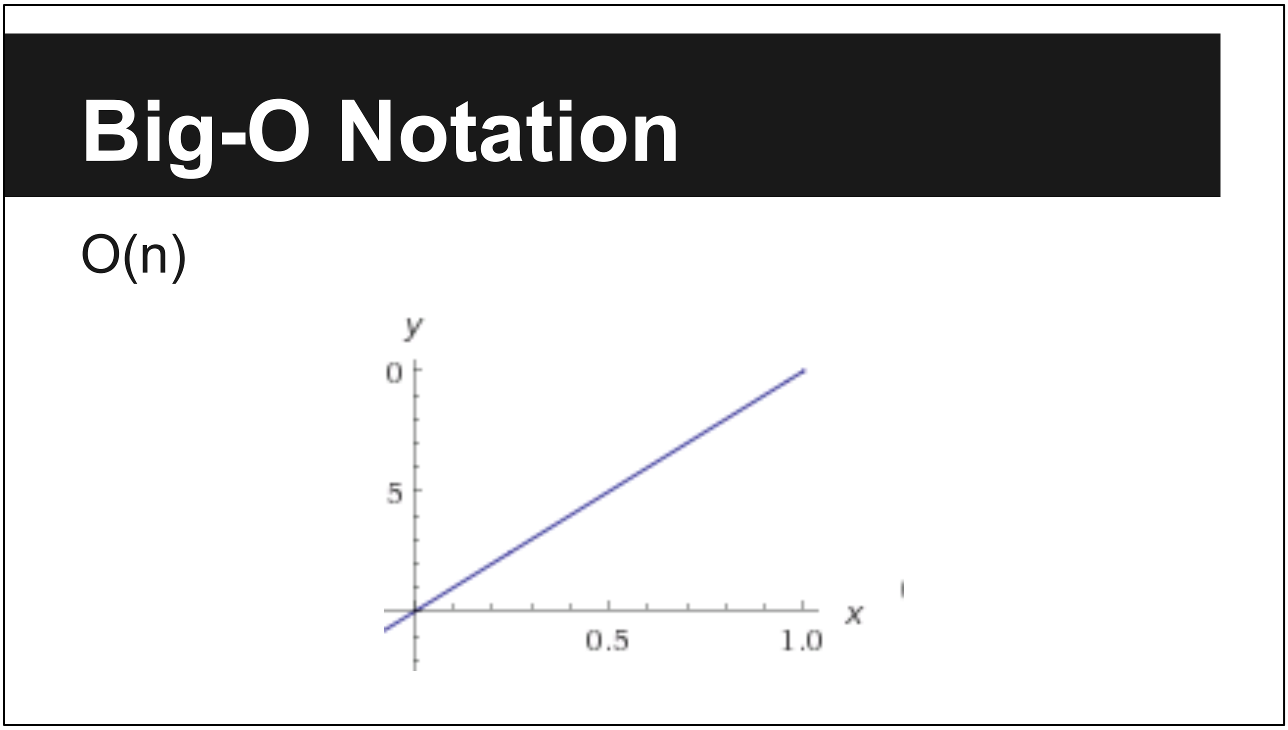
Speed

n

When talking about the performance of a program, you have to make some assumptions. The first is that you have to assume (if you can) that you're doing everything in RAM (it's a program that doesn't, e.g., actually require the disk). So you can ignore swapping and seek time and any of the details related to how the hardware works.

Second, we assume, and this is solely by convention, that the amount of data we need, the number of Turing tape squares or bits or bytes or whatever, is represented by the letter lowercase-n. We don't need to be specific. Just n.

So to describe the performance, we want to describe how fast we think a program will run given some data of size n.



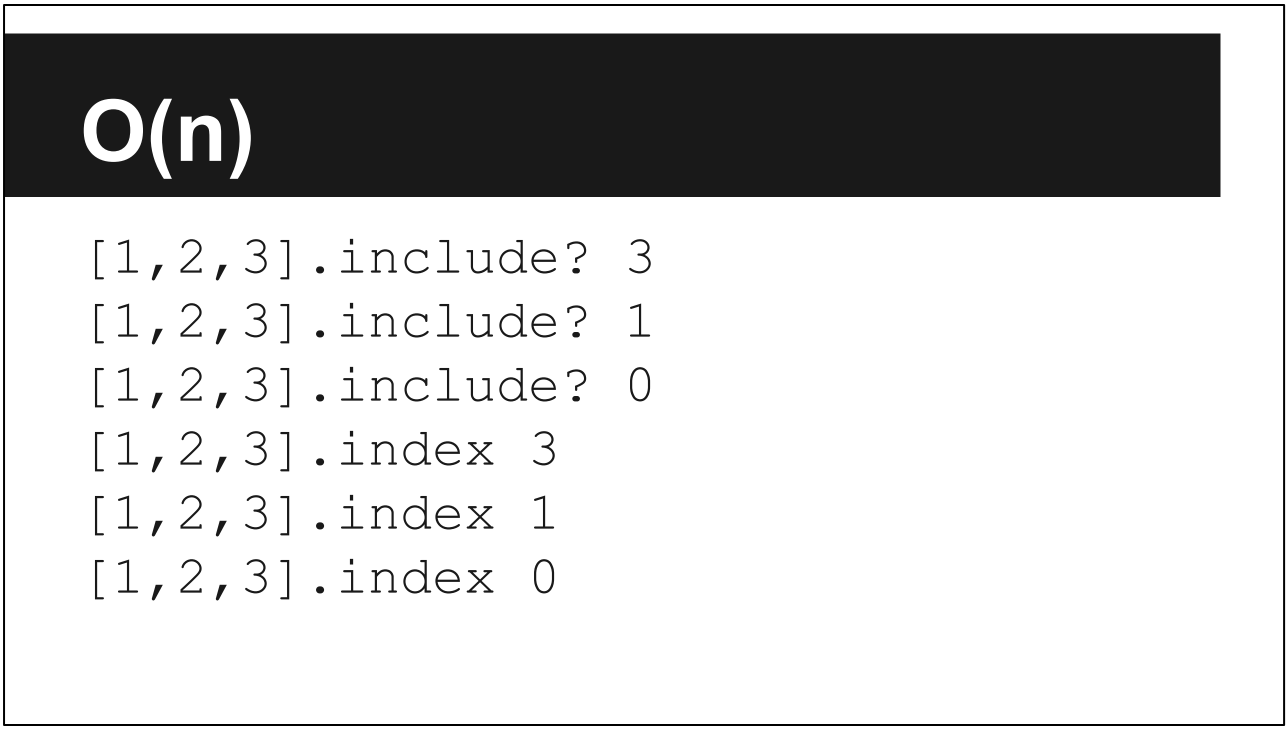
Big-O Notation

O(n)

Big-O notation is how computer programmers express to each other how fast their programs run. This isn’t a line of Ruby code. This is a line you’d write in an online forum or a job interview quiz.

If this - O, open-parens, n, close-parens - is how fast your program runs, you’d say you’re program “runs in order n” or “runs in n-time” or simply “runs in n”.

When your program runs in order-n time, that means that as the data gets bigger (x-axis) your program gets slower (y-axis). Most programs run like this.



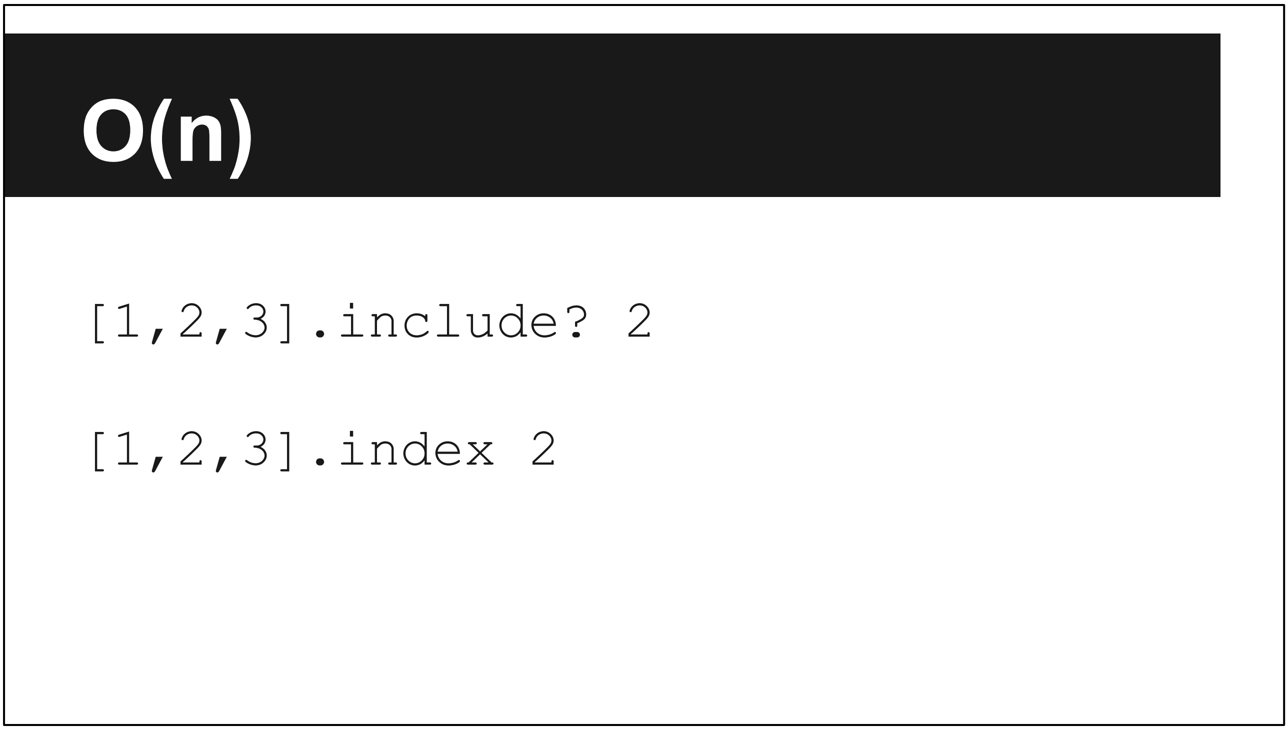
O(n)

```
[1,2,3].include? 3
[1,2,3].include? 1
[1,2,3].include? 0
[1,2,3].index 3
[1,2,3].index 1
[1,2,3].index 0
```

Some examples of Ruby expressions that run in order-n are the Array methods include? and index.

Include? tells you whether or not the argument exists in the Array. To do that, include has to search through every item in the Array, one at a time, until it finds it. If it's lucky, it's the first item. If it's not, it's the last item. If the item doesn't exist at all, it still has to check all the items. So, on average, it has to search some fraction of n, the total number of items in the Array. [board example?]

Index will give you the position of the argument in the Array. Just like include?, sometimes it's lucky, and sometimes it's not.

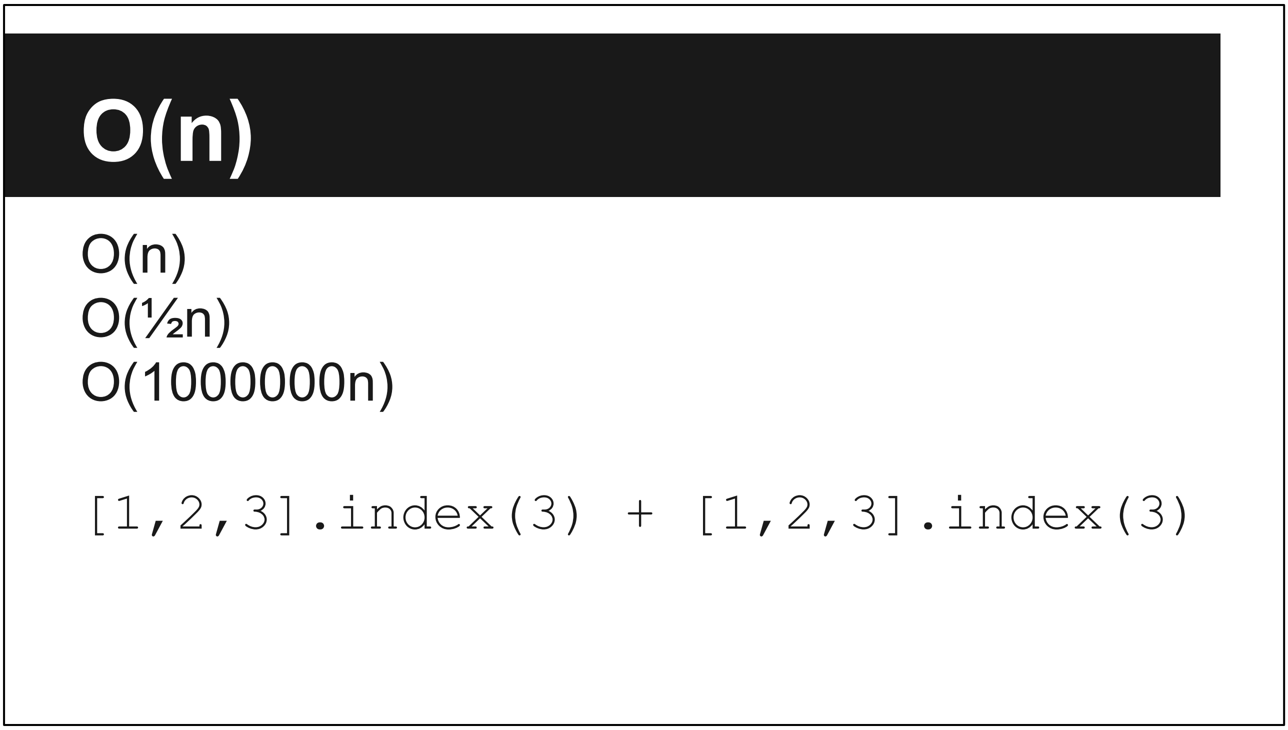


O(n)

```
[1,2,3].include? 2

[1,2,3].index 2
```

But, on average, for a very long Array, isn’t the thing you’re looking for much more likely to be in the middle than at either of the ends? Why do we say it’s n and not half-n or something like that?



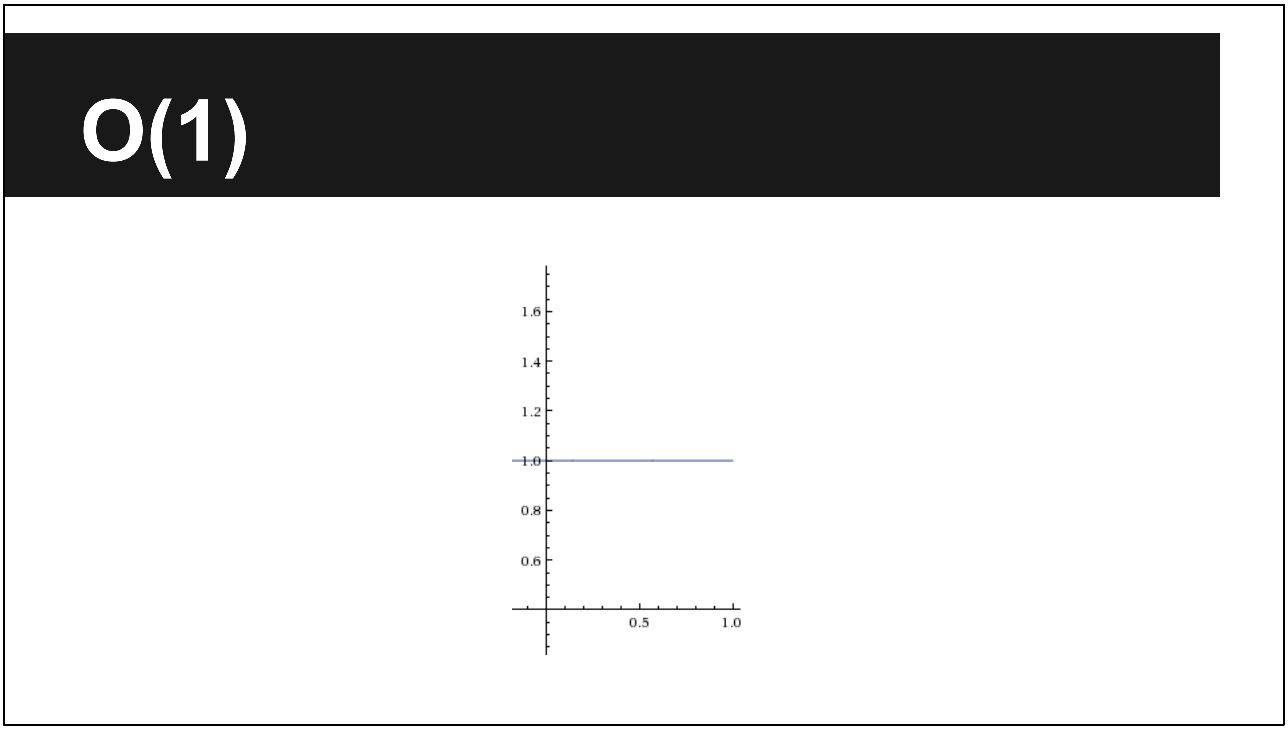
O(n)

O(n)
O(½n)
O(1000000n)

```
[1,2,3].index(3) + [1,2,3].index(3)
```

Big-O notation doesn't care about the actual number of times you have to go over the Array. It lets you to ignore that. The only thing it cares about is that, as n grows, your program gets slower in a predictable way.

So all three of these big-O notations at the top are equivalent. And the line of code at the bottom, even though it is searching through the array twice, is still considered to be running in order-n.

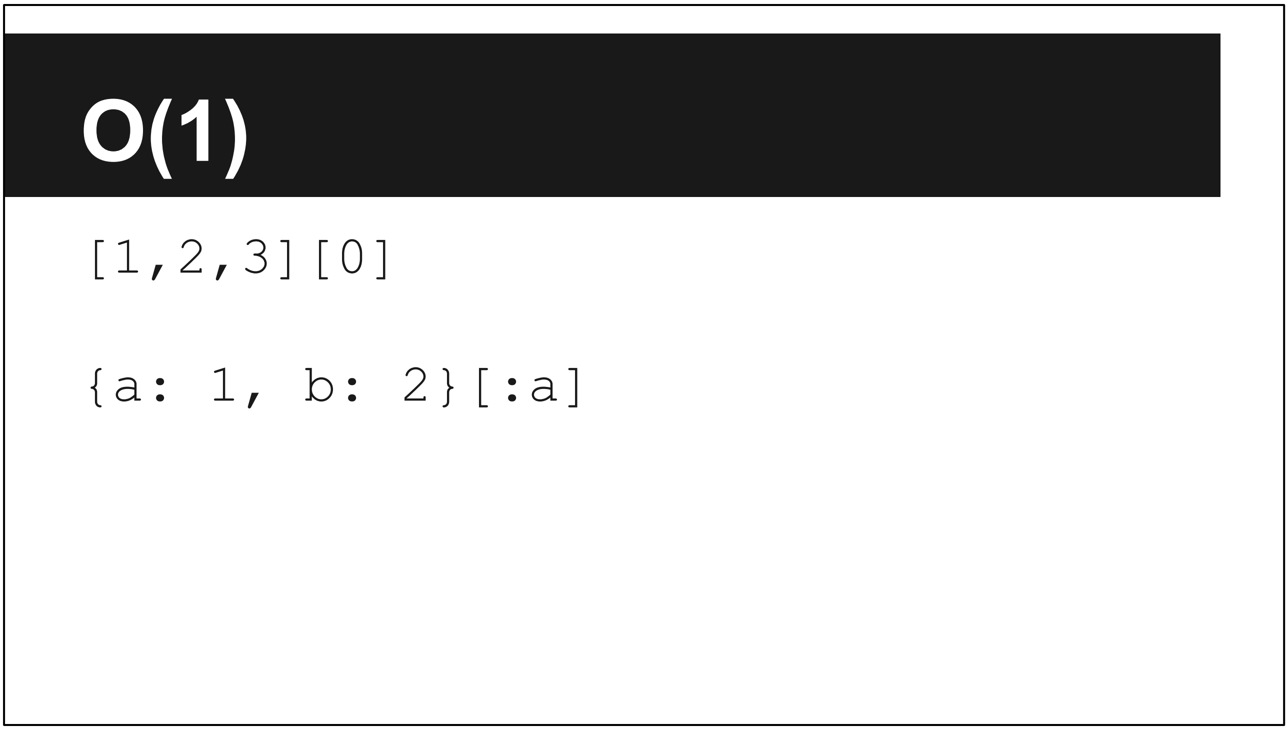


O(1)

Some algorithms are so fast they run in order-1. That means that, no matter how big the data gets, the program will run at exactly the same speed. It doesn’t matter if your Array has 3 values or 3 million.

Order-1 is also known as “constant time”, as in my program “runs in constant time”. The amount of time my program takes is constant, unchanging, regardless of how big my data is.

How is that possible?



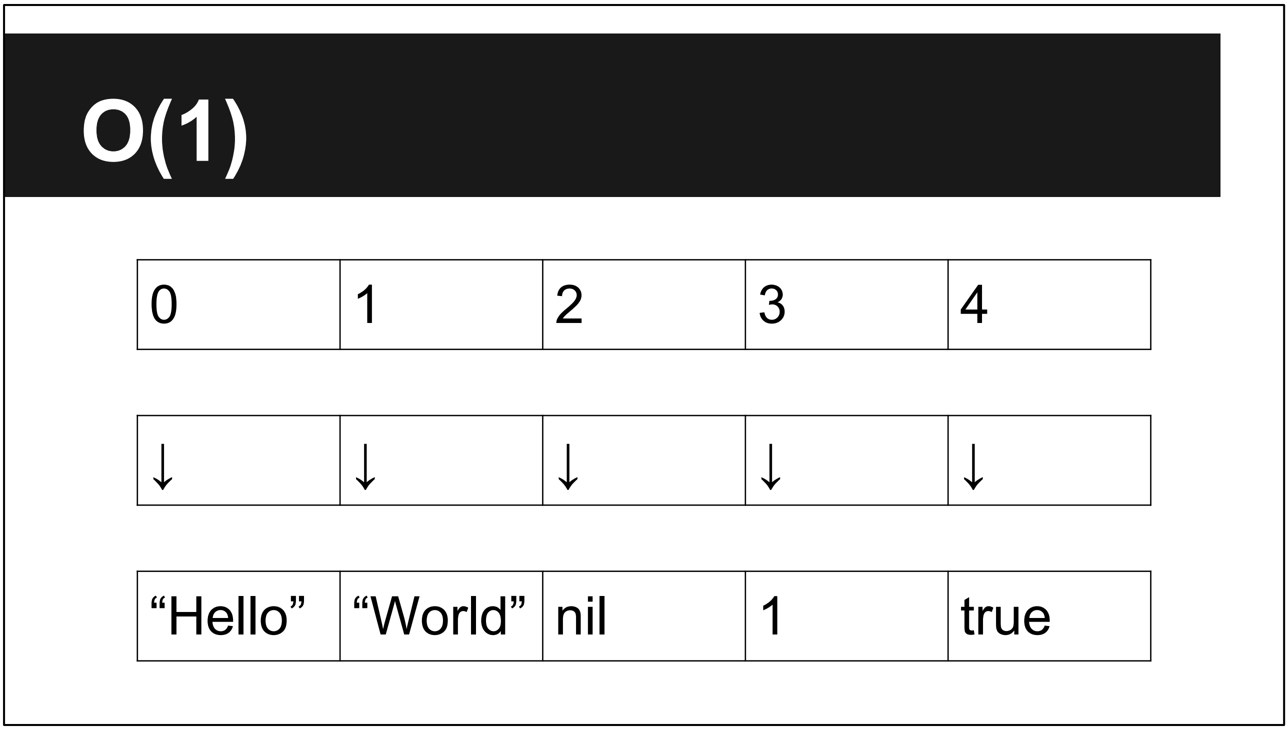
O(1)

```
[1,2,3][0]

{a: 1, b: 2}[:a]
```

These two Ruby expressions run in constant time:
- looking up an item in an Array by it’s index
- and looking up a pair in a Hash by it’s key

No matter how big the Array or Hash or, these methods will return instantly.

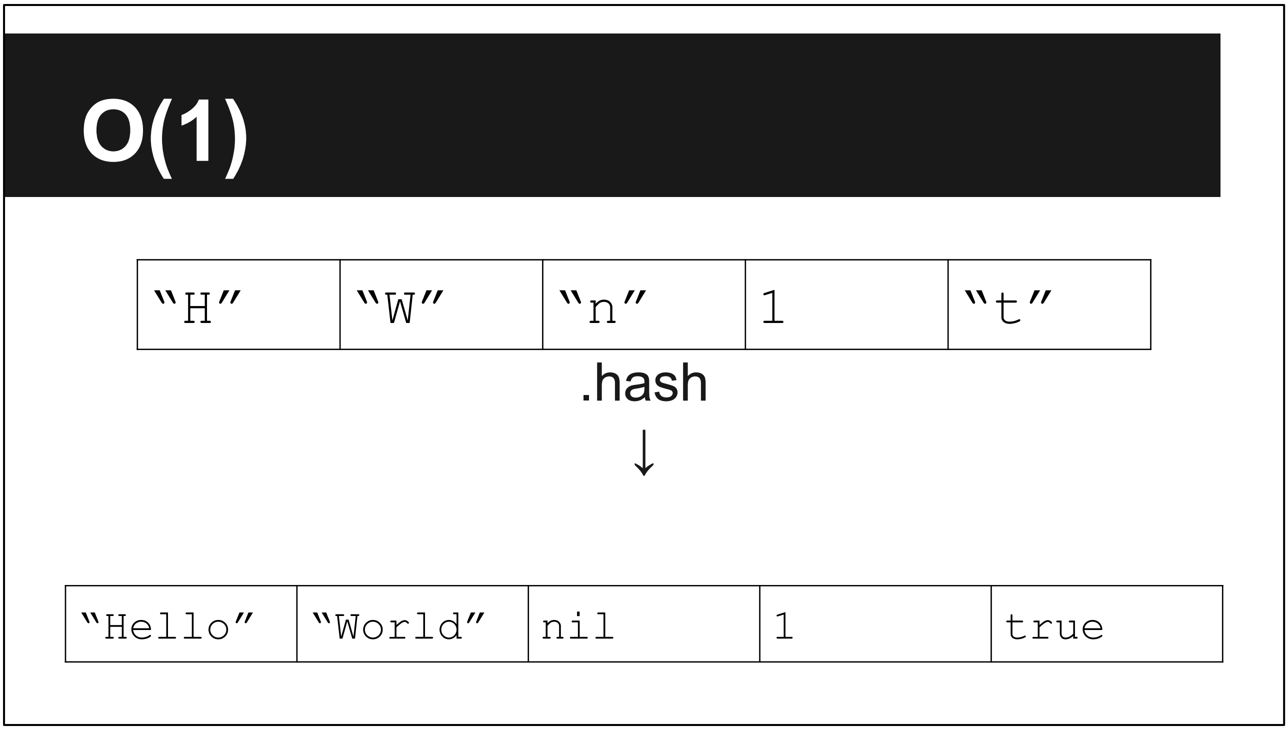


O(1)

0	1	2	3	4
↓	↓	↓	↓	↓
“Hello”	“World”	nil	1	true

I’ve actually already given you two hints that explain how this works. The first was these two slides from the Ruby Data Types lecture.

In the first slide, I showed how, in an Array, each index *points* to it’s value.



O(1)

“H”	“W”	“n”	1	“t”

.hash
↓

“Hello”	“World”	nil	1	true

In this slide, I showed you how, in Hashes, each key’s “hash” number *points* to it’s value.

O(1)

The last hint was when I said: “You just ask the RAM chip “is the 3rd light on” and it’ll tell you right away.”

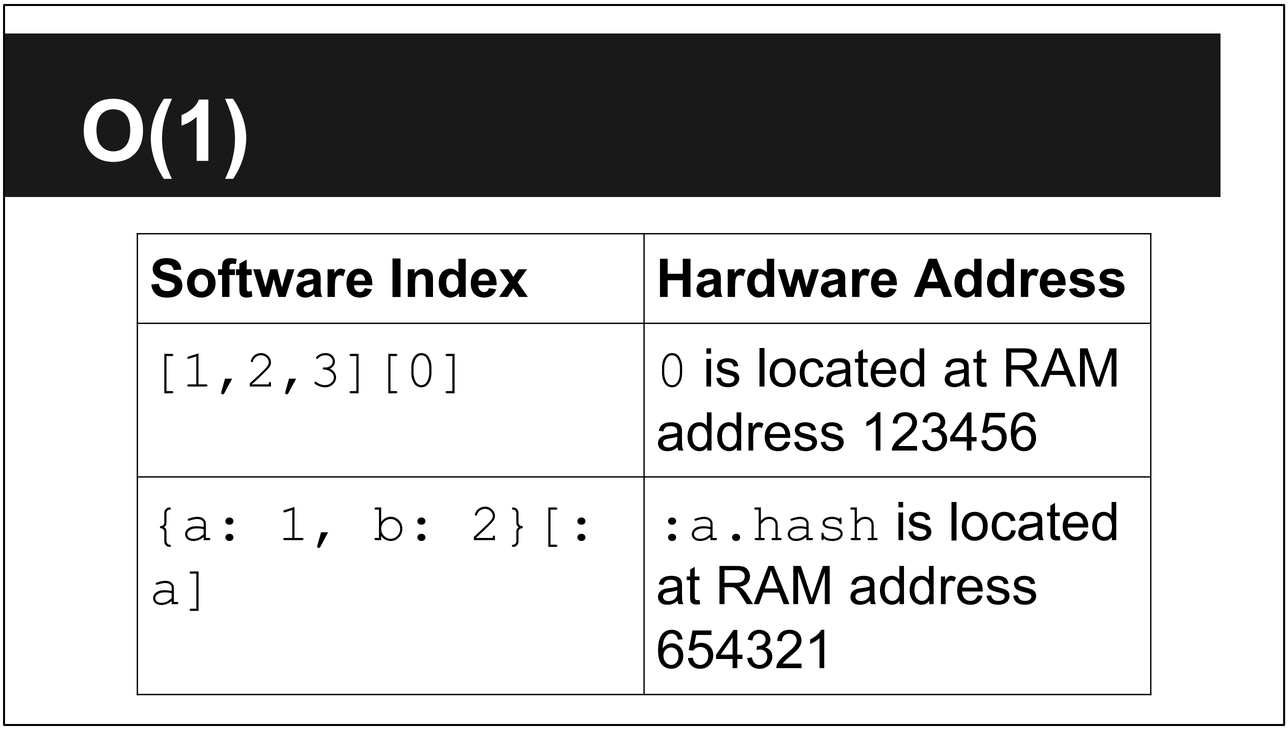
RAM, *random access* memory, can very quickly lookup any value you ask it for. In fact, it can look up values in O(1) time. RAM doesn’t get slower as it gets bigger. All other things being equal (materials, manufacturer, etc.) a 1GB RAM chip is just as fast as a 250GB RAM chip.

O(1 + seek time) == O(1)

Yes, a hard drive is slower. It has a “seek time”. But, even then, the lookup is still O (1). The seek time is constant. It’s the amount of time it takes to spin the disk into place + the amount of time to move the actuator arm into place. It doesn’t take any longer to read the first bit than it does the last bit.

[whiteboard demo]

So, in Big-O notation, the order-“1 + seek time” is the same at order-1.



O(1)

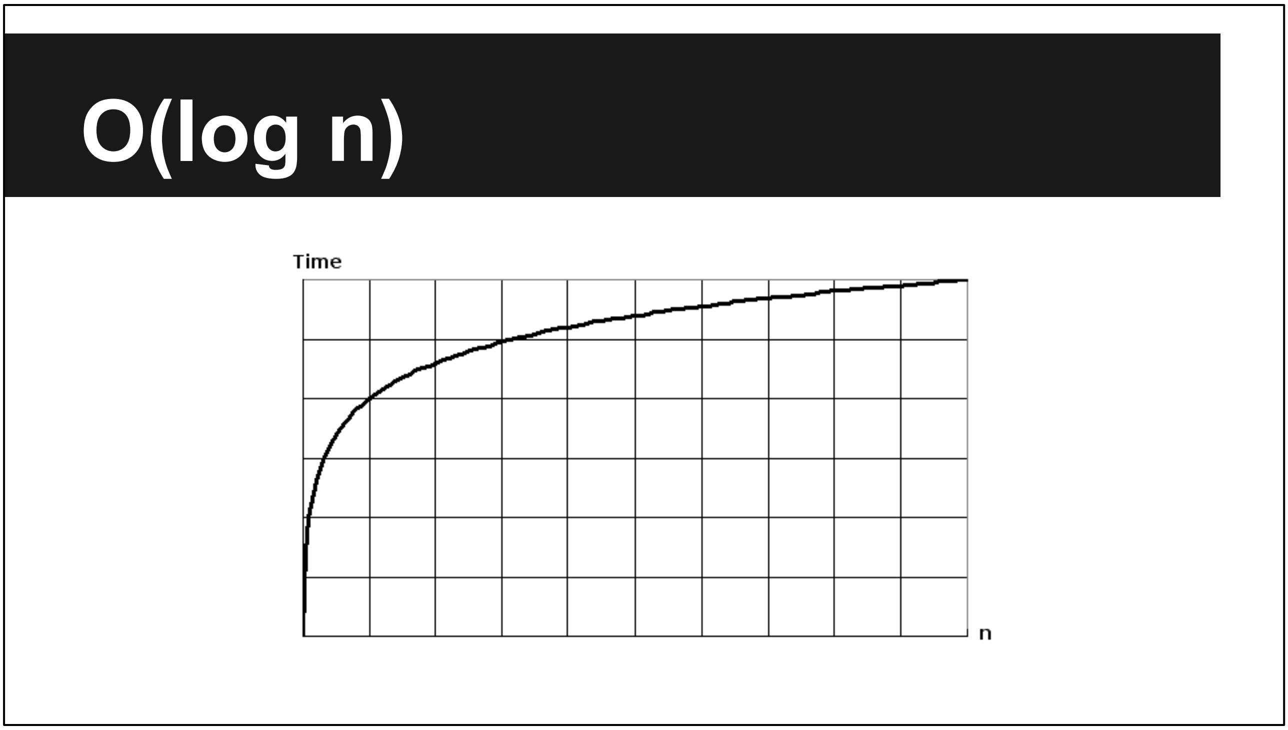
Software Index	**Hardware Address**
`[1,2,3][0]`	`0` is located at RAM address 123456
`{a: 1, b: 2}[:a]`	`:a.hash` is located at RAM address 654321

Behind the scenes, Ruby knows exactly where things are located in memory. Assume that each of the bits in RAM are numbered 0 to 1 billion (Why start at 0? Because Arrays always start at 0!). 1 billion bits is a little over 100 MB.

If you ask for the 0-index of an Array, Ruby knows that item is located at address 123456 in the RAM chip. And the RAM chip can return it immediately, without counting up from 0.

If you ask a Hash for the value associated with the key symbol-a, Ruby first gets the hash number of the key. Then it knows that hash number is located at address 654321 on the RAM chip, which it can retrieve right away.

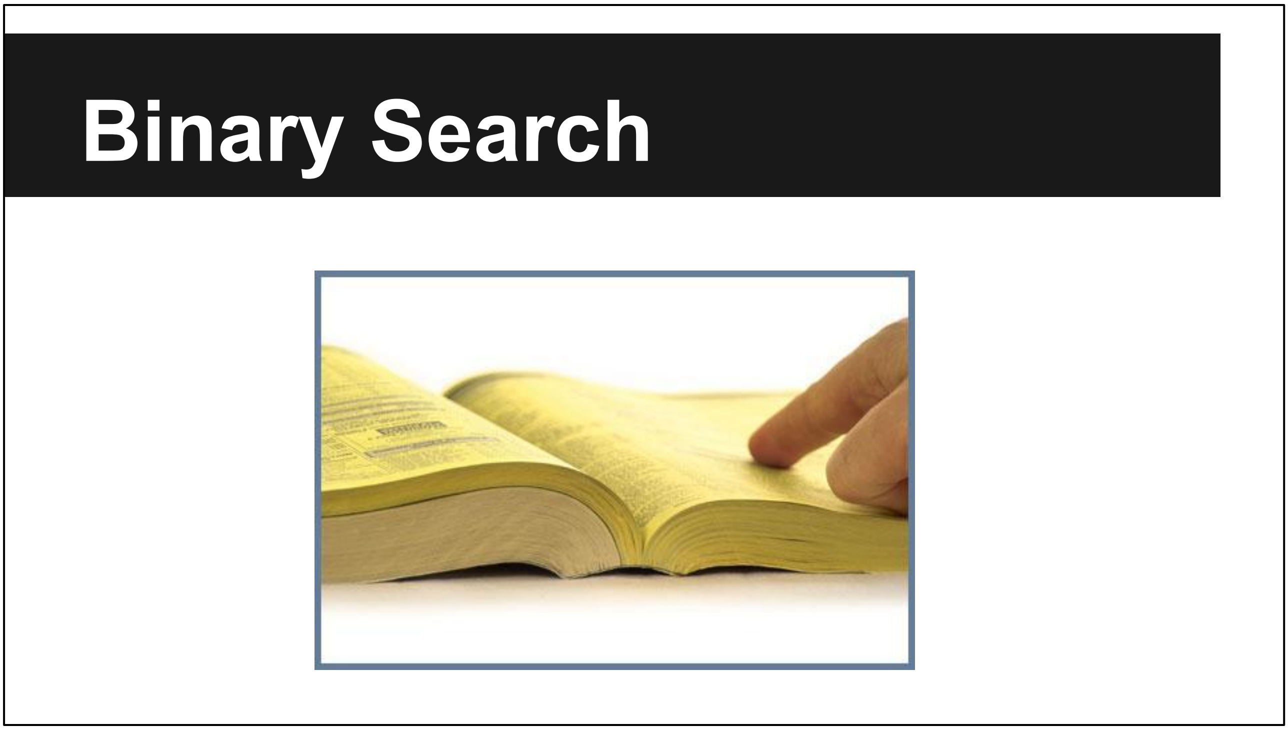
[board?]



O(log n)

There are two other common program performance times that I’m going to touch on briefly right now. The first is log-n.

As you can see, a program that runs in order-log-n gets slow very quickly in the beginning, but levels off as the size of your data gets larger.



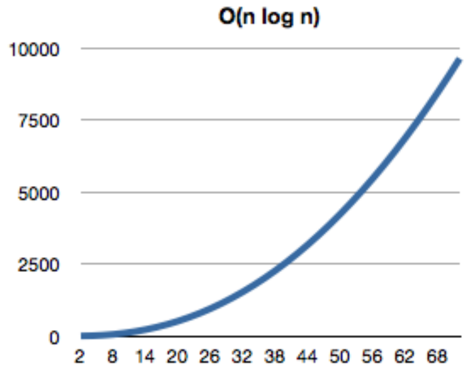
Binary Search

The most common algorithm that runs in log-n is binary-search, also known as the “phone book search” or “dictionary search” method. It starts by assuming that, like the entries in a phone book or dictionary, your array starts out already sorted.

[demo]

O(n log n)

```
[3,2,1].sort
```



Sorting an array, just by itself, takes n-log-n time. As you can see in the graph, that's actually much slower than log-n. It does not level off over time.

You can sort an array by using the "sort" method.

Sort + Binary Search

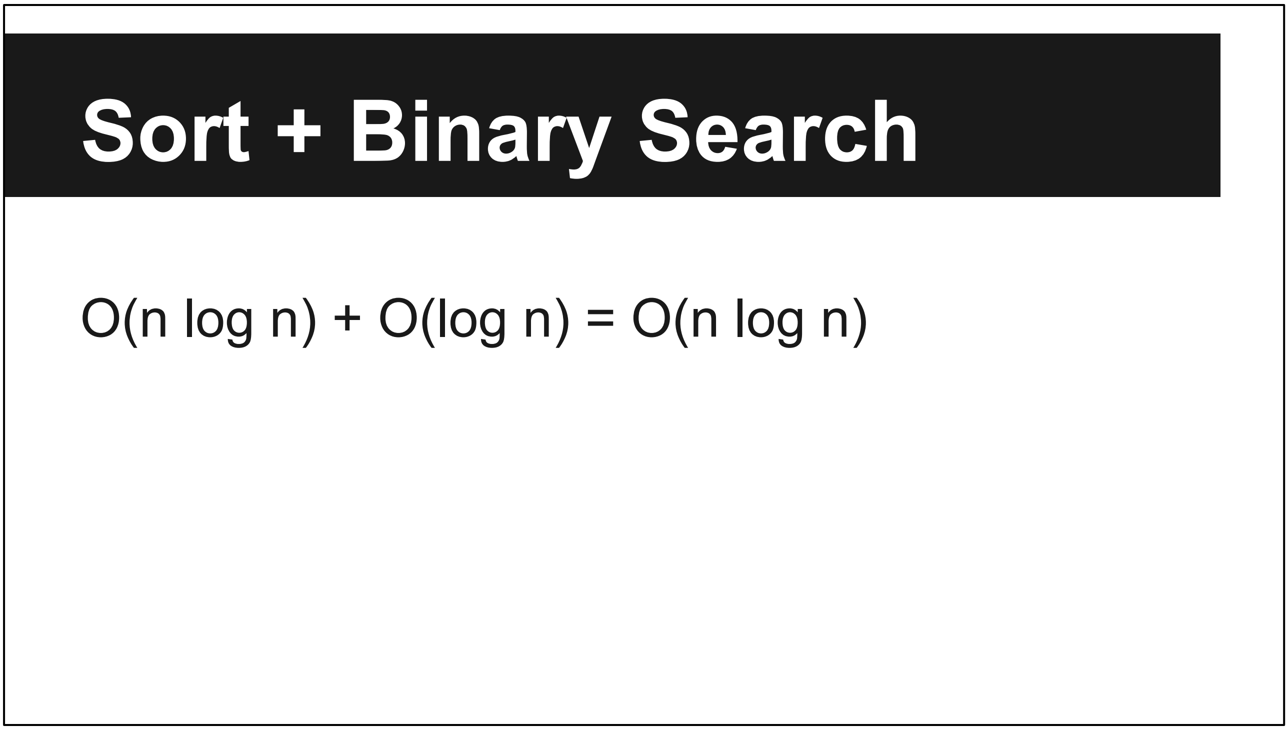
$O(n \log n) + O(\log n) < O(n)$?

Is $O(n \log n) + O(\log n)$ faster than $O(n)$?

Is Array.sort + binary search faster than Array.index?

You may be thinking to yourself: Binary search looks really fast. If I sort my Array first, then use a phone book search, wouldn’t that be faster than Array.index?

Is $O(n \log n) + O(\log n) < O(n)$

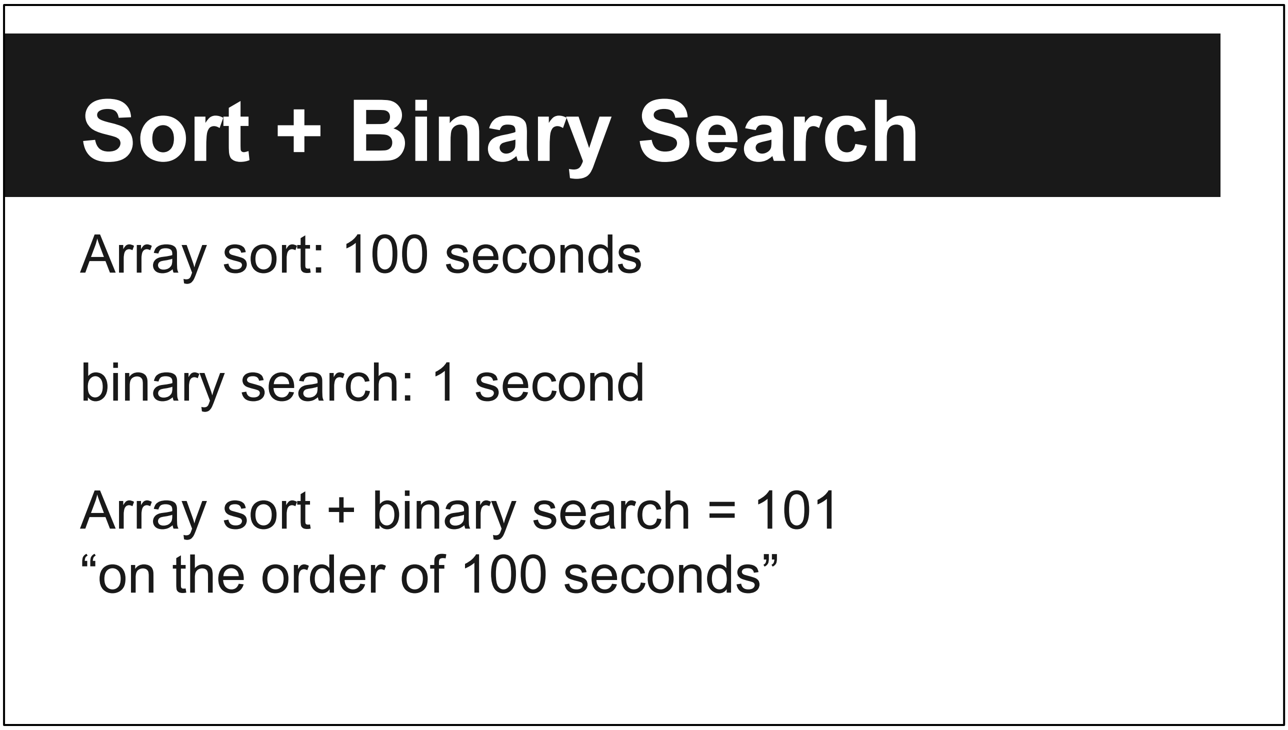


Sort + Binary Search

$O(n \log n) + O(\log n) = O(n \log n)$

No, actually, it’s not.

First, n log n is so much bigger than log n, that when you add them together you get an algorithm that still runs in O(n log n). That means that, over time, as n gets bigger, almost all of your time is spent sorting.



Sort + Binary Search

Array sort: 100 seconds

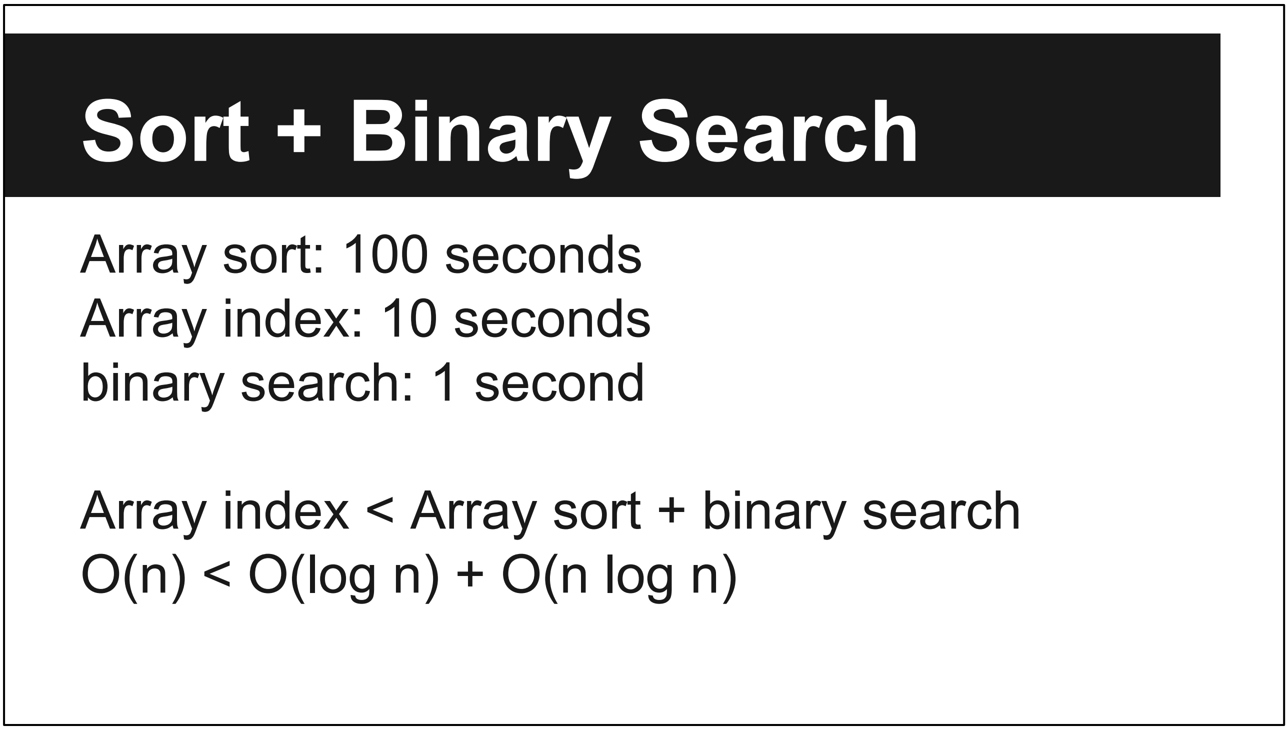
binary search: 1 second

Array sort + binary search = 101
“on the order of 100 seconds”

To put that in perspective, let’s say sorting some Array takes 100 seconds. Then doing a binary search for something in that Array takes 1 second.

The search is really fast. But if you do both, the result is still “on the order of 100 seconds”. Sorting takes so long, that the amount of time spent searching just doesn’t matter.

Big-O notation allows us to represent the concept of something being “on the order of” some number. Think of Big-O as standing for “on the order of”.



Sort + Binary Search

Array sort: 100 seconds
Array index: 10 seconds
binary search: 1 second

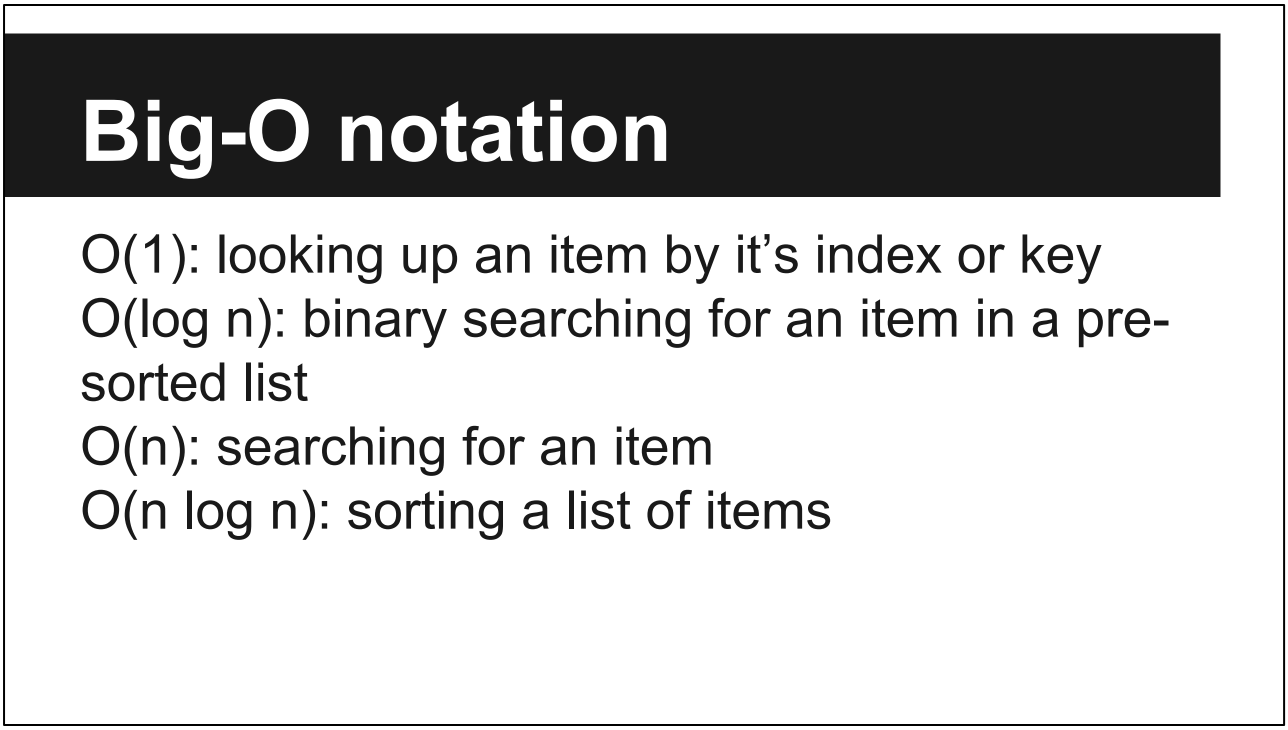
Array index < Array sort + binary search
O(n) < O(log n) + O(n log n)

So the second reason why sort + binary search is slower is:

Array.index lies one order of magnitude between sorting and binary search. So, let’s say it takes 10 seconds.

So sorting and binary searching takes “on the order of 100 seconds”. But just Array.index by itself takes “on the order of 10 seconds”.

So Array.index is actually faster than a combination of Array.sort + a binary search.



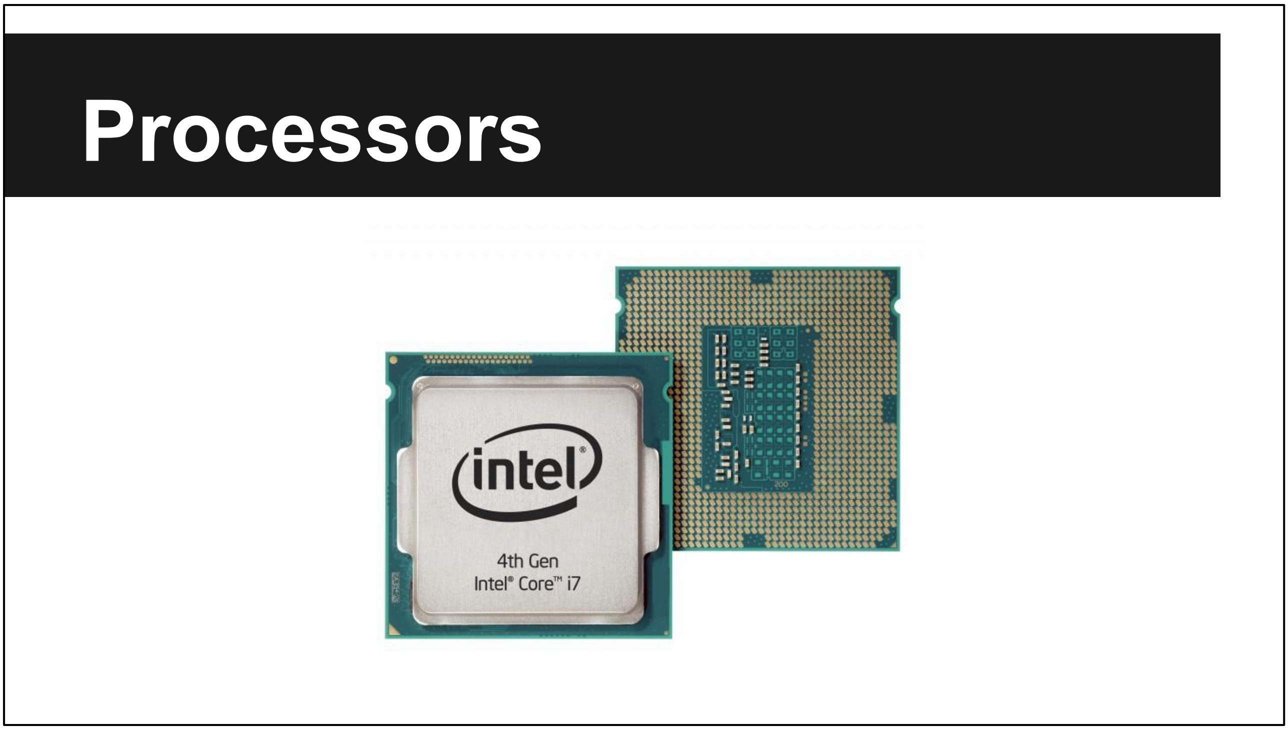
Big-O notation

O(1): looking up an item by it’s index or key
O(log n): binary searching for an item in a pre-sorted list
O(n): searching for an item
O(n log n): sorting a list of items

The orders of magnitude I’ve shown you so far, in order, are…

[slide]

There is one more common ones, but we’ll get to that later.



Processors

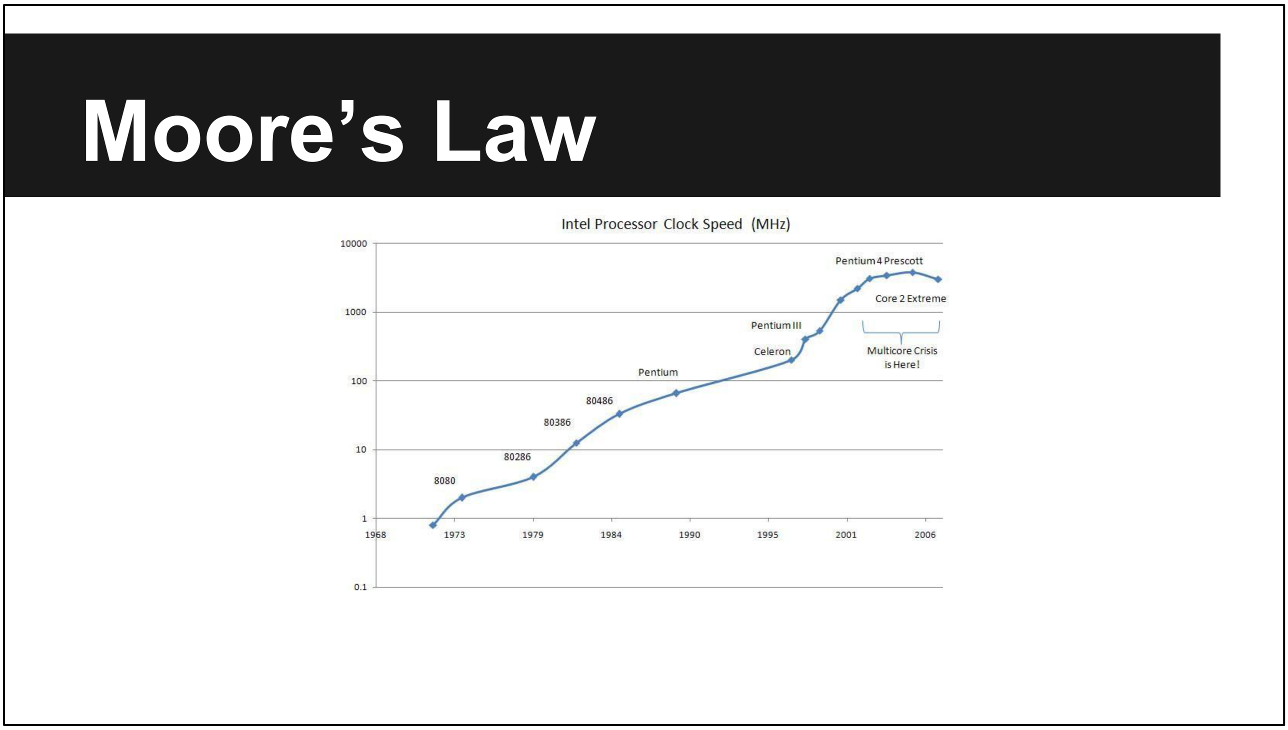
What about processors?

I’ve spent all this time talking about the performance of computer programs, but haven’t once mentioned the one thing all the commercials tell you makes all the difference: the CPU, or central processing unit.

A CPU has two parts. The first, the control unit, is where you send commands to do things. So, for example, if you want to read a bit of RAM or the disk, you’d send the command to the control unit of the CPU.

The second part, the Arithmetic Logic Unit (ALU) is, itself, a very small computer with a very small piece of Turing tape (yes, that’s Turing tape #3). When you hear about a 32-bit or a 64-bit processor, that’s the length of the Turing tape. Just 32 or 64 measly bits.

But it turns out that most of the mathematical operations you do in a computer happen to fit into 32 or 64 bits. So, for those operations, instead of operating at RAM’s millions of operations per second, you can operate at the CPU’s billions of operations per second. That’s the fastest your computer can go. As the maximum speed, it’s an attractive number for people to print in ads and on the sides of computer boxes.

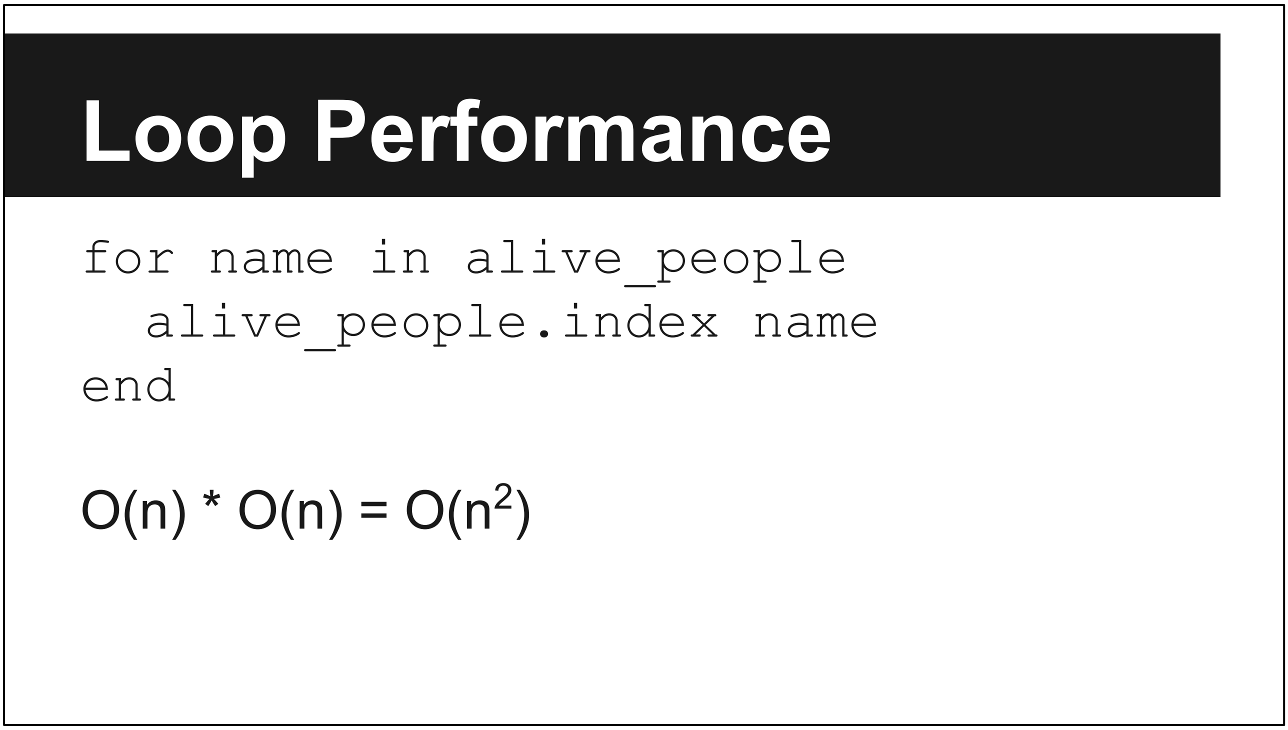


Moore’s Law

We’re approaching the physical limits of processor architecture. Intel co-founder Gordon Moore predicted in 1965 that the number of transistors on integrated circuits would double every two years. But to get that last oomph over the last few years has required not so much faster CPUs, but more of them. Multi-core processors allow you to execute multiple small problems at high speeds simultaneously (aka. “in parallel). So Moore’s law has held on, but no one is sure for how long. It can’t go on forever.

Despite the increasing speed of processors, sometimes you still have to work with numbers or data that are bigger than 64-bits. In that case, your computer starts working in RAM. And, when that’s not enough, it starts working with the disk drive. And when that’s not enough, it’s time to get a new computer.

Source: http://www.enterpriseirregulars.com/71244/moores-law-put-apple-drivers-seat-cost-steve-ballmer-job/



Loop Performance

```
for name in alive_people
  alive_people.index name
end
```

$O(n) * O(n) = O(n^2)$

Recall this slide from the Flow Control lecture:

Once you start writing loops, you enter a world where bad things can happen to your performance. An infinite loop, for example, runs in O-infinity. It never stops.

A loop like this runs in $O(n^2)$. If alive_people has n items, and for each item you run an O(n) method, the end result is O(n-squared). [board?]

Loops are where program performance gets hit the hardest. If you’re looking for a place to make your program faster, focus on making the blocks within loops as small and fast as possible.