Hello world, I'm Carrie Anne, and welcome to Crash Course Computer Science!

Hello world！我是 Carrie Anne，欢迎收看计算机科学速成课！

Over the course of this series, we're going to go from bits, bytes, transistors and logic gates,

在这个系列中，我们会学习 Bits(位)，Bytes(字节)，晶体管, 逻辑门，

all the way to Operating Systems, Virtual Reality and Robots!

一直到操作系统，虚拟现实和机器人!

We're going to cover a lot, but just to clear things up

我们要学很多东西，但预先说明

we ARE NOT going to teach you how to program.

我们 \*不会\* 教你怎么编程

Instead, we're going to explore a range of computing topics as a discipline and a technology.

我们会从高层次上纵览一系列计算机话题

Computers are the lifeblood of today's world.

计算机是当今世界的命脉

If they were to suddenly turn off, all at once,

如果突然关掉所有的计算机

the power grid would shut down, cars would crash, planes would fall,

电网会关闭，车辆会相撞，飞机会坠毁

water treatment plants would stop, stock markets would freeze,

净水厂会关闭，证券市场会停止运作

trucks with food wouldn't know where to deliver, and employees wouldn't get paid.

装满食物的卡车不知运往何方，员工得不到薪水

Even many non-computer objects -like DFTBA shirts and the chair I'm sitting on-

甚至很多和计算机无关的东西，例如 DFTBA 的 T 恤和我现在坐的椅子

are made in factories run by computers.

也都是在计算机管理的工厂中制造的

Computing really has transformed nearly every aspect of our lives.

计算机改变了我们生活中几乎所有方面

And this isn't the first time we've seen this sort of technology-driven global change.

我们也不是第一次遇到推动全球发展的科技了

Advances in manufacturing during the Industrial Revolution

工业革命中生产能力的提高

brought a new scale to human civilization - in agriculture, industry and domestic life.

大幅提升了农业，工业，畜牧业的规模

Mechanization meant superior harvests and more food, mass produced goods,

机械化导致更好的收成，更多的食物，商品可以大批量生产

cheaper and faster travel and communication, and usually a better quality of life.

旅行和通讯变得更便宜更快，生活质量变得更好.

And computing technology is doing the same right now

计算机和工业革命有一样的影响

- from automated farming and medical equipment,

从自动化农业和医疗设备

to global telecommunications and educational opportunities,

到全球通信和教育机会

and new frontiers like Virtual Reality and Self Driving Cars.

还有 虚拟现实 和 无人驾驶汽车 等新领域

We are living in a time likely to be remembered as the Electronic Age.

现在这个时代很可能会被后人总结成 "信息时代"

And with billions of transistors in just your smartphones, computers can seem pretty complicated,

你的智能手机中有数十亿个晶体管，看起来好像很复杂

but really, they're just simple machines

但实际上它是很简单的机器

that perform complex actions through many layers of abstraction.

通过一层层的抽象 来做出复杂操作

So in this series, we're going break down those layers,

在这个系列中，我们会一层层讲解，

and build up from simple 1's and 0's, to logic units, CPUs,

从最底层的1和0，到逻辑门，CPU

operating systems, the entire internet and beyond.

操作系统，整个互联网，以及更多~~

And don't worry, in the same way someone buying t-shirts on a webpage

不用担心，正如在网上买T恤的人 不用知道网站代码是怎么写的

doesn't need to know how that webpage was programmed,

不用担心，正如在网上买T恤的人 不用知道网站代码是怎么写的

or the web designer doesn't need to know how all the packets are routed,

设计师不用知道数据包是怎么传输的

or router engineers don't need to know about transistor logic,

设计路由器的工程师不用理解晶体管的逻辑

this series will build on previous episodes but not be dependent on them.

本系列中每个视频会接着上集继续讲，但并不依赖前面的视频

By the end of this series,

等这个系列结束后

I hope that you can better contextualize computing's role both in your own life and society,

希望你能了解计算机在你的人生 以及社会中扮演什么角色

and how humanity's (arguably) greatest invention is just in its infancy,

以及这个人类史上最伟大的发明（可以这样说啦）是怎么开始的，

with its biggest impacts yet to come.

它对未来还会有更大的影响

But before we get into all that, we should start at computing's origins,

但深入之前，我们应该从计算的起源讲起,

because although electronic computers are relatively new, the need for computation is not.

虽然电子计算机才出现不久，但人类对计算的需求早就有了

The earliest recognized device for computing was the abacus,

公认最早的计算设备是 算盘

invented in Mesopotamia around 2500 BCE.

发明于"美索不达米亚"，大约公元前 2500 年

It's essentially a hand operated calculator,

它是手动计算器，用来帮助加减数字

that helps add and subtract many numbers.

它是手动计算器，用来帮助加减数字

It also stores the current state of the computation, much like your hard drive does today.

它存储着当前的计算状态，类似于如今的硬盘

The abacus was created because,

人们制造算盘是因为

the scale of society had become greater than

社会的规模已经超出个人心算的能力

what a single person could keep and manipulate in their mind.

社会的规模已经超出个人心算的能力

There might be thousands of people in a village or tens of thousands of cattle.

一个村庄可能有上千个人和上万头牛

There are many variants of the abacus,

算盘有很多变种

but let's look at a really basic version with each row representing a different power of ten.

但我们来看一个基础版，每行代表 10 的不同次方

So each bead on the bottom row represents a single unit,

最底下那行，一个珠子代表 10 的 0 次方，也就是 1，

in the next row they represent 10, the row above 100, and so on.

再上面一行是 10 的 1 次方（也就是 10） \N 再上面一行是 10 的 2 次方 （以此类推）

Let's say we have 3 heads of cattle represented by 3 beads on the bottom row on the right side.

假设最底部的 3 颗珠子，代表 3 头牛

If we were to buy 4 more cattle we would just slide 4 more beads to the right for a total of 7.

假设再买 4 头牛，只需要向右移动 4 颗珠子，共 7 个珠子

But if we were to add 5 more after the first 3 we would run out of beads,

但如果再买 5 头，珠子就不够用了

so we would slide everything back to the left,

所以把所有珠子移回左边

slide one bead on the second row to the right, representing ten,

在第二排把 1 颗珠子向右移动，代表 10

and then add the final 2 beads on the bottom row for a total of 12.

然后最底下那行，向右移动 2 颗珠子，代表 12

This is particularly useful with large numbers.

这种方法处理大数字很有效

So if we were to add 1,251

假设要表示 1251

we would just add 1 to the bottom row, 5 to the second row, 2 to the third row, and 1 to the fourth row

从下往上：\N第一行移 1 个，第二行移 5 个\N第三行移 2 个，第四行移 1 个

- we don't have to add in our head and the abacus stores the total for us.

我们不用记在脑子里，算盘会记住.

Over the next 4000 years, humans developed all sorts of clever computing devices,

在接下来 4000 年，人类发明了各种巧妙的计算设备

like the astrolabe, which enabled ships to calculate their latitude at sea.

比如星盘，让船只可以在海上计算纬度

Or the slide rule, for assisting with multiplication and division.

或计算尺，帮助计算乘法和除法

And there are literally hundred of types of clocks created

人们还创造了上百种时钟

that could be used to calculate sunrise, tides, positions of celestial bodies, and even just the time.

算日出，潮汐，天体的位置，或纯粹拿来计时

Each one of these devices made something that was previously laborious to calculate much faster,

这些设备让原先很费力的事变得更快，更简单，更精确

easier, and often more accurate

这些设备让原先很费力的事变得更快，更简单，更精确

- it lowered the barrier to entry,

降低了门槛

and at the same time, amplified our mental abilities -

加强了我们的能力

take note, this is a theme we're going to touch on a lot in this series.

记笔记!（敲黑板）这个系列会多次提到这一点

As early computer pioneer Charles Babbage said:

计算机先驱 Charles Babbage 说过：

"At each increase of knowledge, as well as on the contrivance of every new tool,

"随着知识的增长和新工具的诞生，人工劳力会越来越少"

human labour becomes abridged."

"随着知识的增长和新工具的诞生，人工劳力会越来越少"

However, none of these devices were called "computers".

然而，这些设备那时都不叫 "计算机"

The earliest documented use of the word "computer" is from 1613, in a book by Richard Braithwait.

最早使用 "计算机" 一词的文献 \N 来自 1613 年的一本书，作者 Richard Braithwait

And it wasn't a machine at all - it was a job title.

然而指的不是机器，而是一种职业

Braithwait said,

Braithwait 说：

"I have read the truest computer of times,

"我听说过的计算者里最厉害的，能把好几天的工作量大大缩减"

and the best arithmetician that ever breathed, and he reduceth thy dayes into a short number".

"我听说过的计算者里最厉害的，能把好几天的工作量大大缩减"

In those days, computer was a person who did calculations,

那时, "Computer" 指负责计算的人

sometimes with the help of machines, but often not.

"Computer" 偶尔会用机器帮忙，但大部分时候靠自己

This job title persisted until the late 1800s,

这个职位一直到 1800 年代还存在

when the meaning of computer started shifting to refer to devices.

之后 "Computer" 逐渐开始代表机器

Notable among these devices was the Step Reckoner,

其中"步进计算器"最有名

built by German polymath Gottfried Leibniz in 1694.

由德国博学家 戈特弗里德·莱布尼茨 建造于 1694 年

Leibniz said "... it is beneath the dignity of excellent men to waste their time in calculation

莱布尼茨说过 "... 让优秀的人浪费时间算数简直侮辱尊严

when any peasant could do the work just as accurately with the aid of a machine."

农民用机器能算得一样准"

It worked kind of like the odometer in your car,

"步进计算器"有点像汽车里的里程表，不断累加里程数

which is really just a machine for adding up the number of miles your car has driven.

"步进计算器"有点像汽车里的里程表，不断累加里程数

The device had a series of gears that turned;

它有一连串可以转动的齿轮

each gear had ten teeth, to represent the digits from 0 to 9.

每个齿轮有十个齿，代表数字0到9

Whenever a gear bypassed nine, it rotated back to 0 and advanced the adjacent gear by one tooth.

每当一个齿轮转过 9，它会转回 0，同时让旁边的齿轮前进 1 个齿

Kind of like when hitting 10 on that basic abacus.

就像算盘超过 10 一样.

This worked in reverse when doing subtraction, too.

做减法时，机器会反向运作.

With some clever mechanical tricks,

利用一些巧妙的机械结构

the Step Reckoner was also able to multiply and divide numbers.

步进计算器也能做乘法和除法

Multiplications and divisions are really just many additions and subtractions.

乘法和除法 实际上只是多个加法和减法

For example, if we want to divide 17 by 5, we just subtract 5, then 5, then 5 again,

举例，17除以5，我们只要减5，减5，再减5

and then we can't subtract any more 5's… so we know 5 goes into 17 three times, with 2 left over.

直到不能再减 5，就知道了 17=5x3+2

The Step Reckoner was able to do this in an automated way,

步进计算器 可以自动完成这种操作

and was the first machine that could do all four of these operations.

它是第一台能做"加减乘除"全部四种运算的机器

And this design was so successful it was used for the next three centuries of calculator design.

它的设计非常成功，以至于沿用了 3 个世纪.

Unfortunately, even with mechanical calculators,

不幸的是，即使有机械计算器

most real world problems required many steps of computation before an answer was determined.

许多现实问题 依然需要很多步

It could take hours or days to generate a single result.

算一个结果可能要几小时甚至几天

Also, these hand-crafted machines were expensive, and not accessible to most of the population.

而且这些手工制作的机器非常昂贵，大部分人买不起

So, before 20th century,

所以在 20 世纪以前

most people experienced computing through pre-computed tables

大部分人会用预先算好的计算表

assembled by those amazing "human computers" we talked about.

这些计算表由之前说的 "人力计算器" 编撰

So if you needed to know the square root of 8 million 6 hundred and 75 thousand 3 hundred and 9,

如果你想知道 867,5309 的平方根

instead of spending all day hand-cranking your step reckoner,

与其花一整天来手摇 "步进计算器"

you could look it up in a huge book full of square root tables in a minute or so.

你可以花一分钟在表里找答案

Speed and accuracy is particularly important on the battlefield,

速度和准确性在战场上尤为重要

and so militaries were among the first to apply computing to complex problems.

因此军队很早就开始用计算解决复杂问题

A particularly difficult problem is accurately firing artillery shells,

如何精确瞄准炮弹是一个很难的问题

which by the 1800s could travel well over a kilometer (or a bit more than half a mile).

19世纪，这些炮弹的射程可以达到 1 公里以上（比半英里多一点）

Add to this varying wind conditions, temperature, and atmospheric pressure,

因为风力，温度，大气压力会不断变化

and even hitting something as large as a ship was difficult.

想打中船一样大的物体也非常困难

Range Tables were created that allowed gunners to look up environmental conditions

于是出现了射程表，炮手可以查环境条件和射击距离

and the distance they wanted to fire,

于是出现了射程表，炮手可以查环境条件和射击距离

and the table would tell them the angle to set the canon.

然后这张表会告诉他们，角度要设成多少

These Range Tables worked so well, they were used well into World War Two.

这些射程表很管用，二战中被广泛应用

The problem was, if you changed the design of the cannon or of the shell,

问题是如果改了大炮或炮弹的设计，就要算一张新表

a whole new table had to be computed,

问题是如果改了大炮或炮弹的设计，就要算一张新表

which was massively time consuming and inevitably led to errors.

这样很耗时而且会出错

Charles Babbage acknowledged this problem in 1822

Charles Babbage 在 1822 年写了一篇论文

in a paper to the Royal Astronomical Society entitled:

向皇家天文学会指出了这个问题

"Note on the application of machinery to the computation of astronomical and mathematical tables".

标题叫： "机械在天文与计算表中的应用"

Let's go to the thought bubble.

让我们进入思想泡泡

Charles Babbage proposed a new mechanical device called the Difference Engine,

Charles Babbage 提出了一种新型机械装置叫 "差分机"

a much more complex machine that could approximate polynomials.

一个更复杂的机器，能近似多项式.

Polynomials describe the relationship between several variables

多项式描述了几个变量之间的关系

- like range and air pressure, or amount of pizza Carrie Anne eats and happiness.

比如射程和大气压力，或者 Carrie Anne 要吃多少披萨才开心

Polynomials could also be used to approximate logarithmic and trigonometric functions,

多项式也可以用于近似对数和三角函数

which are a real hassle to calculate by hand.

这些函数手算相当麻烦

Babbage started construction in 1823,

Charles Babbage 在 1823 年开始建造差分机

and over the next two decades, tried to fabricate and assemble the 25,000 components,

并在接下来二十年，试图制造和组装 25,000 个零件

collectively weighing around 15 tons.

总重接近 15 吨

Unfortunately, the project was ultimately abandoned.

不幸的是，该项目最终放弃了

But, in 1991,

但在 1991 年

historians finished constructing a Difference Engine based on Babbage's drawings and writings

历史学家根据 Charles Babbage 的草稿做了一个差分机

- and it worked!

而且它还管用！

But more importantly, during construction of the Difference Engine,

但更重要的是，在差分机的建造期间

Babbage imagined an even more complex machine - the Analytical Engine.

Charles Babbage 构想了一个更复杂的机器 - 分析机

Unlike the Difference Engine,

不像差分机，步进计算器 和以前的其他计算设备

Step Reckoner and all other computational devices before it

不像差分机，步进计算器 和以前的其他计算设备

- the Analytical Engine was a "general purpose computer".

分析机是 "通用计算机"

It could be used for many things, not just one particular computation;

它可以做很多事情，不只是一种特定运算

it could be given data and run operations in sequence;

甚至可以给它数据，然后按顺序执行一系列操作

it had memory and even a primitive printer.

它有内存 甚至一个很原始的打印机

Like the Difference Engine, it was ahead of its time, and was never fully constructed.

就像差分机，这台机器太超前了，所以没有建成

However, the idea of an "automatic computer"

然而，这种 "自动计算机" 的概念

- one that could guide itself through a series of operations automatically,

-计算机可以自动完成一系列操作

was a huge deal, and would foreshadow computer programs.

是个跨时代的概念，预示着计算机程序的诞生

English mathematician Ada Lovelace wrote hypothetical programs for the Analytical Engine, saying,

英国数学家 Ada Lovelace 给分析机写了假想的程序，她说：

"A new, a vast, and a powerful language is developed for the future use of analysis."

"未来会诞生一门全新的，强大的，专为分析所用的语言"

For her work, Ada is often considered the world's first programmer.

因此 Ada 被认为是世上第一位程序员.

The Analytical Engine would inspire, arguably, the first generation of computer scientists,

分析机激励了（可以这么讲）第一代计算机科学家

who incorporated many of Babbage's ideas in their machines.

这些计算机科学家 \N 把很多 Charles Babbage 的点子融入到他们的机器

This is why Babbage is often considered the "father of computing".

所以 Charles Babbage 经常被认为是 "计算之父"

Thanks! Thought Bubble

谢啦！思想泡泡

So by the end of the 19th century,

到了 19 世纪末

computing devices were used for special purpose tasks in the sciences and engineering,

科学和工程领域中的特定任务 会用上计算设备

but rarely seen in business, government or domestic life.

但公司，政府，家庭中很少见到计算设备

However, the US government faced a serious problem for its 1890 census

然而，美国政府在 1890 年的人口普查中面临着严重的问题

that demanded the kind of efficiency that only computers could provide.

只有计算机能提供所需的效率

The US Constitution requires that a census be conducted every ten years,

美国宪法要求 10 年进行一次人口普查

for the purposes of distributing federal funds, representation in congress, and good stuff like that.

目的是分配联邦资金，国会代表，等等

And by 1880s, the US population was booming, mostly due to immigration.

到 1880 年代，美国人口迅速增长，大部分因为移民

That census took seven years to manually compile

人口普查要七年时间来手工编制，等做完都过时了

and by the time it was completed, it was already out of date

人口普查要七年时间来手工编制，等做完都过时了

- and it was predicted that the 1890 census would take 13 years to compute.

而且 1890 年的人口普查，预计要 13 年完成

That's a little problematic when it's required every decade!

但人口普查可是 10 年一次啊！

The Census bureau turned to Herman Hollerith, who had built a tabulating machine.

人口普查局找了 Herman Hollerith，他发明了打孔卡片制表机

His machine was "electro-mechanical"

他的机器是 "电动机械的"

- it used traditional mechanical systems for keeping count,

- 用传统机械来计数

like Leibniz's Step Reckoner - but coupled them with electrically-powered components.

结构类似莱布尼茨的乘法器，但用电动结构连接其他组件

Hollerith's machine used punch cards

Hollerith 的机器用打孔卡

which were paper cards with a grid of locations that can be punched out to represent data.

一种纸卡，上面有网格，用打孔来表示数据.

For example, there was a series of holes for marital status.

举个例子，有一连串孔代表婚姻状况

If you were married, you would punch out the married spot,

如果你结婚了，就在 "结婚" 的位置打孔

then when the card was inserted into Hollerith's machine, little metal pins would come down over the card

当卡插入 Hollerith 的机器时，小金属针会到卡片上

- if a spot was punched out, the pin would pass through the hole in the paper

-如果有个地方打孔了，针会穿过孔

and into a little vial of mercury, which completed the circuit.

泡入一小瓶汞，联通电路

This now completed circuit powered an electric motor,

电路会驱动电机

which turned a gear to add one, in this case, to the "married" total.

然后给 "已婚" 的齿轮 + 1

Hollerith's machine was roughly 10x faster than manual tabulations,

Hollerith 的机器速度是手动的 10 倍左右

and the Census was completed in just two and a half years

使人口普查在短短两年半内完成

- saving the census office millions of dollars.

给人口普查办公室省了上百万美元

Businesses began recognizing the value of computing,

企业开始意识到计算机的价值

and saw its potential to boost profits by improving labor- and data-intensive tasks,

可以提升劳动力以及数据密集型任务 来提升利润

like accounting, insurance appraisals, and inventory management.

比如会计，保险评估和库存管理等行业

To meet this demand, Hollerith founded The Tabulating Machine Company,

为了满足这一需求，Hollerith 成立了制表机器公司

which later merged with other machine makers in 1924

这家公司后来在 1924 年与其它机械制造商合并

to become The International Business Machines Corporation or IBM

成为了 "国际商业机器公司"，简称 IBM

- which you've probably heard of.

-你可能听过 IBM

These electro-mechanical "business machines" were a huge success, transforming commerce and government,

这些电子机械的 "商业机器" 取得了巨大成功，改变了商业和政府.

and by the mid-1900s, the explosion in world population and the rise of globalized trade

到了 1900 年代中叶，世界人口的爆炸和全球贸易的兴起

demanded even faster and more flexible tools for processing data,

要求更快，更灵活的工具来处理数据

setting the stage for digital computers,

为电子计算机的发展奠定了基础

which we'll talk about next week.

我们下周讨论

Our last episode brought us to the start of the 20th century,

上集讲到 20 世纪初

where early, special purpose computing devices, like tabulating machines,

当时的早期计算设备都针对特定用途 比如 制表机

were a huge boon to governments and business

大大推进了政府和企业

- aiding, and sometimes replacing, rote manual tasks.

它们帮助, 甚至代替了人工

But the scale of human systems continued to increase at an unprecedented rate.

然而人类社会的规模 在以前所未有的速度增长

The first half of the 20th century saw the world's population almost double.

20世纪上半叶，世界人口几乎翻倍

World War 1 mobilized 70 million people, and World War 2 involved more than 100 million.

一战动员7千万人，二战1亿多人

Global trade and transit networks became interconnected like never before,

全球贸易和运输更加紧密

and the sophistication of our engineering and scientific endeavors reached new heights

工程和科学的复杂度也达到新高

- we even started to seriously consider visiting other planets.

- 我们甚至开始考虑登陆其他行星

And it was this explosion of complexity, bureaucracy, and ultimately data,

复杂度的增高导致数据量暴增

that drove an increasing need for automation and computation.

人们需要更多自动化 更强的计算能力

Soon those cabinet-sized electro-mechanical computers grew into room-sized behemoths

很快，柜子大小的计算机变成房间大小

that were expensive to maintain and prone to errors.

维护费用高 而且容易出错

And it was these machines that would set the stage for future innovation.

而正是这些机器 为未来的创新打下基础

One of the largest electro-mechanical computers built was the Harvard Mark I,

最大的机电计算机之一是 哈佛马克一号

completed in 1944 by IBM for the Allies during World War 2.

IBM 在 1944 完成建造，给二战同盟国建造的.

It contained 765,000 components, three million connections, and five hundred miles of wire.

它有76万5千个组件，300万个连接点和500英里长的导线

To keep its internal mechanics synchronized,

为了保持内部机械装置同步

it used a 50-foot shaft running right through the machine driven by a five horsepower motor.

它有一个50英尺的传动轴，由一个 5 马力的电机驱动

One of the earliest uses for this technology was running simulations for the Manhattan Project.

这台机器最早的用途之一 是给"曼哈顿计划"跑模拟

The brains of these huge electro-mechanical beasts were relays:

这台机器的大脑是"继电器"

electrically-controlled mechanical switches.

继电器是：用电控制的机械开关

In a relay, there is a control wire that determines whether a circuit is opened or closed.

继电器里，有根"控制线路"，控制电路是开还是关

The control wire connects to a coil of wire inside the relay.

"控制线路" 连着一个线圈

When current flows through the coil, an electromagnetic field is created,

当电流流过线圈，线圈产生电磁场

which in turn, attracts a metal arm inside the relay, snapping it shut and completing the circuit.

吸引金属臂，从而闭合电路

You can think of a relay like a water faucet.

你可以把继电器 想成水龙头

The control wire is like the faucet handle.

把控制线路 想成水龙头把

Open the faucet, and water flows through the pipe.

打开水龙头，水会流出来

Close the faucet, and the flow of water stops.

关闭水龙头，水就没有了

Relays are doing the same thing, just with electrons instead of water.

继电器是一样的，只不过控制的是电子 而不是水

The controlled circuit can then connect to other circuits, or to something like a motor,

这个控制电路可以连到其他电路，比如马达

which might increment a count on a gear,

马达让计数齿轮 +1

like in Hollerith's tabulating machine we talked about last episode.

就像上集中 Hollerith 的制表机一样

Unfortunately, the mechanical arm inside of a relay \*has mass\*,

不幸的是，继电器内的机械臂 \*有质量\*

and therefore can't move instantly between opened and closed states.

因此无法快速开关

A good relay in the 1940's might be able to flick back and forth fifty times in a second.

1940 年代一个好的继电器 1 秒能翻转 50 次

That might seem pretty fast, but it's not fast enough to be useful at solving large, complex problems.

看起来好像很快，但还不够快，不足以解决复杂的大问题

The Harvard Mark I could do 3 additions or subtractions per second;

哈佛马克一号，1 秒能做 3 次加法或减法运算

multiplications took 6 seconds, and divisions took 15.

一次乘法要花 6 秒，除法要花 15 秒

And more complex operations, like a trigonometric function, could take over a minute.

更复杂的操作 比如三角函数，可能要一分钟以上

In addition to slow switching speed, another limitation was wear and tear.

除了速度慢，另一个限制是齿轮磨损

Anything mechanical that moves will wear over time.

任何会动的机械都会随时间磨损

Some things break entirely, and other things start getting sticky, slow, and just plain unreliable.

有些部件会完全损坏，有些则是变黏，变慢，变得不可靠

And as the number of relays increases, the probability of a failure increases too.

并且随着继电器数量增加，故障概率也会增加

The Harvard Mark I had roughly 3500 relays.

哈佛马克一号 有大约 3500 个继电器

Even if you assume a relay has an operational life of 10 years,

哪怕假设继电器的使用寿命是 10 年

this would mean you'd have to replace, on average, one faulty relay every day!

也意味着平均每天得换一个故障继电器！

That's a big problem when you are in the middle of running some important, multi-day calculation.

这个问题很严重，因为有些重要运算要运行好几天

And that's not all engineers had to contend with.

而且还有更多其他问题要考虑

These huge, dark, and warm machines also attracted insects.

这些巨大，黑色，温暖的机器也会吸引昆虫

In September 1947, operators on the Harvard Mark II pulled a dead moth from a malfunctioning relay.

1947年9月，哈佛马克2型的操作员从故障继电器中，拔出一只死虫

Grace Hopper who we'll talk more about in a later episode noted,

Grace Hopper（这位我们以后还会提到）曾说

"From then on, when anything went wrong with a computer,

"从那时起，每当电脑出了问题，

we said it had bugs in it."

我们就说它出了 bug（虫子）"

And that's where we get the term computer bug.

这就是术语 "bug" 的来源

It was clear that a faster, more reliable alternative to electro-mechanical relays was needed

显然，如果想进一步提高计算能力

if computing was going to advance further,

我们需要更快更可靠的东西，来替代继电器

and fortunately that alternative already existed!

幸运的是，替代品已经存在了！

In 1904, English physicist John Ambrose Fleming

在 1904 年，英国物理学家 "约翰·安布罗斯·弗莱明"

developed a new electrical component called a thermionic valve,

开发了一种新的电子组件，叫"热电子管"

which housed two electrodes inside an airtight glass bulb

把两个电极装在一个气密的玻璃灯泡里

- this was the first vacuum tube.

-这是世上第一个真空管

One of the electrodes could be heated, which would cause it to emit electrons

其中一个电极可以加热，从而发射电子

– a process called thermionic emission.

-这叫 "热电子发射"

The other electrode could then attract these electrons to create the flow of our electric faucet,

另一个电极会吸引电子，形成"电龙头"的电流

but only if it was positively charged

但只有带正电才行

- if it had a negative or neutral charge, the electrons would no longer be attracted across the vacuum

- 如果带负电荷或中性电荷，电子就没办法被吸引，越过真空区域

so no current would flow.

因此没有电流

An electronic component that permits the one-way flow of current is called a diode,

电流只能单向流动的电子部件叫 "二极管"

but what was really needed was a switch to help turn this flow on and off.

但我们需要的是，一个能开关电流的东西

Luckily, shortly after, in 1906, American inventor Lee de Forest

幸运的是，不久之后在 1906 年，美国发明家 "李·德富雷斯特"

added a third "control" electrode that sits between the two electrodes in Fleming's design.

他在"弗莱明"设计的两个电极之间，加入了第三个 "控制" 电极

By applying a positive charge to the control electrode, it would permit the flow of electrons as before.

向"控制"电极施加正电荷，它会允许电子流动

But if the control electrode was given a negative charge,

但如果施加负电荷

it would prevent the flow of electrons.

它会阻止电子流动

So by manipulating the control wire, one could open or close the circuit.

因此通过控制线路，可以断开或闭合电路

It's pretty much the same thing as a relay

和继电器的功能一样

- but importantly, vacuum tubes have no moving parts.

- 但重要的是，真空管内没有会动的组件

This meant there was less wear,

这意味着更少的磨损

and more importantly, they could switch thousands of times per second.

更重要的是，每秒可以开闭数千次

These triode vacuum tubes would become the basis of radio, long distance telephone,

因此这些"三极真空管"成为了无线电，长途电话

and many other electronic devices for nearly a half century.

以及其他电子设备的基础，持续了接近半个世纪

I should note here that vacuum tubes weren't perfect

我应该提到，真空管不是完美的

- they're kind of fragile, and can burn out like light bulbs,

-它们有点脆弱，并且像灯泡一样会烧坏

they were a big improvement over mechanical relays.

但比起机械继电器是一次巨大进步

Also, initially vacuum tubes were expensive

起初，真空管非常昂贵

– a radio set often used just one,

收音机一般只用一个

but a computer might require hundreds or thousands of electrical switches.

但计算机可能要上百甚至上千个电气开关

But by the 1940s,

但到了 1940 年代

their cost and reliability had improved to the point where they became feasible for use in computers….

它的成本和可靠性得到改进，可以用在计算机里

at least by people with deep pockets, like governments.

至少有钱人负担得起，比如政府

This marked the shift from electro-mechanical computing to electronic computing.

这标志着计算机 从机电转向电子

Let's go to the Thought Bubble.

我们来进入思想泡泡

The first large-scale use of vacuum tubes for computing was the Colossus MK 1,

第一个大规模使用真空管的计算机是 "巨人1号"

designed by engineer Tommy Flowers and completed in December of 1943.

由工程师 Tommy Flowers 设计，完工于1943年12月

The Colossus was installed at Bletchley Park, in the UK,

巨人1号 在英国的"布莱切利园", 用于破解纳粹通信

and helped to decrypt Nazi communications.

巨人1号 在英国的"布莱切利园", 用于破解纳粹通信

This may sound familiar because two years prior Alan Turing,

听起来可能有点熟，因为 2 年前 阿兰·图灵

often called the father of computer science,

他经常被称为"计算机科学之父"

had created an electromechanical device, also at Bletchley Park, called the Bombe.

图灵也在"布莱切利园"做了台机电装置，叫 "Bombe"

It was an electromechanical machine designed to break Nazi Enigma codes,

这台机器的设计目的是 破解纳粹"英格码"通讯加密设备

but the Bombe wasn't technically a computer,

但 Bombe 严格来说不算计算机

and we'll get to Alan Turing's contributions later.

我们之后会讨论"阿兰·图灵"的贡献

Anyway, the first version of Colossus contained 1,600 vacuum tubes,

总之，巨人1号有 1600 个真空管

and in total, ten Colossi were built to help with code-breaking.

总共造了 10 台巨人计算机，来帮助破解密码

Colossus is regarded as the first programmable, electronic computer.

巨人 被认为是第一个可编程的电子计算机

Programming was done by plugging hundreds of wires into plugboards,

编程的方法是把几百根电线插入插板

sort of like old school telephone switchboards,

有点像老电话交换机

in order to set up the computer to perform the right operations.

这是为了让计算机执行正确操作

So while "programmable", it still had to be configured to perform a specific computation.

虽然"可编程" ，但还是要配置它

Enter the The Electronic Numerical Integrator and Calculator - or ENIAC -

电子数值积分计算机 "ENIAC"

completed a few years later in 1946 at the University of Pennsylvania.

几年后在 1946 年，在"宾夕法尼亚大学"完成建造

Designed by John Mauchly and J. Presper Eckert,

设计者是 John Mauchly 和 J. Presper Eckert

this was the world's first truly general purpose, programmable, electronic computer.

这是世上第一个真正的通用，可编程，电子计算机

ENIAC could perform 5000 ten-digit additions or subtractions per second,

ENIAC 每秒可执行 5000 次十位数加减法

many, many times faster than any machine that came before it.

比前辈快了很多倍

It was operational for ten years,

它运作了十年

and is estimated to have done more arithmetic than the entire human race up to that point.

据估计，它完成的运算，比全人类加起来还多

But with that many vacuum tubes failures were common,

因为真空管很多，所以故障很常见

and ENIAC was generally only operational for about half a day at a time before breaking down.

ENIAC 运行半天左右就会出一次故障

Thanks Thought Bubble.

谢了 思想泡泡

By the 1950's, even vacuum-tube-based computing was reaching its limits.

到 1950 年代，真空管计算机都达到了极限

The US Air Force's AN/FSQ-7 computer, which was completed in 1955,

美国空军的 AN/FSQ-7 计算机于 1955 年完成

was part of the "SAGE" air defense computer system,

是 "SAGE" 防空计算机系统的一部分

which we'll talk more about in a later episode.

之后的视频还会提到.

To reduce cost and size, as well as improve reliability and speed,

为了降低成本和大小，同时提高可靠性和速度

a radical new electronic switch would be needed.

我们需要一种新的电子开关

In 1947, Bell Laboratory scientists John Bardeen, Walter Brattain, and William Shockley

1947 年，贝尔实验室科学家 \N John Bardeen，Walter Brattain，William Shockley

invented the transistor,

发明了晶体管

and with it, a whole new era of computing was born!

一个全新的计算机时代诞生了！

The physics behind transistors is pretty complex, relying on quantum mechanics,

晶体管的物理学相当复杂，牵扯到量子力学

so we're going to stick to the basics.

所以我们只讲基础

A transistor is just like a relay or vacuum tube

晶体管 就像之前提过的"继电器"或"真空管"

- it's a switch that can be opened or closed by applying electrical power via a control wire.

-它是一个开关，可以用控制线路来控制开或关

Typically, transistors have two electrodes separated by a material that sometimes can conduct electricity,

晶体管有两个电极，\N 电极之间有一种材料隔开它们，这种材料有时候导电

and other times resist it

有时候不导电

- a semiconductor.

- 这叫"半导体"

In this case, the control wire attaches to a "gate" electrode.

控制线连到一个 "门" 电极

By changing the electrical charge of the gate,

通过改变 "门" 的电荷

the conductivity of the semiconducting material can be manipulated,

我们可以控制半导体材料的导电性

allowing current to flow or be stopped

来允许或不允许 电流流动

- like the water faucet analogy we discussed earlier.

- 就像之前的水龙头比喻

Even the very first transistor at Bell Labs showed tremendous promise

贝尔实验室的第一个晶体管就展示了巨大的潜力

- it could switch between on and off states 10,000 times per second.

每秒可以开关 10,000 次

Further, unlike vacuum tubes made of glass and with carefully suspended, fragile components,

而且，比起玻璃制成，小心易碎的真空管

transistors were solid material known as a solid state component.

晶体管是固态的

Almost immediately, transistors could be made smaller than the smallest possible relays or vacuum tubes.

晶体管可以远远小于继电器或真空管

This led to dramatically smaller and cheaper computers, like the IBM 608, released in 1957

导致更小更便宜的计算机，比如1957年发布的IBM 608

– the first fully transistor-powered, commercially-available computer.

- 第一个完全用晶体管，而且消费者也可以买到的计算机

It contained 3000 transistors and could perform 4,500 additions,

它有 3000 个晶体管，每秒执行 4500 次加法

or roughly 80 multiplications or divisions, every second.

每秒能执行 80 次左右的乘除法

IBM soon transitioned all of its computing products to transistors,

IBM 很快把所有产品都转向了晶体管

bringing transistor-based computers into offices, and eventually, homes.

把晶体管计算机带入办公室，最终引入家庭

Today, computers use transistors that are smaller than 50 nanometers in size

如今，计算机里的晶体管小于 50 纳米

- for reference, a sheet of paper is roughly 100,000 nanometers thick.

- 而一张纸的厚度大概是 10 万纳米

And they're not only incredibly small, they're super fast

晶体管不仅小，还超级快

- they can switch states millions of times per second, and can run for decades.

- 每秒可以切换上百万次，并且能工作几十年

A lot of this transistor and semiconductor development happened

很多晶体管和半导体的开发在"圣克拉拉谷"

in the Santa Clara Valley, between San Francisco and San Jose, California.

这个地方在加州，位于"旧金山"和"圣荷西"之间

As the most common material used to create semiconductors is silicon,

而生产半导体最常见的材料是 "硅"

this region soon became known as Silicon Valley.

所以这个地区被称为 "硅谷"

Even William Shockley moved there, founding Shockley Semiconductor,

甚至 William Shockley 都搬了过去，创立了"肖克利半导体"

whose employees later founded Fairchild Semiconductors,

里面的员工后来成立了"仙童半导体"

whose employees later founded Intel - the world's largest computer chip maker today.

这里面的员工后来创立了英特尔 - 当今世界上最大的计算机芯片制造商

Ok, so we've gone from relays to vacuum tubes to transistors.

好了，我们从"继电器"到"真空管"到"晶体管"

We can turn electricity on and off really, really, really fast.

我们可以让电路开闭得非常非常快

But how do we get from transistors to actually computing something,

但我们是如何用晶体管做计算的？

especially if we don't have motors and gears?

我们没有马达和齿轮啊？

That's what we're going to cover over the next few episodes.

我们接下来几集会讲

Thanks for watching. See you next week.

感谢观看 下周见

Hi, I'm Carrie Anne and welcome to Crash Course Computer Science!

嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Today we start our journey up the ladder of abstraction,

今天我们开始"抽象"的旅程

where we leave behind the simplicity of being able to see every switch and gear,

不用管底层细节，把精力用来构建更复杂的系统

but gain the ability to assemble increasingly complex systems.

不用管底层细节，把精力用来构建更复杂的系统

Last episode, we talked about how computers evolved from electromechanical devices,

上集，我们谈了计算机最早是机电设备

that often had decimal representations of numbers

一般用十进制计数

- like those represented by teeth on a gear

- 比如用齿轮数来代表十进制

- to electronic computers with transistors that can turn the flow of electricity on or off.

- 再到晶体管计算机

And fortunately, even with just two states of electricity,

幸运的是，只用 开/关 两种状态也可以代表信息

we can represent important information.

幸运的是，只用 开/关 两种状态也可以代表信息

We call this representation Binary

这叫 二进制

-- which literally means "of two states",

- 意思是"用两种状态表示"

in the same way a bicycle has two wheels or a biped has two legs.

就像自行车有两个轮，双足动物有两条腿

You might think two states isn't a lot to work with, and you'd be right!

你可能觉得两种状态不多，你是对的！

But, it's exactly what you need for representing the values "true" and "false".

但如果只需要表示 true 和 false，两个值就够了

In computers, an "on" state, when electricity is flowing, represents true.

电路闭合，电流流过，代表 "真"

The off state, no electricity flowing, represents false.

电路断开，无电流流过，代表"假"

We can also write binary as 1's and 0's instead of true's and false's

二进制也可以写成 1 和 0 而不是 true 和 false

- they are just different expressions of the same signal

- 只是不同的表达方式罢了

- but we'll talk more about that in the next episode.

- 我们下集会讲更多细节

Now it is actually possible to use transistors for more than just turning electrical current on and off,

晶体管的确可以不只是 开/关，还可以让不同大小的电流通过

and to allow for different levels of current.

晶体管的确可以不只是 开/关，还可以让不同大小的电流通过

Some early electronic computers were ternary, that's three states,

一些早期电子计算机是三进制的，有 3 种状态

and even quinary, using 5 states.

甚至五进制，5 种状态

The problem is, the more intermediate states there are,

问题是，状态越多，越难区分信号

the harder it is to keep them all seperate

问题是，状态越多，越难区分信号

-- if your smartphone battery starts running low or there's electrical noise

- 如果手机快没电了或者附近有电噪音

because someone's running a microwave nearby,

因为有人在用微波炉，

the signals can get mixed up...

信号可能会混在一起...

and this problem only gets worse with transistors changing states millions of times per second!

而每秒百万次变化的晶体管会让这个问题变得更糟！

So, placing two signals as far apart as possible

所以我们把两种信号尽可能分开

- using just 'on and off' - gives us the most distinct signal to minimize these issues.

- 只用"开"和"关"两种状态，可以尽可能减少这类问题

Another reason computers use binary

计算机用二进制的另一个原因是

is that an entire branch of mathematics already existed that dealt exclusively with true and false values.

有一整个数学分支存在，专门处理"真"和"假"

And it had figured out all of the necessary rules and operations for manipulating them.

它已经解决了所有法则和运算

It's called Boolean Algebra!

叫"布尔代数"！

George Boole, from which Boolean Algebra later got its name,

乔治·布尔（George Boole）是布尔二字的由来

was a self-taught English mathematician in the 1800s.

是一位 19 世纪自学成才的英国数学家

He was interested in representing logical statements that went "under, over, and beyond"

他有兴趣用数学式子 扩展亚里士多德基于哲学的逻辑方法

Aristotle's approach to logic, which was, unsurprisingly, grounded in philosophy.

他有兴趣用数学式子 扩展亚里士多德基于哲学的逻辑方法

Boole's approach allowed truth to be systematically and formally proven, through logic equations

布尔用 逻辑方程 系统而正式的证明真理(truth)

which he introduced in his first book, "The Mathematical Analysis of Logic" in 1847.

他在 1847 年的第一本书"逻辑的数学分析"中介绍过

In "regular" algebra -- the type you probably learned in high school -- the values of variables

在"常规"代数里 - 你在高中学的那种 - 变量的值

are numbers, and operations on those numbers are things like addition and multiplication.

是数字，可以进行加法或乘法之类的操作

But in Boolean Algebra, the values of variables are true and false, and the operations are logical.

但在布尔代数中，变量的值是 true 和 false，\N 能进行逻辑操作

There are three fundamental operations in Boolean Algebra: a NOT, an AND, and an OR operation.

布尔代数中有三个基本操作：NOT, AND 和 OR

And these operations turn out to be really useful so we're going to look at them individually.

这些操作非常有用，我们一个个来看

A NOT takes a single boolean value, either true or false, and negates it.

NOT 操作把布尔值反转，\N把 true 进行 NOT 就会变成 false，反之亦然

It flips true to false, and false to true.

NOT 操作把布尔值反转，\N把 true 进行 NOT 就会变成 false，反之亦然

We can write out a little logic table that shows the original value under Input,

我们可以根据 NOT 操作的输入和输出，做出这个表

and the outcome after applying the operation under Output.

我们可以根据 NOT 操作的输入和输出，做出这个表

Now here's the cool part -- we can easily build boolean logic out of transistors.

酷的地方是 - 用晶体管可以轻松实现这个逻辑

As we discussed last episode, transistors are really just little electrically controlled switches.

上集说过，晶体管只是电控制的开关

They have three wires: two electrodes and one control wire.

有 3 根线：2 根电极和 1 根控制线

When you apply electricity to the control wire,

控制线通电时

it lets current flow through from one electrode, through the transistor, to the other electrode.

电流就可以从一个电极流到另一个电极

This is a lot like a spigot on a pipe

就像水龙头一样

-- open the tap, water flows,

- 打开水龙头，就有水流出来

close the tap, water shuts off.

关掉水龙头，就没水了

You can think of the control wire as an input,

可以把控制线，当做输入 ( input ) \N 底部的电极，当做输出（output）

and the wire coming from the bottom electrode as the output.

可以把控制线，当做输入 ( input ) \N 底部的电极，当做输出（output）

So with a single transistor, we have one input and one output.

所以 1 个晶体管，有一个输入和一个输出

If we turn the input on, the output is also on because the current can flow through it.

如果我们打开输入（input on) \N 输出也会打开（output on） \N 因为电流可以流过

If we turn the input off, the output is also off and the current can no longer pass through.

如果关闭输入（input off） \N 输出也会关闭（output off）\N 因为电流无法通过

Or in boolean terms, when the input is true, the output is true.

或者用布尔术语来说\N 输入为 真，输出为 真

And when the input is false, the output is also false.

输入为 假 \N 输出为 假

Which again we can show on a logic table.

我们也可以把这个做成"真值表"

This isn't a very exciting circuit though because its not doing anything

这个电路没什么意思，因为它没做什么事

-- the input and output are the same.

- 输入和输出是一样的

But, we can modify this circuit just a little bit to create a NOT.

但我们可以稍加修改，实现 NOT

Instead of having the output wire at the end of the transistor, we can move it before.

与其把下面那根线当做 输出，我们可以把 输出 放到上面

If we turn the input on, the transistor allows current to pass through it to the "ground",

如果打开 输入，电流可以流过然后 "接地"

and the output wire won't receive that current

输出就没有电流，所以输出是 off

- so it will be off.

输出就没有电流，所以输出是 off

In our water metaphor grounding would be like

如果用水来举例

if all the water in your house was flowing out of a huge hose

就像家里的水都从一个大管子流走了

so there wasn't any water pressure left for your shower.

打开淋浴头一点水也没有

So in this case if the input is on, output is off.

如果输入是 on，输出是 off

When we turn off the transistor, though, current is prevented from flowing down it to the ground,

当输入是 off，电流没法接地，就流过了输出，所以输出是 on

so instead, current flows through the output wire.

当输入是 off，电流没法接地，就流过了输出，所以输出是 on

So the input will be off and the output will be on.

如果输入是 off，输出是 on

And this matches our logic table for NOT, so congrats, we just built a circuit that computes NOT!

和 NOT 操作表一样！太棒了！我们做了个有点用的电路！

We call them NOT gates - we call them gates because they're controlling the path of our current.

我们叫它 "NOT 门" \N 之所以叫 "门"，是因为它能控制电流的路径

The AND Boolean operation takes two inputs, but still has a single output.

"AND"操作有 2 个输入，1 个输出

In this case the output is only true if both inputs are true.

如果 2 个输入都是 true，输出才是 true

Think about it like telling the truth.

你可以想成是 说真话

You're only being completely honest if you don't lie even a little.

如果完全不说谎，才是诚实

For example, let's take the statement,

举例，看如下这个句子

"My name is Carrie Anne AND I'm wearing a blue dress".

我叫 Carrie Anne "而且"我穿着蓝色的衣服

Both of those facts are true, so the whole statement is true.

2 个都是真的，所以整个是真的

But if I said, "My name is Carrie Anne AND I'm wearing pants" that would be false,

但如果说，我叫 Carrie Anne"而且"我穿了裤子, 就是假的

because I'm not wearing pants.

因为我没穿裤子

Or trousers.

或长裤，如果你是英国人你会用这个词……（英/美单词不同梗）

If you're in England.

或长裤，如果你是英国人你会用这个词……（英/美单词不同梗）

The Carrie Anne part is true, but a true AND a false, is still false.

虽然前半句是真的，但是 真 "AND" 假，还是假

If I were to reverse that statement it would still obviously be false,

就算把前后顺序反过来，也依然是 假

and if I were to tell you two complete lies that is also false,

如果我说 2 个假的事情，那么结果是假。

and again we can write all of these combinations out in a table.

和上次一样，可以给"AND"做个表

To build an AND gate, we need two transistors connected together

为了实现 "AND 门"，我们需要 2 个晶体管连在一起

so we have our two inputs and one output.

这样有 2 个输入和 1 个输出

If we turn on just transistor A, current won't flow because the current is stopped by transistor B.

如果只打开 A，不打开 B \N 电流无法流到 output，所以输出是 false

Alternatively, if transistor B is on, but the transistor A is off,

如果只打开 B，不打开 A ，也一样，电流无法流到 output

the same thing, the current can't get through.

如果只打开 B，不打开 A ，也一样，电流无法流到 output

Only if transistor A AND transistor B are on does the output wire have current.

只有 A 和 B 都打开了，output 才有电流

The last boolean operation is OR

最后一个是 OR （前面讲了 NOT 和 AND)

-- where only one input has to be true for the output to be true.

只要 2 个输入里，其中 1 个是 true，输出就是 true

For example, my name is Margaret Hamilton OR I'm wearing a blue dress.

比如，我叫 Margaret Hamilton"或"我穿着蓝色衣服

This is a true statement because although I'm not Margaret Hamilton unfortunately,

结果是 true，虽然我不是 Margaret Hamilton

I am wearing a blue dress, so the overall statement is true.

但是我穿着蓝色衣服，所以结果是 true

An OR statement is also true if both facts are true.

对于"OR 操作"来说，\N如果 2 个 输入都是 true，输出也是 true

The only time an OR statement is false is if both inputs are false.

只有 2 个输入都是 false，OR 的结果才是 false

Building an OR gate from transistors needs a few extra wires.

实现 "OR 门" 除了晶体管还要额外的线

Instead of having two transistors in series -- one after the other --

不是串联起来。而是并联

we have them in parallel.

不是串联起来。而是并联

We run wires from the current source to both transistors.

然后左边这条线有电流输入

We use this little arc to note that the wires jump over one another and aren't connected,

我们用"小拱门"代表 2 条线没连在一起，只是跨过而已

even though they look like they cross.

虽然看起来像连在一起

If both transistors are turned off, the current is prevented from flowing to the output,

如果 A 和 B 都是 off，电流无法流过

so the output is also off.

所以输出是 off

Now, if we turn on just Transistor A, current can flow to the output.

如果打开 A，电流可以流过。输出是 on

Same thing if transistor A is off, but Transistor B in on.

如果只打开 B 也一样

Basically if A OR B is on, the output is also on.

只要 A OR B 是 on， 输出就是 on

Also, if both transistors are on, the output is still on.

如果 A 和 B 都 on，结果是 on

Ok, now that we've got NOT, AND, and OR gates,

好，现在 NOT 门, AND 门, OR 门 都搞定了

and we can leave behind the constituent transistors and move up a layer of abstraction.

我们可以进行一次抽象

The standard engineers use for these gates are a triangle with a dot for a NOT,

NOT 门的画法是三角形前面一个圆点

a D for the AND, and a spaceship for the OR.

AND 门用 D 表示 ，OR 门用太空船表示

Those aren't the official names, but that's howI like to think of them.

"D 形状和太空船"不是标准叫法, 只是我喜欢这样叫而已

Representing them and thinking about them this way allows us to build even bigger components

我们可以用这种方法表示它们，构建更大的组件

while keeping the overall complexity relatively the same

就不会变得很复杂

- just remember that that mess of transistors and wires is still there.

- 晶体管和电线依然在那里，我们只是用符号来代表而已

For example, another useful boolean operation in computation is called an Exclusive OR

除了前面说的三个\N 另一个有用的布尔操作叫 "异或"

- or XOR for short.

- 简称 XOR

XOR is like a regular OR, but with one difference:

XOR 就像普通 OR，但有一个区别：

if both inputs are true, the XOR is false.

如果 2 个输入都是 true，XOR 输出 false

The only time an XOR is true is when one input is true and the other input is false.

想要 XOR 输出 true \N 一个输入必须是 true，另一个必须是 false

It's like when you go out to dinner and your meal comes with a side salad OR a soup

就像你出去吃晚饭，你点的饭要么配沙拉，要么配汤

- sadly, you can't have both!

- 你不能两个都要！

And building this from transistors is pretty confusing,

用晶体管实现 XOR 门有点烧脑子

but we can show how an XOR is created from our three basic boolean gates.

但我可以展示一下\N 怎么用前面提到的 3 种门来做 XOR 门

We know we have two inputs again -- A and B -- and one output.

我们有 2 个输入，A 和 B ，还有 1 个输出.

Let's start with an OR gate, since the logic table looks almost identical to an OR.

我们先放一个 OR 门. 因为 OR 和 XOR 的逻辑表很像

There's only one problem - when A and B are true, the logic is different from OR,

只有 1 个问题 - 当 A 和 B 都是 true 时 \N OR 的输出和想要的 XOR 输出不一样

and we need to output "false".

我们想要 false

And XOR turns out to be a very useful component,

XOR 超有用的

and we'll get to it in another episode,

我们下次再说它

so useful in fact engineers gave it its own symbol too -- an OR gate with a smile :)

因为超有用，\N 工程师给了它一个符号，一个 OR 门 + 一个笑脸

But most importantly, we can now put XOR into our metaphorical toolbox

重要的是，现在可以把 XOR 放入"工具箱"了

and not have to worry about the individual logic gates that make it up,

不用担心 XOR 具体用了几个门

or the transistors that make up those gates,

这几个门又是怎么用晶体管拼的

or how electrons are flowing through a semiconductor.

或电子是怎么流过半导体的

Moving up another layer of abstraction.

再次向上抽象

When computer engineers are designing processors, they rarely work at the transistor level,

工程师设计处理器时，很少在晶体管的层面上思考，

and instead work with much larger blocks, like logic gates, and even larger components

而是用更大的组件，比如逻辑门，或者由逻辑门组成的更大组件，

made up of logic gates, which we'll discuss in future episodes.

我们以后会讲

And even if you are a professional computer programmer,

就算是专业程序员

it's not often that you think about

也不用考虑逻辑是怎样在物理层面实现的

how the logic that you are programming is actually implemented

也不用考虑逻辑是怎样在物理层面实现的

in the physical world by these teeny tiny components.

也不用考虑逻辑是怎样在物理层面实现的

We've also moved from thinking about raw electrical signals to our first representation of data

我们从电信号开始，到现在第一次表示数据

- true and false - and we've even gotten a little taste of computation.

- 真和假 - 开始有点"计算"的感觉了

With just the logic gates in this episode,

仅用这集讲的 逻辑门

we could build a machine that evaluates complex logic statements,

我们可以判断复杂的语句 比如：

like if "Name is John Green AND after 5pm OR is Weekend AND near Pizza Hut",

[如果是 John Green] AND [下午 5 点后] \N OR [周末] AND [在比萨店附近]

then "John will want pizza" equals true.

那么 "John 想要比萨" = 真

And with that, I'm starving, I'll see you next week.

我都说饿了，下周见

Hi I'm Carrie Anne, this is Crash Course Computer Science

嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

and today we're going to talk about how computers store and represent numerical data.

今天，我们讲计算机如何存储和表示数字

Which means we've got to talk about Math!

所以会有一些数学

But don't worry.

不过别担心

Every single one of you already knows exactly what you need to know to follow along.

你们的数学水平绝对够用了

So, last episode we talked about how transistors can be used to build logic gates,

上集我们讲了，怎么用晶体管做逻辑门

which can evaluate boolean statements.

逻辑门可以判断布尔语句

And in boolean algebra, there are only two, binary values: true and false.

布尔代数只有两个值：True 和 False

But if we only have two values,

但如果只有两个值，我们怎么表达更多东西？

how in the world do we represent information beyond just these two values?

但如果只有两个值，我们怎么表达更多东西？

That's where the Math comes in.

这就需要数学了

So, as we mentioned last episode, a single binary value can be used to represent a number.

上集提到，1 个二进制值可以代表 1 个数

Instead of true and false, we can call these two states 1 and 0 which is actually incredibly useful.

我们可以把真和假 ，当做 1 和 0

And if we want to represent larger things we just need to add more binary digits.

如果想表示更多东西，加位数就行了

This works exactly the same way as the decimal numbers that we're all familiar with.

和我们熟悉的十进制一样

With decimal numbers there are "only" 10 possible values a single digit can be; 0 through 9,

十进制只有 10 个数（0到9）

and to get numbers larger than 9 we just start adding more digits to the front.

要表示大于 9 的数，加位数就行了

We can do the same with binary.

二进制也可以这样玩

For example, let's take the number two hundred and sixty three.

拿 263 举例

What does this number actually represent?

这个数字 "实际" 代表什么？

Well, it means we've got 2 one-hundreds, 6 tens, and 3 ones.

2 个 100 \N6 个 10 \N 3 个 1

If you add those all together, we've got 263.

加在一起，就是 263

Notice how each column has a different multiplier.

注意每列有不同的乘数

In this case, it's 100, 10, and 1.

100, 10, 1

Each multiplier is ten times larger than the one to the right.

每个乘数都比右边大十倍

That's because each column has ten possible digits to work with, 0 through 9,

因为每列有 10 个可能的数字（0到9）

after which you have to carry one to the next column.

如果超过 9，要在下一列进 1.

For this reason, it's called base-ten notation, also called decimal since deci means ten.

因此叫 "基于十的表示法" 或十进制

AND Binary works exactly the same way, it's just base-two.

二进制也一样，只不过是基于 2 而已

That's because there are only two possible digits in binary - 1 and 0.

因为二进制只有两个可能的数， 1 和 0

This means that each multiplier has to be two times larger than the column to its right.

意味着每个乘数必须是右侧乘数的两倍

Instead of hundreds, tens, and ones, we now have fours, twos and ones.

就不是之前的 100, 10, 1 \N 而是 4, 2, 1

Take for example the binary number: 101.

拿二进制数 101 举例

This means we have 1 four, 0 twos, and 1 one.

意味着有\N 1个 "4" \N 0个 "2" \N 1个 "1"

Add those all together and we've got the number 5 in base ten.

加在一起，得到十进制的 5

But to represent larger numbers, binary needs a lot more digits.

为了表示更大的数字，二进制需要更多位数

Take this number in binary 10110111.

拿二进制数 10110111 举例

We can convert it to decimal in the same way.

我们可以用相同的方法转成十进制

We have 1 x 128, 0 x 64, 1 x 32, 1 x 16, 0 x 8, 1 x 4, 1 x 2, and 1 x 1.

1 x 128 ，0 x 64 ，1 x 32 ，1 x 16 \N 0 x 8 ，1 x 4 ，1 x 2 ，1 x 1

Which all adds up to 183.

加起来等于 183

Math with binary numbers isn't hard either.

二进制数的计算也不难

Take for example decimal addition of 183 plus 19.

以十进制数 183 加 19 举例

First we add 3 + 9, that's 12, so we put 2 as the sum and carry 1 to the ten's column.

首先 3 + 9，得到 12，然后位数记作 2，向前进 1

Now we add 8 plus 1 plus the 1 we carried, thats 10, so the sum is 0 carry 1.

现在算 8+1+1=10，所以位数记作0，再向前进 1

Finally we add 1 plus the 1 we carried, which equals 2.

最后 1+1=2，位数记作2

So the total sum is 202.

所以和是202

Here's the same sum but in binary.

二进制也一样

Just as before, we start with the ones column.

和之前一样，从个位开始

Adding 1+1 results in 2, even in binary.

1+1=2，在二进制中也是如此

But, there is no symbol "2" so we use 10 and put 0 as our sum and carry the 1.

但二进制中没有 2，所以位数记作 0 ，进 1

Just like in our decimal example.

就像十进制的例子一样

1 plus 1, plus the 1 carried,

1+1，再加上进位的1

equals 3 or 11 in binary,

等于 3，用二进制表示是 11

so we put the sum as 1 and we carry 1 again, and so on.

所以位数记作 1，再进 1，以此类推

We end up with this number, which is the same as the number 202 in base ten.

最后得到这个数字，跟十进制 202 是一样的

Each of these binary digits, 1 or 0, is called a "bit".

二进制中，一个 1 或 0 叫一"位"

So in these last few examples, we were using 8-bit numbers with their lowest value of zero

上个例子我们用了 8 位 , 8 位能表示的最小数是 0, 8位都是0

and highest value is 255, which requires all 8 bits to be set to 1.

最大数是 255，8 位都是 1

Thats 256 different values, or 2 to the 8th power.

能表示 256 个不同的值，2 的 8 次方

You might have heard of 8-bit computers, or 8-bit graphics or audio.

你可能听过 8 位机，8 位图像，8 位音乐

These were computers that did most of their operations in chunks of 8 bits.

意思是计算机里\N 大部分操作都是 8 位 8 位这样处理的

But 256 different values isn't a lot to work with, so it meant things like 8-bit games

但 256 个值不算多，意味着 8位游戏只能用 256 种颜色

were limited to 256 different colors for their graphics.

但 256 个值不算多，意味着 8位游戏只能用 256 种颜色

And 8-bits is such a common size in computing, it has a special word: a byte.

8 位是如此常见，以至于有专门的名字：字节

A byte is 8 bits.

1 字节 = 8 位 \N 1 bytes = 8 bits

If you've got 10 bytes, it means you've really got 80 bits.

如果有 10 个字节，意味着有 80 位

You've heard of kilobytes, megabytes, gigabytes and so on.

你听过 千字节（kb）兆字节（mb）千兆字节（gb）

These prefixes denote different scales of data.

不同前缀代表不同数量级

Just like one kilogram is a thousand grams,

就像 1 千克 = 1000 克，1 千字节 = 1000 字节

1 kilobyte is a thousand bytes.

就像 1 千克 = 1000 克，1 千字节 = 1000 字节

or really 8000 bits.

或 8000 位

Mega is a million bytes (MB), and giga is a billion bytes (GB).

Mega 是百万字节（MB）, Giga 是十亿字节（GB）

Today you might even have a hard drive that has 1 terabyte (TB) of storage.

如今你可能有 1 TB 的硬盘

That's 8 trillion ones and zeros.

8 万亿个1和0

But hold on!

等等，我们有另一种计算方法

That's not always true.

等等，我们有另一种计算方法

In binary, a kilobyte has two to the power of 10 bytes, or 1024.

二进制里，1 千字节 = 2的10次方 = 1024 字节

1000 is also right when talking about kilobytes,

1000 也是千字节（KB）的正确单位

but we should acknowledge it isn't the only correct definition.

1000 和 1024 都对

You've probably also heard the term 32-bit or 64-bit computers

你可能听过 32 位 或 64 位计算机

C you're almost certainly using one right now.

你现在用的电脑肯定是其中一种

What this means is that they operate in chunks of 32 or 64 bits.

意思是一块块处理数据，每块是 32 位或 64 位

That's a lot of bits!

这可是很多位

The largest number you can represent with 32 bits is just under 4.3 billion.

32 位能表示的最大数，是 43 亿左右

Which is thirty-two 1's in binary.

也就是 32 个 1

This is why our Instagram photos are so smooth and pretty

所以 Instagram 照片很清晰

- they are composed of millions of colors,

- 它们有上百万种颜色

because computers today use 32-bit color graphics

因为如今都用 32 位颜色

Of course, not everything is a positive number

当然，不是所有数字都是正数

- like my bank account in college.

比如我上大学时的银行账户 T\_T

So we need a way to represent positive and negative numbers.

我们需要有方法表示正数和负数

Most computers use the first bit for the sign:

大部分计算机用第一位表示正负：

1 for negative, 0 for positive numbers,

1 是负，0 是正

and then use the remaining 31 bits for the number itself.

用剩下 31 位来表示数字

That gives us a range of roughly plus or minus two billion.

能表示的数字范围是 正 20 亿到负 20 亿

While this is a pretty big range of numbers, it's not enough for many tasks.

虽然是很大的数，但有时还不够用

There are 7 billion people on the earth, and the US national debt is almost 20 trillion dollars after all.

全球有 70 亿人口，美国国债近 20 万亿美元

This is why 64-bit numbers are useful.

所以 64 位数很有用

The largest value a 64-bit number can represent is around 9.2 quintillion!

64 位能表达最大数是 9.2x10

That's a lot of possible numbers and will hopefully stay above the US national debt for a while!

希望美国国债在一段时间内不会超过这个数！

Most importantly, as we'll discuss in a later episode,

重要的是（我们之后的视频会深入讲）

computers must label locations in their memory,

计算机必须给内存中每一个位置，做一个 "标记"

known as addresses, in order to store and retrieve values.

这个标记叫 "位址", 目的是为了方便存取数据

As computer memory has grown to gigabytes and terabytes - that's trillions of bytes

如今硬盘已经增长到 GB 和 TB，上万亿个字节！

it was necessary to have 64-bit memory addresses as well.

内存地址也应该有 64 位

In addition to negative and positive numbers,

除了负数和正数，计算机也要处理非整数

computers must deal with numbers that are not whole numbers,

除了负数和正数，计算机也要处理非整数

like 12.7 and 3.14, or maybe even stardate: 43989.1.

比如 12.7 和 3.14，或"星历 43989.1"

These are called "floating point" numbers,

这叫 浮点数

because the decimal point can float around in the middle of number.

因为小数点可以在数字间浮动

Several methods have been developed to represent floating point numbers.

有好几种方法 表示浮点数

The most common of which is the IEEE 754 standard.

最常见的是 IEEE 754 标准

And you thought historians were the only people bad at naming things!

你以为只有历史学家取名很烂吗？

In essence, this standard stores decimal values sort of like scientific notation.

它用类似科学计数法的方法，来存十进制值

For example, 625.9 can be written as 0.6259 x 10^3.

例如，625.9 可以写成 0.6259×10 ^ 3

There are two important numbers here: the .6259 is called the significand.

这里有两个重要数字：.6259 叫 "有效位数" , 3 是指数

And 3 is the exponent.

这里有两个重要数字：.6259 叫 "有效位数" , 3 是指数

In a 32-bit floating point number,

在 32 位浮点数中

the first bit is used for the sign of the number -- positive or negative.

第 1 位表示数字的正负

The next 8 bits are used to store the exponent

接下来 8 位存指数

and the remaining 23 bits are used to store the significand.

剩下 23 位存有效位数

Ok, we've talked a lot about numbers, but your name is probably composed of letters,

好了，聊够数字了，但你的名字是字母组成的

so it's really useful for computers to also have a way to represent text.

所以我们也要表示文字

However, rather than have a special form of storage for letters,

与其用特殊方式来表示字母 \N 计算机可以用数字表示字母

computers simply use numbers to represent letters.

与其用特殊方式来表示字母 \N 计算机可以用数字表示字母

The most straightforward approach might be to simply number the letters of the alphabet:

最直接的方法是给字母编号：

A being 1, B being 2, C 3, and so on.

A是1，B是2，C是3，以此类推

In fact, Francis Bacon, the famous English writer,

著名英国作家 弗朗西斯·培根（Francis Bacon）

used five-bit sequences to encode all 26 letters of the English alphabet

曾用 5位序列 来编码英文的 26 个字母

to send secret messages back in the 1600s.

在十六世纪传递机密信件

And five bits can store 32 possible values - so that's enough for the 26 letters,

五位（bit）可以存 32 个可能值（2^5） - 这对26个字母够了

but not enough for punctuation, digits, and upper and lower case letters.

但不能表示 标点符号，数字和大小写字母

Enter ASCII, the American Standard Code for Information Interchange.

ASCII，美国信息交换标准代码

Invented in 1963, ASCII was a 7-bit code, enough to store 128 different values.

发明于 1963 年，ASCII 是 7 位代码，足够存 128 个不同值

With this expanded range, it could encode capital letters, lowercase letters,

范围扩大之后，可以表示大写字母，小写字母,

digits 0 through 9, and symbols like the @ sign and punctuation marks.

数字 0 到 9, @ 这样的符号, 以及标点符号

For example, a lowercase 'a' is represented by the number 97, while a capital 'A' is 65.

举例，小写字母 a 用数字 97 表示，大写字母 A 是 65

A colon is 58 and a closed parenthesis is 41.

: 是58 \n ) 是41

ASCII even had a selection of special command codes,

ASCII 甚至有特殊命令符号

such as a newline character to tell the computer where to wrap a line to the next row.

比如换行符，用来告诉计算机换行

In older computer systems,

在老计算机系统中

the line of text would literally continue off the edge of the screen if you didn't include a new line character!

如果没换行符，文字会超出屏幕

Because ASCII was such an early standard,

因为 ASCII 是个很早的标准

it became widely used,

所以它被广泛使用

and critically, allowed different computers built by different companies to exchange data.

让不同公司制作的计算机，能互相交换数据

This ability to universally exchange information is called "interoperability".

这种通用交换信息的能力叫 "互用性"

However, it did have a major limitation: it was really only designed for English.

但有个限制：它是为英语设计的

Fortunately, there are 8 bits in a byte, not 7,

幸运的是，一个字节有8位，而不是7位

and it soon became popular to use codes 128 through 255,

128 到 255 的字符渐渐变得常用

previously unused, for "national" characters.

这些字符以前是空的，是给各个国家自己 "保留使用的"

In the US, those extra numbers were largely used to encode additional symbols,

在美国，这些额外的数字主要用于编码附加符号

like mathematical notation, graphical elements, and common accented characters.

比如数学符号，图形元素和常用的重音字符

On the other hand, while the Latin characters were used universally,

另一方面，虽然拉丁字符被普遍使用

Russian computers used the extra codes to encode Cyrillic characters,

在俄罗斯，他们用这些额外的字符表示西里尔字符

and Greek computers, Greek letters, and so on.

而希腊电脑用希腊字母，等等

And national character codes worked pretty well for most countries.

这些保留下来给每个国家自己安排的空位，\N 对大部分国家都够用

The problem was,

问题是

if you opened an email written in Latvian on a Turkish computer,

如果在 土耳其 电脑上打开 拉脱维亚语 写的电子邮件

the result was completely incomprehensible.

会显示乱码

And things totally broke with the rise of computing in Asia,

随着计算机在亚洲兴起，这种做法彻底失效了

as languages like Chinese and Japanese have thousands of characters.

中文和日文这样的语言有数千个字符

There was no way to encode all those characters in 8-bits!

根本没办法用 8 位来表示所有字符！

In response, each country invented multi-byte encoding schemes,

为了解决这个问题，每个国家都发明了多字节编码方案

all of which were mutually incompatible.

但不相互兼容

The Japanese were so familiar with this encoding problem that they had a special name for it:

日本人总是碰到编码问题，以至于专门有词来称呼：

"mojibake", which means "scrambled text".

"mojibake" 意思是 乱码

And so it was born - Unicode - one format to rule them all.

所以 Unicode 诞生了 - 统一所有编码的标准

Devised in 1992 to finally do away with all of the different international schemes

设计于 1992 年，解决了不同国家不同标准的问题

it replaced them with one universal encoding scheme.

Unicode 用一个统一编码方案

The most common version of Unicode uses 16 bits with space for over a million codes -

最常见的 Unicode 是 16 位的，有超过一百万个位置 -

enough for every single character from every language ever used

对所有语言的每个字符都够了

more than 120,000 of them in over 100 types of script

100 多种字母表加起来占了 12 万个位置。

plus space for mathematical symbols and even graphical characters like Emoji.

还有位置放数学符号，甚至 Emoji

And in the same way that ASCII defines a scheme for encoding letters as binary numbers,

就像 ASCII 用二进制来表示字母一样

other file formats - like MP3s or GIFs -

其他格式 - 比如 MP3 或 GIF -

use binary numbers to encode sounds or colors of a pixel in our photos, movies, and music.

用二进制编码声音/颜色，表示照片,电影,音乐

Most importantly, under the hood it all comes down to long sequences of bits.

重要的是，这些标准归根到底是一长串位

Text messages, this YouTube video, every webpage on the internet,

短信，这个 YouTube 视频，互联网上的每个网页

and even your computer's operating system, are nothing but long sequences of 1s and 0s.

甚至操作系统，只不过是一长串 1 和 0

So next week,

下周

we'll start talking about how your computer starts manipulating those binary sequences,

我们会聊计算机怎么操作二进制

for our first true taste of computation.

初尝"计算"的滋味

Thanks for watching. See you next week.

感谢观看，下周见

Hi, I'm Carrie Ann and this is Crash Course Computer Science.

嗨，我是 Carrie Anne，欢迎收看计算机科学速成课

So last episode, we talked about how numbers can be represented in binary.

上集，我们谈了如何用二进制表示数字

Representing Like, 00101010 is 42 in decimal.

比如二进制 00101010 是十进制的 42

Representing and storing numbers is an important function of a computer,

表示和存储数字是计算机的重要功能

but the real goal is computation, or manipulating numbers in a structured and purposeful way,

但真正的目标是计算，有意义的处理数字

like adding two numbers together.

比如把两个数字相加

These operations are handled by a computer's Arithmetic and Logic Unit,

这些操作由计算机的 "算术逻辑单元 "处理

but most people call it by its street name:

但大家会简称：ALU

the ALU.

但大家会简称：ALU

The ALU is the mathematical brain of a computer.

ALU 是计算机的数学大脑

When you understand an ALU's design and function,

等你理解了 ALU 的设计和功能之后

you'll understand a fundamental part of modern computers.

你就理解了现代计算机的基石

It is THE thing that does all of the computation in a computer,

ALU \*就是\* 计算机里负责运算的组件\N 基本其他所有部件都用到了它

so basically everything uses it.

ALU \*就是\* 计算机里负责运算的组件\N 基本其他所有部件都用到了它

First though, look at this beauty.

先来看看这个美人

This is perhaps the most famous ALU ever, the Intel 74181.

这可能是最著名的 ALU，英特尔 74181

When it was released in 1970,

1970 年发布时，它是第一个封装在单个芯片内的完整 ALU

it was It was the first complete ALU that fit entirely inside of a single chip -

1970 年发布时，它是第一个封装在单个芯片内的完整 ALU

Which was a huge engineering feat at the time.

这在当时是惊人的工程壮举

So today we're going to take those Boolean logic gates we learned about last week

今天我们用上周学的布尔逻辑门

to build a simple ALU circuit with much of the same functionality as the 74181.

做一个简单的 ALU 电路，功能和 74181 一样

And over the next few episodes we'll use this to construct a computer from scratch.

然后接下来几集，用它从头做出一台电脑

So it's going to get a little bit complicated,

所以会有点复杂

but I think you guys can handle it.

但我觉得你们搞的定

An ALU is really two units in one

ALU 有 2 个单元，1 个算术单元和 1 个逻辑单元

-- there's an arithmetic unit and a logic unit.

ALU 有 2 个单元，1 个算术单元和 1 个逻辑单元

Let's start with the arithmetic unit,

我们先讲"算术单元"，它负责计算机里的所有数字操作

which is responsible for handling all numerical operations in a computer,

我们先讲"算术单元"，它负责计算机里的所有数字操作

like addition and subtraction.

比如加减法

It also does a bunch of other simple things like add one to a number,

它还做很多其他事情，比如给某个数字+1

which is called an increment operation, but we'll talk about those later.

这叫增量运算，我们之后会说

Today, we're going to focus on the piece of rsistance, the crme de la crme of operations

今天的重点是一切的根本 - "把两个数字相加"

that underlies almost everything else a computer does - adding two numbers together.

今天的重点是一切的根本 - "把两个数字相加"

We could build this circuit entirely out of individual transistors,

我们可以用单个晶体管一个个拼，把这个电路做出来，\N 但很快就会复杂的难以理解

but that would get confusing really fast.

我们可以用单个晶体管一个个拼，把这个电路做出来，\N 但很快就会复杂的难以理解

So instead as we talked about in Episode 3

所以与其用晶体管，我们会像第 3 集

- we can use a high-level of abstraction and build our components out of logic gates,

- 用更高层的抽象，用逻辑门来做

in this case: AND, OR, NOT and XOR gates.

我们会用到 AND，OR，NOT 和 XOR 逻辑门

The simplest adding circuit that we can build takes two binary digits, and adds them together.

最简单的加法电路， \N 是拿 2 个 bit 加在一起（bit 是 0 或 1）

So we have two inputs, A and B, and one output, which is the sum of those two digits.

有 2 个输入：A 和 B， 1 个输出：就是两个数字的和

Just to clarify: A, B and the output are all single bits.

需要注意的是：A, B, 输出，这3个都是单个 Bit （ 0 或 1 ）

There are only four possible input combinations.

输入只有四种可能

The first three are: 0+0 = 0

前三个是\N 0 + 0 = 0 \N 1 + 0 = 1 \N 0 + 1 = 1

1+0 = 1 0+1 = 1

前三个是\N 0 + 0 = 0 \N 1 + 0 = 1 \N 0 + 1 = 1

Remember that in binary, 1 is the same as true, and 0 is the same as false.

记住二进制里，1 与 true 相同，0 与 false 相同

So this set of inputs exactly matches the boolean logic of an XOR gate,

这组输入和输出，和 XOR 门的逻辑完全一样

and we can use it as our 1-bit adder.

所以我们可以把 XOR 用作 1 位加法器（adder）

But the fourth input combination, 1 + 1, is a special case. 1 + 1 is 2 (obviously)

但第四个输入组合，1+1，是个特例 \N 1+1=2（显然）

but there's no 2 digit in binary,

但二进制里没有 2

so as we talked about last episode, the result is 0 and the 1 is carried to the next column.

上集说过，二进制 1+1 的结果是0，1进到下一位

So the sum is really 10 in binary.

和是 10 (二进制)

Now, the output of our XOR gate is partially correct - 1 plus 1, outputs 0.

XOR 门的输出，只对了一部分， 1+1 输出 0

But, we need an extra output wire for that carry bit.

但我们需要一根额外的线代表 "进位"

The carry bit is only "true" when the inputs are 1 AND 1,

只有输入是 1 和 1 时，进位才是 "true"

because that's the only time when the result (two) is bigger than 1 bit can store

因为算出来的结果用 1 个 bit 存不下

and conveniently we have a gate for that!

方便的是，我们刚好有个逻辑门能做这个事！

It's not that complicated - just two logic gates -

没那么复杂 - 就两个逻辑门而已

but let's abstract away even this level of detail

让我们抽象化

and encapsulate our newly minted half adder as its own component,

把 "半加器" 封装成一个单独组件

with two inputs - bits A and B - and two outputs, the sum and the carry bits.

两个输入 A 和 B 都是 1 位 \N 两个输出 "总和" 与 "进位"

This takes us to another level of abstraction

这进入了另一层抽象

heh I feel like I say that a lot.

我好像说了很多次，说不定会变成一个梗

I wonder if this is going to become a thing.

我好像说了很多次，说不定会变成一个梗

Anyway, If you want to add more than 1 + 1

如果想处理超过 1+1 的运算，我们需要"全加器"

we're going to need a "Full Adder."

如果想处理超过 1+1 的运算，我们需要"全加器"

That half-adder left us with a carry bit as output.

半加器 输出了进位

That means that when we move on to the next column in a multi-column addition,

意味着，我们算下一列的时候

and every column after that, we are going to have to add three bits together, no two.

还有之后的每一列，我们得加 3 个位在一起，并不是 2 个

A full adder is a bit more complicated

全加器复杂了一点点

全加器表格

- it takes three bits as inputs: A, B and C.

有 3 个输入：A, B, C （都是 1 个 bit)

So the maximum possible input is 1 + 1 + 1,

所以最大的可能是 1 + 1 + 1

which equals 1 carry out 1, so we still only need two output wires: sum and carry.

"总和"1 "进位"1 \N 所以要两条输出线： "总和"和"进位"

We can build a full adder using half adders.

我们可以用 半加器 做 全加器

To do this, we use a half adder to add A plus B

我们先用半加器将 A 和 B 相加

just like before - but then feed that result and input C into a second half adder.

然后把 C 输入到第二个半加器

Lastly, we need a OR gate to check if either one of the carry bits was true.

最后用一个 OR 门检查进位是不是 true

That's it, we just made a full adder!

这样就做出了一个全加器！

Again,we can go up a level of abstraction and wrap up this full adder as its own component.

我们可以再提升一层抽象，把全加器作为独立组件

It takes three inputs, adds them, and outputs the sum and the carry, if there is one.

全加器会把 A，B，C 三个输入加起来 \N 输出 "总和" 和 "进位"

Armed with our new components, we can now build a circuit that takes two, 8-bit numbers

现在有了新组件，我们可以相加两个 8 位数字

Let's call them A and B and adds them together.

叫两个数字叫 A 和 B 好了

Let's start with the very first bit of A and B,

我们从 A 和 B 的第一位开始

which we'll call A0 and B0.

叫 A0 和 B0 好了

At this point, there is no carry bit to deal with,

现在不用处理任何进位，因为是第一次加法

because this is our first addition.

现在不用处理任何进位，因为是第一次加法

So we can use our half adder to add these two bits together.

所以我们可以用半加器，来加这2个数字

The output is sum0.

输出叫 sum0

Now we want to add A1 and B1 together.

现在加 A1 和 B1

It's possible there was a carry from the previous addition of A0 and B0,

因为 A0 和 B0 的结果有可能进位

so this time we need to use a full adder that also inputs the carry bit.

所以这次要用全加器，除了 A1 和 B1，还要连上进位

We output this result as sum1.

输出叫 sum1

Then, we take any carry from this full adder,

然后，把这个全加器的进位 \N 连到下个全加器的输入，处理 A2 和 B2

and run it into the next full adder that handles A2 and B2.

然后，把这个全加器的进位 \N 连到下个全加器的输入，处理 A2 和 B2

And we just keep doing this in a big chain until all 8 bits have been added.

以此类推，把 8 个 bit 都搞定

Notice how the carry bits ripple forward to each subsequent adder.

注意每个进位是怎么连到下一个全加器的

For this reason, this is called an 8-bit ripple carry adder.

所以叫 "8位行波进位加法器"

Notice how our last full adder has a carry out.

注意最后一个全加器有 "进位" 的输出

If there is a carry into the 9th bit, it means the sum of the two numbers is too large to fit into 8-bits.

如果第 9 位有进位，代表着 2 个数字的和太大了，超过了 8 位

This is called an overflow.

这叫 "溢出" (overflow)

In general, an overflow occurs when the result of an addition is too large

一般来说 "溢出" 的意思是, 两个数字的和太大了

to be represented by the number of bits you are using.

超过了用来表示的位数

This can usually cause errors and unexpected behavior.

这会导致错误和不可预期的结果

Famously, the original PacMan arcade game used 8 bits to keep track of what level you were on.

著名的例子是，吃豆人用 8 位存当前关卡数

This meant that if you made it past level 255 - the largest number storablein 8 bits - to level 256,

如果你玩到了第 256 关（ 8 位 bit 最大表示 255）

the ALU overflowed.

ALU 会溢出

This caused a bunch of errors and glitches making the level unbeatable.

造成一连串错误和乱码，使得该关卡无法进行

The bug became a rite of passage for the greatest PacMan players.

这个 bug 成了厉害吃豆人玩家的代表

So if we want to avoid overflows,

如果想避免溢出

we can extend our circuit with more full adders, allowing us to add 16 or 32 bit numbers.

我们可以加更多全加器，可以操作 16 或 32 位数字

This makes overflows less likely to happen, but at the expense of more gates.

让溢出更难发生，但代价是更多逻辑门

An additional downside is that it takes a little bit of time for each of the carries to ripple forward.

另外一个缺点是，每次进位都要一点时间

Admittedly, not very much time, electrons move pretty fast,

当然时间不久，因为电子移动的很快

so we're talking about billionths of a second,

但如今的量级是每秒几十亿次运算，所以会造成影响

but that's enough to make a difference in today's fast computers.

但如今的量级是每秒几十亿次运算，所以会造成影响

For this reason, modern computers use a slightly different adding circuit

所以，现代计算机用的加法电路有点不同

called a 'carry-look-ahead' adder

叫 "超前进位加法器"

which is faster, but ultimately does exactly the same thing

它更快，做的事情是一样的 - 把二进制数相加

-- adds binary numbers.

它更快，做的事情是一样的 - 把二进制数相加

The ALU's arithmetic unit also has circuits for other math operations

ALU 的算术单元，也能做一些其他数学运算

and in general these 8 operations are always supported.

一般支持这 8 个操作

And like our adder, these other operations are built from individual logic gates.

就像加法器一样，这些操作也是由逻辑门构成的

Interestingly, you may have noticed that there are no multiply and divide operations.

有趣的是，你可能注意到没有乘法和除法

That's because simple ALUs don't have a circuit for this,

因为简单的 ALU 没有专门的电路来处理

and instead just perform a series of additions.

而是把乘法用多次加法来实现

Let's say you want to multiply 12 by 5.

假设想算 12x5

That's the same thing as adding 12 to itself 5 times.

这和把 "12" 加 5 次是一样的

So it would take 5 passes through the ALU to do this one multiplication.

所以要 5 次 ALU 操作来实现这个乘法

And this is how many simple processors,

很多简单处理器都是这样做的

like those in your thermostat, TV remote, and microwave, do multiplication.

比如恒温器，电视遥控器和微波炉

It's slow, but it gets the job done.

慢是慢，但是搞的定

However, fancier processors, like those in your laptop or smartphone,

然而笔记本和手机有更好的处理器

have arithmetic units with dedicated circuits for multiplication.

有专门做乘法的算术单元

And as you might expect, the circuit is more complicated than addition

你可能猜到了，乘法电路比加法复杂

-- there's no magic, it just takes a lot more logic gates

- 没什么魔法，只是更多逻辑门

which is why less expensive processors don't have this feature.

所以便宜的处理器没有.

Ok, let's move on to the other half of the ALU:

好了，我们现在讲 ALU 的另一半：逻辑单元

the Logic Unit.

好了，我们现在讲 ALU 的另一半：逻辑单元

Instead of arithmetic operations, the Logic Unit performs well...

逻辑单元执行逻辑操作

logical operations, like AND, OR and NOT, which we've talked about previously.

比如之前讨论过的 AND，OR 和 NOT 操作

It also performs simple numerical tests,

它也能做简单的数值测试

like checking if a number is negative.

比如一个数字是不是负数

For example, here's a circuit that tests if the output of the ALU is zero.

例如，这是检查 ALU 输出是否为 0 的电路

It does this using a bunch of OR gates to see if any of the bits are 1.

它用一堆 OR 门检查其中一位是否为 1

Even if one single bit is 1,

哪怕只有一个 Bit (位) 是1，

we know the number can't be zero and then we use a final NOT gate to flip this input

我们就知道那个数字肯定不是 0，然后用一个 NOT 门取反

so the output is 1 only if the input number is 0.

所以只有输入的数字是 0，输出才为 1

So that's a high level overview of what makes up an ALU.

以上就是 ALU 的一个高层次概括

We even built several of the main components from scratch, like our ripple adder.

我们甚至从零做了几个主要组件，比如行波进位加法器

As you saw, it's just a big bunch of logic gates connected in clever ways.

它们只是一大堆逻辑门巧妙的连在一起而已.

Which brings us back to that ALU you admired so much at the beginning of the episode.

让我们回到视频开始时的 ALU，英特尔 74181

The Intel 74181.

让我们回到视频开始时的 ALU，英特尔 74181

Unlike the 8-bit ALU we made today, the 74181 could only handle 4-bit inputs,

和我们刚刚做的 8 位 ALU 不同，74181 只能处理 4 位输入

which means

也就是说

YOU BUILT AN ALU THAT'S LIKE TWICE AS GOOD AS THAT SUPER FAMOUS ONE. WITH YOUR MIND!

你刚做了一个比英特尔 74181 还好的 ALU ！

Well.. sort of.

其实 差不多啦..

We didn't build the whole thing

我们虽然没有全部造出来

but you get the idea.

但你理解了整体概念

The 74181 used about 70 logic gates, and it couldn't multiply or divide.

74181 用了大概 70 个逻辑门，但不能执行乘除.

But it was a huge step forward in miniaturization,

但它向小型化迈出了一大步

opening the doors to more capable and less expensive computers.

让计算机可以更强大更便宜

This 4-bit ALU circuit is already a lot to take in,

4 位 ALU 已经要很多逻辑门了

but our 8-bit ALU would require hundreds of logic gates to fully build

但我们的 8 位 ALU 会需要数百个逻辑门

and engineers don't want to see all that complexity when using an ALU,

工程师不想在用 ALU 时去想那些事情,

so they came up with a special symbol to wrap it all up, which looks like a big V'.

所以想了一个特殊符号来代表它，看起来像一个大 "V"

Just another level of abstraction!

又一层抽象！

Our 8-bit ALU has two inputs, A and B, each with 8 bits.

我们的 8 位 ALU 有两个输入，A和B，都是 8 位 (bits)

We also need a way to specify what operation the ALU should perform,

我们还需要告诉 ALU 执行什么操作

for example, addition or subtraction.

例如加法或减法

For that, we use a 4-bit operation code.

所以我们用 4 位的操作代码

We'll talk about this more in a later episode,

我们之后的视频会再细说

but in brief, 1000 might be the command to add, while 1100 is the command for subtract.

简言之,"1000"可能代表加法命令 \N "1100"代表减法命令

Basically, the operation code tells the ALU what operation to perform.

操作代码告诉 ALU 执行什么操作

And the result of that operation on inputs A and B is an 8-bit output.

输出结果是 8 位的

ALUs also output a series of Flags,

ALU 还会输出一堆标志（Flag）

which are 1-bit outputs for particular states and statuses.

"标志"是1位的，代表特定状态.

For example, if we subtract two numbers, and the result is 0,

比如相减两个数字，结果为 0

our zero-testing circuit, the one we made earlier, sets the Zero Flag to True (1).

我们的零测试电路（前面做的）\N 会将零标志设为 True（1）

This is useful if we are trying to determine if two numbers are are equal.

如果想知道两个数字是否相等，这个非常有用

If we wanted to test if A was less than B,

如果想知道： A 是否小于 B

we can use the ALU to calculate A subtract B and look to see if the Negative Flag was set to true.

可以用 ALU 来算 A 减 B，看负标志是否为 true

If it was, we know that A was smaller than B.

如果是 true，我们就知道 A 小于 B

And finally, there's also a wire attached to the carry out on the adder we built,

最后，还有一条线连到加法器的进位

so if there is an overflow, we'll know about it.

如果有溢出，我们就知道

This is called the Overflow Flag.

这叫溢出标志

Fancier ALUs will have more flags,

高级 ALU 有更多标志

but these three flags are universal and frequently used.

但这 3 个标志是 ALU 普遍用的

In fact, we'll be using them soon in a future episode.

其实，我们之后的视频会用到它们

So now you know how your computer does all its basic mathematical operations digitally

现在你知道了\N 计算机是怎样在没有齿轮或杠杆的情况下 进行运算

with no gears or levers required.

现在你知道了\N 计算机是怎样在没有齿轮或杠杆的情况下 进行运算

We're going to use this ALU when we construct our CPU two episodes from now.

接下来两集 我们会用 ALU 做 CPU

But before that, our computer is going to need some memory!

但在此之前，计算机需要一些 "记忆" ！

We'll talk about that next week.

我们下周会讲

Hi, I'm Carrie Anne and welcome to Crash Course Computer Science.

嗨，我是 Carrie Anne，欢迎收看计算机科学速成课

So last episode, using just logic gates, we built a simple ALU,

上集，我们用逻辑门做了个简单 ALU

which performs arithmetic and logic operations, hence the 'A' and the 'L'.

它能执行算术(Arithmetic)和逻辑(Logic)运算 \N ALU 里的 A 和 L 因此得名

But of course, there's not much point in calculating a result only to throw it away

当然，算出来之后如果扔掉就没什么意义了

- it would be useful to store that value somehow,

得找个方法存起来

and maybe even run several operations in a row.

可能还要进行多个连续操作

That's where computer memory comes in!

这就用到计算机内存了

If you've ever been in the middle of a long RPG campaign on your console,

如果你在主机上打过一场长时间的对局

or slogging through a difficult level on Minesweeper on your desktop,

或玩困难模式的 "扫雷"

and your dog came by, tripped and pulled the power cord out of the wall,

然后狗跑过来，被电源线绊倒，把插头拔了出来

you know the agony of losing all your progress.

你知道失去进度的痛苦

Condolences.

真同情你 :(

But the reason for your loss is that your console, your laptop and your computers

你损失数据的原因是 \N 电脑用的是"随机存取存储器"，简称"RAM"

make use of Random Access Memory, or RAM,

你损失数据的原因是 \N 电脑用的是"随机存取存储器"，简称"RAM"

which stores things like game state - as long as the power stays on.

它只能在有电的情况下存储东西，比如游戏状态

Another type of memory, called persistent memory, can survive without power,

另一种存储 (memory) 叫持久存储，电源关闭时数据也不会丢失

and it's used for different things;

它用来存其他东西.

We'll talk about the persistence of memory in a later episode.

我们之后会讨论存储 (memory) 的持久性问题

Today, we're going to start small

今天我们从简单开始

- literally by building a circuit that can store one.. single.. bit of information.

- 做只能存储 1 位的电路

After that, we'll scale up, and build our very own memory module,

之后再扩大，做出我们的内存模块

and we'll combine it with our ALU next time, when we finally build our very own CPU!

下次和 ALU 结合起来，做出 CPU！

All of the logic circuits we've discussed so far go in one direction

我们至今说过的电路都是单向的

- always flowing forward.

- 总是向前流动

like our 8-bit ripple adder from last episode.

比如上集的 8 位 "脉动进位加法器"

But we can also create circuits that loop back on themselves.

但也可以做回向电路，把输出连回输入

Let's try taking an ordinary OR gate, and feed the output back into one of its inputs

我们拿一个 OR 门试试，把输出连回输入

and see what happens.

看看会发生什么

First, let's set both inputs to 0.

首先，两个输入都设为 0

So 0 OR 0 is 0, and so this circuit always outputs 0.

"0 OR 0" 是 0，所以电路输出0

If we were to flip input A to 1.

如果将 A 变成1

1 OR 0 is 1, so now the output of the OR gate is 1.

"1 OR 0" 为 1，所以输出 1

A fraction of a second later, that loops back around into input B,

一转眼的功夫，输出回到 B

so the OR gate sees that both of its inputs are now 1.

OR 门看到两个输入都是 1

1 OR 1 is still 1, so there is no change in output.

"1 OR 1" 仍然为1，所以输出不变

If we flip input A back to 0, the OR gate still outputs 1.

如果将 A 变成 0，OR 门依然输出 1

So now we've got a circuit that records a "1" for us.

现在我们有个电路能记录 "1"

Except, we've got a teensy tiny problem - this change is permanent!

然而有个小问题：这是永久的！

No matter how hard we try, there's no way to get this circuit to flip back from a 1 to a 0.

无论怎么试，都没法从 1 变回 0

Now let's look at this same circuit, but with an AND gate instead.

我们换成 AND 门看看会怎样

We'll start inputs A and B both at 1.

开始时，A 和 B 都设 1

1 AND 1 outputs 1 forever.

"1 AND 1" 永远输出 1

But, if we then flip input A to 0, because it's an AND gate, the output will go to 0.

如果之后 A 设为 0，由于是 AND 门，输出会变成 0

So this circuit records a 0, the opposite of our other circuit.

这个电路能记录 0，和之前那个相反

Like before, no matter what input we apply to input A afterwards, the circuit will always output 0.

就像之前，无论 A 设什么值，电路始终输出 0

Now we've got circuits that can record both 0s and 1s.

现在有了能存 0 和 1 的电路

The key to making this a useful piece of memory is to combine our two circuits into what is

为了做出有用的存储 (memory) \N 我们把两个电路结合起来

called the AND-OR Latch.

这叫 "AND-OR 锁存器"

It has two inputs, a "set" input, which sets the output to a 1,\Ncalled the AND-OR Latch.

It has two inputs, a "set" input, which sets the output to a 1,

它有两个输入\N "设置"输入, 把输出变成 1\N "复位"输入, 把输出变成 0

and a "reset" input, which resets the output to a 0.

它有两个输入\N "设置"输入, 把输出变成 1\N "复位"输入, 把输出变成 0

If set and reset are both 0, the circuit just outputs whatever was last put in it.

如果"设置"和"复位"都是 0，电路会输出最后放入的内容

In other words, it remembers a single bit of information!

也就是说，它存住了 1 位的信息！

Memory!

存储！

This is called a "latch" because it "latches onto" a particular value and stays that way.

这叫"锁存", 因为它"锁定"了一个值

The action of putting data into memory is called writing, whereas getting the data out is called reading.

放入数据的动作叫 "写入" ，拿出数据的动作叫 "读取"

Ok, so we've got a way to store a single bit of information!

现在我们终于有办法存一个位了！

Great!

超棒！

Unfortunately, having two different wires for input - set and reset - is a bit confusing.

麻烦的是, 用两条线 "设置"和"复位" 来输入, 有点难理解

To make this a little easier to use, we really want a single wire to input data,

为了更容易用，我们希望只有一条输入线

that we can set to either 0 or 1 to store the value.

将它设为 0 或 1 来存储值

Additionally, we are going to need a wire that enables the memory

还需要一根线来"启用"内存

to be either available for writing or "locked" down

启用时允许写入，没启用时就 "锁定"

- which is called the write enable line.

- 这条线叫 "允许写入线"

By adding a few extra logic gates, we can build this circuit,

加一些额外逻辑门，可以做出这个电路

which is called a Gated Latch since the "gate" can be opened or closed.

这叫"门锁"，因为门可以打开和关上

Now this circuit is starting to get a little complicated.

现在有点复杂了

We don't want to have to deal with all the individual logic gates...

我们不想关心单独的逻辑门

so as before, we're going to bump up a level of abstraction,

所以我们提升一层抽象

and put our whole Gated Latch circuit in a box -- a box that stores one bit.

把 "门锁" 放到盒子里 - 这个盒子能存一个 bit

Let's test out our new component!

我们来测一下新组件！

Let's start everything at 0.

一切从 0 开始

If we toggle the Data wire from 0 to 1 or 1 to 0,

数据输入从0换到1, 从1换到0

nothing happens - the output stays at 0.

什么也不会发生 - 输出依然是 0

That's because the write enable wire is off, which prevents any change to the memory.

因为 "允许写入线" 是关闭的，所以内容不会变化

So we need to "open" the "gate" by turning the write enable wire to 1.

所以要给 "允许写入线" 输入 1, "打开" 门

Now we can put a 1 on the data line to save the value 1 to our latch.

现在往 "数据线" 放 1，1 就能存起来了

Notice how the output is now 1.\NNow we can put a 1 on the data line to save the value 1 to our latch.

Notice how the output is now 1.

注意输出现在是 1 了

Success!

成功！

We can turn off the enable line and the output stays as 1.

现在可以关掉 "允许写入线" ，输出会保持 1

Once again, we can toggle the value on the data line all we want,

现在不管给 "数据线" 什么值

but the output will stay the same.

输出都不会变

The value is saved in memory.

值存起来了

Now let's turn the enable line on again use our data line to set the latch to 0.

现在又打开 "允许写入线" \N "数据线" 设为0

Done.

完成

Enable line off, and the output is 0.

"允许写入线" 关闭，输出 0

And it works!

成功了！

Now, of course, computer memory that only stores one bit of information isn't very useful

当然，只能存 1 bit 没什么大用

-- definitely not enough to run Frogger.

- 肯定玩不了游戏

Or anything, really.

或做其它事情

But we're not limited to using only one latch.

但我们没限制只能用一个锁存器

If we put 8 latches side-by-side, we can store 8 bits of information like an 8-bit number.

如果我们并排放 8 个锁存器，\N 可以存 8 位信息，比如一个 8 bit 数字

A group of latches operating like this is called a register,

一组这样的锁存器叫 "寄存器"

which holds a single number, and the number of bits in a register is called its width.

寄存器能存一个数字，这个数字有多少位，叫"位宽"

Early computers had 8-bit registers, then 16, 32,

早期电脑用 8 位寄存器，然后是 16 位，32 位

and today, many computers have registers that are 64-bits wide.

如今许多计算机都有 64 位宽的寄存器

To write to our register, we first have to enable all of the latches.

写入寄存器前，要先启用里面所有锁存器

We can do this with a single wire that connects to all of their enable inputs, which we set to 1.

我们可以用一根线连接所有 "允许输入线", 把它设为 1

We then send our data in using the 8 data wires, and then set enable back to 0,

然后用 8 条数据线发数据，然后将 "允许写入线" 设回 0

and the 8 bit value is now saved in memory.

现在 8 位的值就存起来了

Putting latches side-by-side works ok for a small-ish number of bits.

如果只有很少的位(bits)，把锁存器并排放置，也勉强够用了.

A 64-bit register would need 64 wires running to the data pins, and 64 wires running to the outputs.

64 位寄存器要 64 根数据线，64 根连到输出端

Luckily we only need 1 wire to enable all the latches, but that's still 129 wires.

幸运的是，我们只要 1 根线启用所有锁存器 \N 但加起来也有 129 条线了

For 256 bits, we end up with 513 wires!

如果存 256 位要 513 条线！

The solution is a matrix!

解决方法是矩阵！

In this matrix, we don't arrange our latches in a row,

在矩阵中，我们不并列排放锁存器

we put them in a grid.

而是做成网格

For 256 bits, we need a 16 by 16 grid of latches with 16 rows and columns of wires.

存 256 位，我们用 16x16 网格的锁存器，有 16 行 16 列

To activate any one latch, we must turn on the corresponding row AND column wire.

要启用某个锁存器，就打开相应的 行线 和 列线

Let's zoom in and see how this works.

放大看看怎么做的

We only want the latch at the intersection of the two active wires to be enabled,

我们只想打开交叉处 锁存器的 "允许写入线"

but all of the other latches should stay disabled.

所有其他锁存器，保持关闭

For this, we can use our trusty AND gate!

我们可以用 AND 门！

The AND gate will output a 1 only if the row and the column wires are both 1.

只有 行线和列线 均为 1 ，AND 门才输出 1

So we can use this signal to uniquely select a single latch.

所以可以用选择单个锁存器

This row/column setup connects all our latches with a single, shared, write enable wire.

这种行/列排列法，用一根 "允许写入线" 连所有锁存器

In order for a latch to become write enabled,

为了让锁存器变成 "允许写入"

the row wire, the column wire, and the write enable wire must all be 1.

行线，列线和 "允许写入线" 都必须是 1

That should only ever be true for one single latch at any given time.

每次只有 1 个锁存器会这样

This means we can use a single, shared wire for data.

代表我们可以只用一根 "数据线" \N 连所有锁存器来传数据

Because only one latch will ever be write enabled, only one will ever save the data

因为只有一个锁存器会启用，只有那个会存数据

-- the rest of the latches will simply ignore values on the data wire because they are not write enabled.

其他锁存器会忽略数据线上的值，因为没有 "允许写入"

We can use the same trick with a read enable wire to read the data later,

我们可以用类似的技巧, 做"允许读取线"来读数据

to get the data out of one specific latch.

从一个指定的锁存器，读取数据

This means in total, for 256 bits of memory,

所以对于 256 位的存储

we only need 35 wires - 1 data wire, 1 write enable wire, 1 read enable wire,

只要 35 条线 \N1条"数据线", 1条"允许写入线", 1条"允许读取线"

and 16 rows and columns for the selection.

还有16行16列的线用于选择锁存器 \N （16+16=32, 32+3=35）

That's significant wire savings!

这省了好多线！

But we need a way to uniquely specify each intersection.

但我们需要某种方法来 唯一指定 交叉路口

We can think of this like a city,

我们可以想成城市

where you might want to meet someone at 12th avenue and 8th street

你可能想和别人 在第 12 大道和第 8 街的交界碰面

-- that's an address that defines an intersection.

- 这是一个交叉点的地址

The latch we just saved our one bit into has an address of row 12 and column 8.

我们刚刚存了一位的地址是 "12行 8列"

Since there is a maximum of 16 rows, we store the row address in a 4 bit number.

由于最多 16 行, 用 4 位就够了

12 is 1100 in binary.

12 用二进制表示为 1100

We can do the same for the column address: 8 is 1000 in binary.

列地址也可以这样： 8 用二进制表示为 1000

So the address for the particular latch we just used can be written as 11001000.

刚才说的"12行 8列"可以写成 11001000

To convert from an address into something that selects the right row or column,

为了将地址转成 行和列

we need a special component called a multiplexer

我们需要 "多路复用器"

-- which is the computer component with a pretty cool name at least compared to the ALU.

- 这个名字起码比 ALU 酷一点

Multiplexers come in all different sizes,

多路复用器有不同大小

but because we have 16 rows, we need a 1 to 16 multiplexer.

因为有 16 行，我们需要 1 到 16 多路复用器

It works like this.

工作方式是

You feed it a 4 bit number, and it connects the input line to a corresponding output line.

输入一个 4 位数字，它会把那根线，连到相应的输出线

So if we pass in 0000, it will select the very first column for us.

如果输入 0000，它会选择第一列

If we pass in 0001, the next column is selected, and so on.

如果输入 0001，会选择下一列，依此类推

We need one multiplexer to handle our rows and another multiplexer to handle the columns.

一个多路复用器处理行(row) \N 另一个多路复用器处理列(column)

Ok, it's starting to get complicated again,

好吧，开始有点复杂了

so let's make our 256-bit memory its own component.

那么把 256 位内存当成一个整体好了

Once again a new level of abstraction!

又提升了一层抽象！

It takes an 8-bit address for input - the 4 bits for the column and 4 for the row.

它输入一个 8 位地址：4 位代表列，4 位代表行

We also need write and read enable wires.

我们还需要 "允许写入线" 和 "允许读取线"

And finally, we need just one data wire, which can be used to read or write data.

最后，还需要一条数据线，用于读/写数据

Unfortunately, even 256-bits of memory isn't enough to run much of anything,

不幸的是， 256 位的内存也没法做什么事

so we need to scale up even more!

所以还要扩大规模

We're going to put them in a row.

把它们并排放置

Just like with the registers.

就像寄存器一样

We'll make a row of 8 of them, so we can store an 8 bit number - also known as a byte.

一行8个，可以存一个 8 位数字 \N 8 位也叫一个字节（byte）

To do this, we feed the exact same address into all 8 of our 256-bit memory components at the same time,

为了存一个 8 位数字，我们同时给 8 个 256 位内存一样的地址

and each one saves one bit of the number.

每个地址存 1 位

That means the component we just made can store 256 bytes at 256 different addresses.

意味着这里总共能存 256 个字节 （byte）

Again, to keep things simple, we want to leave behind this inner complexity.

再次，为了简单，我们不管内部

Instead of thinking of this as a series of individual memory modules and circuits,

不看作是一堆独立的存储模块和电路

we'll think of it as a uniform bank of addressable memory.

而是看成一个整体的可寻址内存

We have 256 addresses,

我们有 256 个地址

and at each address, we can read or write an 8-bit value.

每个地址能读或写一个 8 位值

We're going to use this memory component next episode when we build our CPU.

我们下集做 CPU 时会用到这个内存

The way that modern computers scale to megabytes and gigabytes of memory

现代计算机的内存 \N 扩展到上兆字节（MB）和千兆字节（GB）的方式

is by doing the same thing we've been doing here

和我们这里做的一样

-- keep packaging up little bundles of memory into larger, and larger, and larger arrangements.

不断把内存打包到更大规模

As the number of memory locations grow, our addresses have to grow as well.

随着内存地址增多，内存地址也必须增长

8 bits hold enough numbers to provide addresses for 256 bytes of our memory,

8 位最多能代表 256 个内存地址 \N（1111 1111 是255，0~255 一共 256 个数字）

but that's all.

只有这么多

To address a gigabyte - or a billion bytes of memory - we need 32-bit addresses.

要给千兆或十亿字节的内存寻址，需要 32 位的地址

An important property of this memory is that we can access any memory location, at any time, and in a random order.

内存的一个重要特性是：可以随时访问任何位置

For this reason, it's called Random-Access Memory or RAM.

因此叫 "随机存取存储器" ，简称 RAM

When you hear people talking about how much RAM a computer has

当你听到有人说 RAM 有多大

- that's the computer's memory.

他的意思是内存有多大

RAM is like a human's short term or working memory,

RAM 就像人类的短期记忆

where you keep track of things going on right now

记录当前在做什么事

- like whether or not you had lunch or paid your phone bill.

比如吃了午饭没，或有没有交电话费

Here's an actual stick of RAM - with 8 memory modules soldered onto the board.

这是一条真的内存，上面焊了 8 个内存模块

If we carefully opened up one of these modules and zoomed in,

如果打开其中一个，然后放大

The first thing you would see are 32 squares of memory.

会看到 32 个内存方块

Zoom into one of those squares, and we can see each one is comprised of 4 smaller blocks.

放大其中一个方块，可以看到有 4 个小块

If we zoom in again, we get down to the matrix of individual bits.

如果再放大，可以看到存一个"位"的矩阵

This is a matrix of 128 by 64 bits.

这个矩阵是 128 位 x 64 位

That's 8192 bits in total.

总共 8192 位

Each of our 32 squares has 4 matrices, so that's 32 thousand, 7 hundred and 68 bits.

每个方格 4 个矩阵 \N 所以一个方格有 32768 个位 （8192 x 4 = 32768）

And there are 32 squares in total.

而一共 32 个方格

So all in all, that's roughly 1 million bits of memory in each chip.

总而言之，1 个芯片大约存 100 万位

Our RAM stick has 8 of these chips, so in total, this RAM can store 8 millions bits,

RAM 有 8 个芯片，所以总共 800 万位

otherwise known as 1 megabyte.

也就是 1 兆字节（1 MB）

That's not a lot of memory these days -- this is a RAM module from the 1980's.

1 MB 如今不算大 - 这是 1980 年代的 RAM

Today you can buy RAM that has a gigabyte or more of memory

如今你可以买到千兆字节（GB）的 RAM

- that's billions of bytes of memory.

那可是数十亿字节的内存

So, today, we built a piece of SRAM - Static Random-Access Memory - which uses latches.

今天，我们用锁存器做了一块 SRAM（静态随机存取存储器）

There are other types of RAM, such as DRAM, Flash memory, and NVRAM.

还有其他类型的 RAM，如 DRAM，闪存和 NVRAM

These are very similar in function to SRAM,

它们在功能上与 SRAM 相似

but use different circuits to store the individual bits

但用不同的电路存单个位

- for example, using different logic gates, capacitors, charge traps, or memristors.

- 比如用不同的逻辑门，电容器，电荷捕获或忆阻器

But fundamentally, all of these technologies store bits of information

但根本上 这些技术都是矩阵层层嵌套，来存储大量信息

in massively nested matrices of memory cells.

但根本上 这些技术都是矩阵层层嵌套，来存储大量信息

Like many things in computing, the fundamental operation is relatively simple.

就像计算机中的很多事情，底层其实都很简单

it's the layers and layers of abstraction that's mind blowing

让人难以理解的，是一层层精妙的抽象

-- like a russian doll that keeps getting smaller and smaller and smaller.

像一个越来越小的俄罗斯套娃

I'll see you next week.

下周见

Hi, I'm Carrie Anne, this is Crash Course Computer Science

嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

and today, we're talking about processors.

今天我们讲 处理器

Just a warning though - this is probably the most complicated episode in the series.

提示下 - 这集可能是最难的一集

So once you get this, you're golden.

所以一旦你理解了，就会变得超厉害der~

We've already made a Arithmetic and Logic Unit,

我们已经做了一个算术逻辑单元（ALU）

which takes in binary numbers and performs calculations,

输入二进制，它会执行计算

and we've made two types of computer memory:

我们还做了两种内存：

Registers -- small, linear chunks of memory, useful for storing a single value

寄存器 - 很小的一块内存，能存一个值

and then we scaled up, and made some RAM,

之后我们增大做出了 RAM

a larger bank of memory that can store a lot of numbers located at different addresses.

RAM 是一大块内存，能在不同地址存大量数字

Now it's time to put it all together and build ourselves the heart of any computer,

现在是时候把这些放在一起，组建计算机的 "心脏" 了

but without any of the emotional baggage that comes with human hearts.

但这个 "心脏" 不会有任何包袱，比如人类情感.

For computers, this is the Central Processing Unit, most commonly called the CPU.

计算机的心脏是"中央处理单元"，简称 "CPU"

A CPU's job is to execute programs.

CPU 负责执行程序

Programs, like Microsoft Office, Safari, or your beloved copy of Half Life: 2,

比如 Office，Safari 浏览器，你最爱的 《半条命2》

are made up of a series of individual operations,

程序由一个个操作组成

called instructions, because they "instruct" the computer what to do.

这些操作叫"指令"(Instruction) \N 因为它们"指示"计算机要做什么

If these are mathematical instructions, like add or subtract,

如果是数学指令，比如加/减

the CPU will configure its ALU to do the mathematical operation.

CPU 会让 ALU 进行数学运算

Or it might be a memory instruction,

也可能是内存指令，CPU 会和内存通信，然后读/写值

in which case the CPU will talk with memory to read and write values.

也可能是内存指令，CPU 会和内存通信，然后读/写值

There are a lot of parts in a CPU,

CPU 里有很多组件.

so we're going to lay it out piece by piece, building up as we go.

所以我们一边说一边建

We'll focus on functional blocks, rather than showing every single wire.

我们把重点放在功能，而不是一根根线具体怎么连

When we do connect two components with a line,

当我们用一条线连接两个组件时

this is an abstraction for all of the necessary wires.

这条线只是所有必须线路的一个抽象

This high level view is called the microarchitecture.

这种高层次视角叫 "微体系架构"

OK, first, we're going to need some memory.

好，我们首先要一些内存，把上集做的 RAM 拿来就行

Lets drop in the RAM module we created last episode.

好，我们首先要一些内存，把上集做的 RAM 拿来就行

To keep things simple, we'll assume it only has 16 memory locations, each containing 8 bits.

为了保持简单，假设它只有 16 个位置，每个位置存 8 位

Let's also give our processor four, 8-bit memory registers, labeled A, B, C and D

再来四个 8 位寄存器，叫 A，B，C，D

which will be used to temporarily store and manipulate values.

寄存器用来 临时存数据 和 操作数据

We already know that data can be stored in memory as binary values

我们已经知道数据 是以二进制值存在内存里

and programs can be stored in memory too.

程序也可以存在内存里

We can assign an ID to each instruction supported by our CPU.

我们可以给 CPU 支持的所有指令，分配一个 ID

指令表

指令

描述

4位操作码

地址或寄存器

In our hypothetical example, we use the first four bits to store the "operation code",

在这个假设的例子 \N 我们用前四位存 "操作代码" （operation code）

or opcode for short.

简称 "操作码" （opcode）

The final four bits specify where the data for that operation should come from -

后四位代表数据来自哪里

this could be registers or an address in memory.

- 可以是寄存器或内存地址

We also need two more registers to complete our CPU.

我们还需要两个寄存器，来完成 CPU.

First, we need a register to keep track of where we are in a program.

1. 一个寄存器追踪程序运行到哪里了，我们叫它 "指令地址寄存器"

For this, we use an instruction address register,

1. 一个寄存器追踪程序运行到哪里了，我们叫它 "指令地址寄存器"

which as the name suggests, stores the memory address of the current instruction.

顾名思义，存当前指令的内存地址

And then we need the other register to store the current instruction,

2. 另一个寄存器存当前指令，叫 "指令寄存器"

which we'll call the instruction register.

2. 另一个寄存器存当前指令，叫 "指令寄存器"

When we first boot up our computer, all of our registers start at 0.

当启动计算机时，所有寄存器从 0 开始

As an example, we've initialized our RAM with a simple computer program that we'll to through today.

为了举例，我们在 RAM 里放了一个程序，我们今天会过一遍

The first phase of a CPU's operation is called the fetch phase.

CPU 的第一个阶段叫 "取指令阶段"

This is where we retrieve our first instruction.

负责拿到指令

First, we wire our Instruction Address Register to our RAM module.

首先，将 "指令地址寄存器" 连到 RAM

The register's value is 0, so the RAM returns whatever value is stored in address 0.

寄存器的值为 0，因此 RAM 返回地址 0 的值

In this case, 0010 1110.

0010 1110 会复制到 "指令寄存器" 里

Then this value is copied into our instruction register.

0010 1110 会复制到 "指令寄存器" 里

Now that we've fetched an instruction from memory,

现在指令拿到了

we need to figure out what that instruction is

要弄清是什么指令，才能执行（execute）

so we can execute it.

要弄清是什么指令，才能执行（execute）

That is run it.

而不是杀死（kill）它

Not kill it.

而不是杀死（kill）它

This is called the decode phase.

这是 "解码阶段"

指令表

指令

描述

4位操作码

地址或寄存器

In this case the opcode, which is the first four bits, is: 0010.

前 4 位 0010 是 LOAD A 指令

This opcode corresponds to the "LOAD A" instruction,

前 4 位 0010 是 LOAD A 指令

which loads a value from RAM into Register A.

意思是，把 RAM 的值放入寄存器 A

The RAM address is the last four bits of our instruction which are 1110, or 14 in decimal.

后 4 位 1110 是 RAM 的地址, 转成十进制是 14

Next, instructions are decoded and interpreted by a Control Unit.

接下来，指令由 "控制单元" 进行解码

Like everything else we've built, it too is made out of logic gates.

就像之前的所有东西 \N "控制单元" 也是逻辑门组成的

For example, to recognize a LOAD A instruction,

比如，为了识别 "LOAD A" 指令

we need a circuit that checks if the opcode matches 0010

需要一个电路，检查操作码是不是 0010

which we can do with a handful of logic gates.

我们可以用很少的逻辑门来实现.

Now that we know what instruction we're dealing with,

现在知道了是什么指令

we can go ahead and perform that instruction which is the beginning of the execute phase!

就可以开始执行了，开始 "执行阶段"

Using the output of our LOAD\_A checking circuit,

用 "检查是否 LOAD\_A 指令的电路"

we can turn on the RAM's read enable line and send in address 14.

可以打开 RAM 的 "允许读取线", 把地址 14 传过去

The RAM retrieves the value at that address, which is 00000011, or 3 in decimal.

RAM 拿到值，0000 0011，十进制的 3

Now, because this is a LOAD\_A instruction,

因为是 LOAD\_A 指令 \N 我们想把这个值只放到寄存器 A，其他寄存器不受影响

we want that value to only be saved into Register A and not any of the other registers.

因为是 LOAD\_A 指令 \N 我们想把这个值只放到寄存器 A，其他寄存器不受影响

So if we connect the RAM's data wires to our four data registers,

所以需要一根线，把 RAM 连到 4 个寄存器

we can use our LOAD\_A check circuit to enable the write enable only for Register A.

用 "检查是否 LOAD\_A 指令的电路" \N 启用寄存器 A 的 "允许写入线"

And there you have it

这就成功了

-- we've successfully loaded the value at RAM address 14 into Register A.

- 把 RAM 地址 14 的值，放到了寄存器 A.

We've completed the instruction, so we can turn all of our wires off,

既然指令完成了，我们可以关掉所有线路

and we are ready to fetch the next instruction in memory.

去拿下一条指令

To do this, we increment the Instruction Address Register by 1 which completes the execute phase.

我们把 "指令地址寄存器"+1，"执行阶段"就此结束.

LOAD\_A is just one of several possible instructions that our CPU can execute.

LOAD\_A 只是 CPU 可以执行的各种指令之一

Different instructions are decoded by different logic circuits,

不同指令由不同逻辑电路解码

which configure the CPU's components to perform that action.

这些逻辑电路会配置 CPU 内的组件来执行对应操作

Looking at all those individual decode circuits is too much detail,

具体分析这些解码电路太繁琐了

so since we looked at one example,

既然已经看了 1 个例子，

we're going to go head and package them all up as a single Control Unit to keep things simple.

干脆把 "控制单元 "包成一个整体，简洁一些.

That's right a new level of abstraction.

没错，一层新抽象

The Control Unit is comparable to the conductor of an orchestra,

控制单元就像管弦乐队的指挥

directing all of the different parts of the CPU.

"指挥" CPU 的所有组件

Having completed one full fetch/decode/execute cycle,

"取指令→解码→执行" 完成后

we're ready to start all over again, beginning with the fetch phase.

现在可以再来一次，从 "取指令" 开始

The Instruction Address Register now has the value 1 in it,

"指令地址寄存器" 现在的值是 1

so the RAM gives us the value stored at address 1, which is 0001 1111.

所以 RAM 返回地址 1 里的值：0001 1111

On to the decode phase!

到 "解码" 阶段！

0001 is the "LOAD B" instruction, which moves a value from RAM into Register B.

0001 是 LOAD B 指令 \N 从 RAM 里把一个值复制到寄存器 B

The memory location this time is 1111, which is 15 in decimal.

这次内存地址是 1111，十进制的 15

Now to the execute phase!

现在到 "执行阶段"！

The Control Unit configures the RAM to read address 15 and configures Register B to receive the data.

"控制单元" 叫 RAM 读地址 15，并配置寄存器 B 接收数据

Bingo, we just saved the value 00001110, or the number 14 in decimal, into Register B.

成功，我们把值 0000 1110 \N 也就是十进制的 14 存到了寄存器 B

Last thing to do is increment our instruction address register by 1,

最后一件事是 "指令地址寄存器" +1

and we're done with another cycle.

我们又完成了一个循环

Our next instruction is a bit different.

下一条指令有点不同

Let's fetch it.

来取它吧

1000 0100.

1000 0100

That opcode 1000 is an ADD instruction.

1000 是 ADD 指令

Instead of an 4-bit RAM address, this instruction uses two sets of 2 bits.

这次后面的 4 位不是 RAM 地址，\N 而是 2 位 2 位分别代表 2 个寄存器

Remember that 2 bits can encode 4 values,

2 位可以表示 4 个值

so 2 bits is enough to select any one of our 4 registers.

所以足够表示 4 个寄存器

The first set of 2 bits is 01, which in this case corresponds to Register B,

第一个地址是 01, 代表寄存器B

and 00, which is Register A.

第二个地址是 00, 代表寄存器A

So "1000 01 00" is the instruction for adding the value in Register B into the value in register A.

因此，1000 0100，代表把寄存器 B 的值，加到寄存器 A 里

So to execute this instruction, we need to integrate the ALU we made in Episode 5 into our CPU.

为了执行这个指令，我们要整合第 5 集的 ALU

The Control Unit is responsible for selecting the right registers to pass in as inputs,

"控制单元" 负责选择正确的寄存器作为输入

and configuring the ALU to perform the right operation.

并配置 ALU 执行正确的操作

For this ADD instruction, the Control Unit enables Register B

对于 "ADD" 指令， "控制单元" 会

and feeds its value into the first input of the ALU.

启用寄存器 B，作为 ALU 的第一个输入

It also enables Register A and feeds it into the second ALU input.

还启用寄存器 A，作为 ALU 的第二个输入

As we already discussed, the ALU itself can perform several different operations,

之前说过，ALU 可以执行不同操作

so the Control Unit must configure it to perform an ADD operation by passing in the ADD opcode.

所以控制单元必须传递 ADD 操作码告诉它要做什么

Finally, the output should be saved into Register A.

最后，结果应该存到寄存器 A

But it can't be written directly

但不能直接写入寄存器 A

because the new value would ripple back into the ALU and then keep adding to itself.

这样新值会进入 ALU ，不断和自己相加

So the Control Unit uses an internal register to temporarily save the output,

因此，控制单元用一个自己的寄存器暂时保存结果

turn off the ALU, and then write the value into the proper destination register.

关闭 ALU，然后把值写入正确的寄存器

In this case, our inputs were 3 and 14, and so the sum is 17, or 00010001 in binary,

这里 3+14=17，二进制是 0001 0001

which is now sitting in Register A.

现在存到了寄存器 A

As before, the last thing to do is increment our instruction address by 1,

和之前一样，最后一件事是把指令地址 + 1

and another cycle is complete.

这个循环就完成了

Okay, so let's fetch one last instruction: 0100 1101.

好，来看最后一个指令：0100 1101

When we decode it we see that 0100 is a STORE\_A instruction, with a RAM address of 13.

解码得知是 STORE A 指令（把寄存器 A 的值放入内存） \N RAM 地址 13

As usual, we pass the address to the RAM module,

接下来，把地址传给 RAM

but instead of read-enabling the memory, we write-enable it.

但这次不是 "允许读取" ，而是 "允许写入"

At the same time, we read-enable Register A.

同时，打开寄存器 A 的 "允许读取"

This allows us to use the data line to pass in the value stored in register A.

这样就可以把寄存器 A 里的值，传给 RAM

Congrats, we just ran our first computer program!

恭喜，我们刚运行了第一个电脑程序！

It loaded two values from memory, added them together,

它从内存中加载两个值，相加，然后把结果放回内存

and then saved that sum back into memory.

它从内存中加载两个值，相加，然后把结果放回内存

Of course, by me talking you through the individual steps,

刚刚是我一步步来讲的,

I was manually transitioning the CPU through its fetch, decode and execute phases.

我们人工切换 CPU 的状态 "取指令→解码→执行"

But there isn't a mini Carrie Anne inside of every computer.

但不是每台电脑里都有一个迷你 Carrie Anne

So the responsibility of keeping the CPU ticking along falls to a component called the clock.

其实是 "时钟" 来负责管理 CPU 的节奏

As it's name suggests, the clock triggers an electrical signal at a precise and regular interval.

时钟以精确的间隔 触发电信号

Its signal is used by the Control Unit to advance the internal operation of the CPU,

控制单元会用这个信号，推进 CPU 的内部操作

keeping everything in lock-step

确保一切按步骤进行

- like the dude on a Roman galley drumming rhythmically at the front,

- 就像罗马帆船的船头，有一个人负责按节奏的击鼓,

keeping all the rowers synchronized... or a metronome.

让所有划船的人同步... 就像节拍器一样

Of course you can't go too fast,

节奏不能太快

because even electricity takes some time to travel down wires and for the signal to settle.

因为就算是电也要一定时间来传输

The speed at which a CPU can carry out each step of the fetch-decode-execute cycle

CPU "取指令→解码→执行" 的速度叫 "时钟速度"

is called its Clock Speed.

CPU "取指令→解码→执行" 的速度叫 "时钟速度"

This speed is measured in Hertz - a unit of frequency.

单位是赫兹 - 赫兹是用来表示频率的单位

One Hertz means one cycle per second.

1 赫兹代表一秒 1 个周期

Given that it took me about 6 minutes to talk you through 4 instructions

因为我花了大概 6 分钟，给你讲了 4 条指令

LOAD, LOAD, ADD and STORE

读取→读取→相加→存储

that means I have an effective clock speed of roughly .03 Hertz.

所以我的时钟速度大概是 0.03 赫兹

Admittedly, I'm not a great computer

我承认我算数不快

but even someone handy with math might only be able to do one calculation in their head every second or 1 Hertz.

但哪怕有人算数很快，最多也就是一秒一次，或 1 赫兹

The very first, single-chip CPU was the Intel 4004, a 4-bit CPU released in 1971.

第一个单芯片 CPU 是 "英特尔 4004" \N 1971 年发布的 4 位CPU

It's microarchitecture is actually pretty similar to our example CPU.

它的微架构 很像我们之前说的 CPU

Despite being the first processor of its kind,

虽然是第一个单芯片的处理器

it had a mind-blowing clock speed of 740 Kilohertz

但它的时钟速度达到了 740 千赫兹 - 每秒 74 万次

-- that's 740 thousand cycles per second.

但它的时钟速度达到了 740 千赫兹 - 每秒 74 万次

You might think that's fast,

你可能觉得很快

but it's nothing compared to the processors that we use today.

但和如今的处理器相比不值一提

One megahertz is one million clock cycles per second,

一兆赫兹是 1 秒 1 百万个时钟周期

and the computer or even phone that you are watching this video on right now is no doubt a few gigahertz

你现在看视频的电脑或手机，肯定有几千兆赫兹

-- that's BILLIONs of CPU cycles every single... second.

- 1 秒 10 亿次时钟周期

Also, you may have heard of people overclocking their computers.

你可能听过有人会把计算机超频

This is when you modify the clock to speed up the tempo of the CPU

意思是修改时钟速度，加快 CPU 的速度

-- like when the drummer speeds up when the Roman Galley needs to ram another ship.

- 就像罗马帆船要撞另一艘船时，鼓手会加快敲鼓速度

Chip makers often design CPUs with enough tolerance to handle a little bit of overclocking,

芯片制造商经常给 CPU 留一点余地，可以接受一点超频

but too much can either overheat the CPU,

但超频太多会让 CPU 过热

or produce gobbledygook as the signals fall behind the clock.

或产生乱码，因为信号跟不上时钟

And although you don't hear very much about underclocking,

你可能很少听说降频

it's actually super useful.

但降频其实很有用

Sometimes it's not necessary to run the processor at full speed...

有时没必要让处理器全速运行

maybe the user has stepped away, or just not running a particularly demanding program.

可能用户走开了，或者在跑一个性能要求较低的程序

By slowing the CPU down, you can save a lot of power,

把 CPU 的速度降下来，可以省很多电

which is important for computers that run on batteries, like laptops and smartphones.

省电对用电池的设备很重要，比如笔记本和手机

To meet these needs,

为了尽可能省电

many modern processors can increase or decrease their clock speed based on demand,

很多现代处理器可以按需求 加快或减慢时钟速度

which is called dynamic frequency scaling.

这叫 "动态调整频率"

So, with the addition of a clock, our CPU is complete.

加上时钟后，CPU 才是完整的.

We can now put a box around it, and make it its own component.

现在可以放到盒子里，变成一个独立组件

Yup.

对

A new level of abstraction!

一层新的抽象！

RAM, as I showed you last episode,

RAM，上集说过，是在 CPU 外面的独立组件

lies outside the CPU as its own component,

RAM，上集说过，是在 CPU 外面的独立组件

and they communicate with each other using address, data and enable wires.

CPU 和 RAM 之间 \N 用 "地址线" "数据线" 和 "允许读/写线" 进行通信

Although the CPU we designed today is a simplified example,

虽然今天我们设计的 CPU 是简化版的,

many of the basic mechanics we discussed are still found in modern processors.

但我们提到的很多机制，依然存在于现代处理器里

Next episode, we're going to beef up our CPU,

下一集，我们要加强 CPU，给它扩展更多指令.

extending it with more instructions as we take our first baby steps into software.

同时开始讲软件.

I'll see you next week.

下周见

Hi, I’m Carrie Anne and this is Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Last episode, we combined an ALU, control unit, some memory, and a clock together to

上集我们把 ALU, 控制单元, RAM, 时钟 结合在一起

make a basic, but functional Central Processing Unit – or CPU

做了个基本，但可用的"中央处理单元", 简称 CPU

– the beating, ticking heart of a computer.

它是计算机的核心

We’ve done all the hard work of building many of these components from the electronic

我们已经用电路做了很多组件.

circuits up, and now it’s time to give our CPU some actual instructions to process!

这次我们给 CPU 一些指令来运行!

The thing that makes a CPU powerful is the fact that it is programmable –

CPU 之所以强大，是因为它是可编程的 -

if you write a different sequence of instructions, then the CPU will perform a different task.

如果写入不同指令，就会执行不同任务

So the CPU is a piece of hardware which is controlled by easy-to-modify software!

CPU 是一块硬件，可以被软件控制!

Let’s quickly revisit the simple program that we stepped through last episode.

我们重新看一下上集的简单程序

The computer memory looked like this.

内存里有这些值

Each address contained 8 bits of data.

每个地址可以存 8 位数据

For our hypothetical CPU, the first four bits specified the operation code, or opcode, and

因为我们的 CPU 是假设的，这里前4位是"操作码"

the second set of four bits specified an address or registers.

后4位指定一个内存地址，或寄存器.

In memory address zero we have 0010 1110.

内存地址 0 是 0010 1110

Again, those first four bits are our opcode which corresponds to a "LOAD\_A" instruction.

前 4 位代表 LOAD\_A 指令

This instruction reads data from a location of memory specified in those last four bits

意思是：把后 4 位指定的内存地址的值，放入寄存器 A

of the instruction and saves it into Register A. In this case, 1110, or 14 in decimal.

后 4 位是 1110，十进制的 14

So let’s not think of this of memory address 0 as "0010 1110", but rather as the instruction

我们来把 0010 1110 看成 "LOAD\_A 14" 指令

"LOAD\_A 14".

我们来把 0010 1110 看成 "LOAD\_A 14" 指令

That’s much easier to read and understand!

这样更好理解！

And for me to say!

也更方便说清楚

And we can do the same thing for the rest of the data in memory.

可以对内存里剩下的数也这样转换.

In this case, our program is just four instructions long,

这里，我们的程序只有4个指令

and we’ve put some numbers into memory too, 3 and 14.

还有数字 3 和 14

So now let’s step through this program:

现在一步步看

First is LOAD\_A 14, which takes the value in address 14, which is the number 3,

"LOAD\_A 14" 是从地址 14 中拿到数字3，放入寄存器A

and stores it into Register A.

"LOAD\_A 14" 是从地址 14 中拿到数字3，放入寄存器A

Then we have a "LOAD\_B 15" instruction, which takes the value in memory location 15,

"LOAD\_B 15" 是从地址 15 中拿到数字14，放入寄存器B

which is the number 14, and saves it into Register B.

"LOAD\_B 15" 是从地址 15 中拿到数字14，放入寄存器B

Okay.

好.

Easy enough.

挺简单的！

But now we have an "ADD" instruction.

下一个是 ADD 指令

This tells the processor to use the ALU to add two registers together,

"ADD B A" 告诉 ALU \N 把寄存器 B 和寄存器 A 里的数字加起来

in this case, B and A are specified.

"ADD B A" 告诉 ALU \N 把寄存器 B 和寄存器 A 里的数字加起来

The ordering is important, because the resulting sum is saved into the second register that’s specified.

（B和A的）顺序很重要，因为结果会存在第二个寄存器

So in this case, the resulting sum is saved into Register A.

也就是寄存器 A

And finally, our last instruction is "STORE\_A 13", which instructs the CPU to write whatever

最后一条指令是 "STORE\_A 13" \N 把寄存器 A 的值存入内存地址 13

value is in Register A into memory location 13.

最后一条指令是 "STORE\_A 13" \N 把寄存器 A 的值存入内存地址 13

Yesss!

好棒！

Our program adds two numbers together.

我们把 2 个数加在了一起!

That’s about as exciting as it gets when we only have four instructions to play with.

毕竟只有4个指令，也只能做这个了.

So let’s add some more!

加多一些指令吧!

Now we’ve got a subtract function, which like ADD, specifies two registers to operate on.

SUB 是减法，和 ADD 一样也要 2 个寄存器来操作.

We’ve also got a fancy new instruction called JUMP.

还有 JUMP（跳转）

As the name implies, this causes the program to "jump" to a new location.

让程序跳转到新位置

This is useful if we want to change the order of instructions, or choose to skip some instructions.

如果想改变指令顺序，或跳过一些指令，这个很实用

For example, a JUMP 0, would cause the program to go back to the beginning.

举例, JUMP 0 可以跳回开头

At a low level, this is done by writing the value specified in the last four bits into

JUMP 在底层的实现方式是 \N 把指令后 4 位代表的内存地址的值

the instruction address register, overwriting the current value.

覆盖掉 "指令地址寄存器" 里的值

We’ve also added a special version of JUMP called JUMP\_NEGATIVE.

还有一个特别版的 JUMP 叫 JUMP\_NEGATIVE

"This only jumps the program if the ALU’s negative flag is set to true.

它只在 ALU 的 "负数标志" 为真时，进行 JUMP

As we talked about in Episode 5, the negative flag is only set

第5集讲过，算数结果为负，"负数标志"才是真

when the result of an arithmetic operation is negative.

第5集讲过，算数结果为负，"负数标志"才是真

If the result of the arithmetic was zero or positive, the negative flag would not be set.

结果不是负数时, "负数标志"为假

So the JUMP NEGATIVE won’t jump anywhere, and the CPU will just continue on to the next instruction.

如果是假，JUMP\_NEGATIVE 就不会执行 \N 程序照常进行

Our previous program really should have looked like this to be correct,

我们之前的例子程序，其实应该是这样，才能正确工作.

otherwise the CPU would have just continued on after the STORE instruction, processing all those 0’s.

否则跑完 STORE\_A 13 之后，\N CPU 会不停运行下去，处理后面的 0

But there is no instruction with an opcode of 0, and so the computer would have crashed!

因为 0 不是操作码，所以电脑会崩掉!

It’s important to point out here that we’re storing

我还想指出一点，指令和数据都是存在同一个内存里的.

both instructions and data in the same memory.

我还想指出一点，指令和数据都是存在同一个内存里的.

There is no difference fundamentally -- it’s all just binary numbers.

它们在根本层面上毫无区别 - 都是二进制数

So the HALT instruction is really important because it allows us to separate the two.

HALT 很重要，能区分指令和数据

Okay, so let’s make our program a bit more interesting, by adding a JUMP.

好，现在用 JUMP 让程序更有趣一些.

We’ll also modify our two starting values in memory to 1 and 1.

我们还把内存中 3 和 14 两个数字，改成 1 和 1

Lets step through this program just as our CPU would.

现在来从 CPU 的视角走一遍程序

First, LOAD\_A 14 loads the value 1 into Register A.

首先 LOAD\_A 14，把 1 存入寄存器A \N（因为地址 14 里的值是 1）

Next, LOAD\_B 15 loads the value 1 into Register B.

然后 LOAD\_B 15，把 1 存入寄存器B\N（因为地址 15 里的值也是 1）

As before, we ADD registers B and A together, with the sum going into Register A. 1+1 = 2,

然后 ADD B A 把寄存器 B 和 A 相加 \N 结果放到寄存器 A 里

so now Register A has the value 2 in it (stored in binary of course)

现在寄存器 A 的值是 2 \N (当然是以二进制存的）

Then the STORE instruction saves that into memory location 13.

然后 STORE\_A 13 指令，把寄存器 A 的值存入内存地址 13

Now we hit a "JUMP 2" instruction.

现在遇到 JUMP 2 指令

This causes the processor to overwrite the value in the instruction address register,

CPU 会把"指令地址寄存器"的值，现在是 4，改成 2

which is currently 4, with the new value, 2.

CPU 会把"指令地址寄存器"的值，现在是 4，改成 2

Now, on the processor’s next fetch cycle, we don’t fetch HALT,

因此下一步不再是 HALT

instead we fetch the instruction at memory location 2, which is ADD B A.

而是读内存地址 2 里的指令，也就是 ADD B A

We’ve jumped!

我们跳转了!

Register A contains the value 2, and register B contains the value 1.

寄存器 A 里是 2，寄存器 B 里是 1

So 1+2 = 3, so now Register A has the value 3.

1+2=3，寄存器 A 变成 3

We store that into memory.

存入内存

And we’ve hit the JUMP again, back to ADD B A.

又碰到 JUMP 2，又回到 ADD B A.

1+3=4

1+3=4

So now register A has the value 4.

现在寄存器 A 是 4

See what's happening here?

发现了吗？

Every loop, we’re adding one.

每次循环都+1

Its counting up!

不断增多

Cooooool.

酷

But notice there’s no way to ever escape.

但没法结束啊

We’re never.. ever.. going to get to that halt instruction,

永远不会碰到 HALT

because we’re always going to hit that JUMP.

总是会碰到 JUMP

This is called an infinite loop – a program that runs forever… ever… ever… ever…

这叫无限循环 - 这个程序会永远跑下去.. 下去.. 下去.. 下去

ever

下去

To break the loop, we need a conditional jump.

为了停下来，我们需要有条件的 JUMP

A jump that only happens if a certain condition is met.

只有特定条件满足了，才执行 JUMP.

Our JUMP\_NEGATIVE is one example of a conditional jump,

比如 JUMP NEGATIVE 就是条件跳转的一个例子

but computers have other types too - like JUMP IF EQUAL and JUMP IF GREATER.

还有其他类型的条件跳转，比如\N JUMP IF EQUAL（如果相等）\N JUMP IF GREATER（如果更大）

So let’s make our code a little fancier and step through it.

现在把代码弄花哨一点，再过一遍代码

Like before, the program starts by loading values from memory into registers A and B.

就像之前，程序先把内存值放入寄存器 A 和 B.

In this example, the number 11 gets loaded into Register A, and 5 gets loaded into Register B.

寄存器 A 是 11，寄存器 B 是 5

Now we subtract register B from register A. That’s 11 minus 5, which is 6,

SUB B A，用 A 减 B，11-5=6

and so 6 gets saved into Register A.

6 存入寄存器 A

Now we hit our JUMP NEGATIVE.

JUMP NEGATIVE 出场

The last ALU result was 6.

上一次 ALU 运算的结果是 6

That’s a positive number, so the the negative flag is false.

是正数，所以 "负数标志" 是假

That means the processor does not jump.

因此处理器不会执行 JUMP

So we continue on to the next instruction...

继续下一条指令

...which is a JUMP 2.

JUMP 2

No conditional on this one, so we jump to instruction 2 no matter what.

JUMP 2 没有条件，直接执行！

Ok, so we’re back at our SUBTRACT Register B from Register A. 6 minus 5 equals 1.

又回到寄存器 A-B，6-5=1

So 1 gets saved into register A.

A 变成 1

Next instruction.

下一条指令

We’re back again at our JUMP NEGATIVE.

又是 JUMP NEGATIVE

1 is also a positive number, so the CPU continues on to the JUMP 2, looping back around again

因为 1 还是正数，因此 JUMP NEGATIVE 不会执行 \N 来到下一条指令，JUMP 2

to the SUBTRACT instruction.

又来减一次

This time is different though.

这次就不一样了

1 minus 5 is negative 4.

1-5=-4

And so the ALU sets its negative flag to true for the first time.

这次ALU的 "负数标志" 是真

Now, when we advance to the next instruction,

现在下一条指令

JUMP\_NEGATIVE 5, the CPU executes the jump to memory location 5.

JUMP NEGATIVE 5, CPU 的执行跳到内存地址 5

We’re out of the infinite loop!

跳出了无限循环！

Now we have a ADD B to A. Negative 4 plus 5, is positive 1, and we save that into Register A.

现在的指令是 ADD B A，-4+5=1，1 存入寄存器 A

Next we have a STORE instruction that saves Register A into memory address 13.

下一条指令 STORE\_A 13，把 A 的值存入内存地址 13

Lastly, we hit our HALT instruction and the computer rests.

最后碰到 HALT 指令，停下来.

So even though this program is only 7 instructions long, the CPU ended up executing 13 instructions,

虽然程序只有 7 个指令，但 CPU 执行了 13 个指令,

and that's because it looped twice internally.

因为在内部循环了 2 次.

This code calculated the remainder if we divide 5 into 11, which is one.

这些代码其实是算余数的，11除5余1

With a few extra lines of code, we could also keep track of how many loops we did, the count

如果加多几行指令，我们还可以跟踪循环了多少次

of which would be how many times 5 went into 11… we did two loops, so that means 5 goes

11除5，循环2次

into 11 two times... with a remainder of 1.

余1

And of course this code could work for any two numbers, which we can just change in memory

当然，我们可以用任意2个数，7和81，18和54，什么都行

to whatever we want: 7 and 81, 18 and 54, it doesn’t matter

当然，我们可以用任意2个数，7和81，18和54，什么都行

-- that’s the power of software!

这就是软件的强大之处

Software also allowed us to do something our hardware could not.

软件还让我们做到硬件做不到的事

Remember, our ALU didn’t have the functionality to divide two numbers,

ALU 可没有除法功能

instead it’s the program we made that gave us that functionality.

是程序给了我们这个功能.

And then other programs can use our divide program to do even fancier things.

别的程序也可以用我们的除法程序，来做其他事情

And you know what that means.

这意味着 一层新抽象！

New levels of abstraction!

这意味着 一层新抽象！

So, our hypothetical CPU is very basic – all of its instructions are 8 bits long,

我们这里假设的 CPU 很基础，所有指令都是 8 位,

with the opcode occupying only the first four bits.

操作码只占了前面 4 位

So even if we used every combination of 4 bits, our CPU would only be able to support

即便用尽 4 位，也只能代表 16 个指令

a maximum of 16 different instructions.

即便用尽 4 位，也只能代表 16 个指令

On top of that, several of our instructions used the last 4 bits to specify a memory location.

而且我们有几条指令，是用后 4 位来指定内存地址

But again, 4 bits can only encode 16 different values,

因为 4 位最多只能表示 16 个值，

meaning we can address a maximum of 16 memory locations - that’s not a lot to work with.

所以我们只能操作 16 个地址，这可不多.

For example, we couldn’t even JUMP to location 17,

我们甚至不能 JUMP 17

because we literally can’t fit the number 17 into 4 bits.

因为 4 位二进制无法表示数字 17

For this reason, real, modern CPUs use two strategies.

因此，真正的现代 CPU 用两种策略

The most straightforward approach is just to have bigger instructions, with more bits,

最直接的方法是用更多位来代表指令，比如 32 位或 64 位

like 32 or 64 bits.

最直接的方法是用更多位来代表指令，比如 32 位或 64 位

This is called the instruction length.

这叫 指令长度

Unsurprisingly.

毫不意外

The second approach is to use variable length instructions.

第二个策略是 "可变指令长度"

For example, imagine a CPU that uses 8 bit opcodes.

举个例子，比如某个 CPU 用 8 位长度的操作码

When the CPU sees an instruction that needs no extra values, like the HALT instruction,

如果看到 HALT 指令，HALT 不需要额外数据

it can just execute it immediately.

那么会马上执行.

However, if it sees something like a JUMP instruction, it knows it must also fetch

如果看到 JUMP，它得知道位置值

the address to jump to, which is saved immediately behind the JUMP instruction in memory.

这个值在 JUMP 的后面

This is called, logically enough, an Immediate Value.

这叫 "立即值"

In such processor designs, instructions can be any number of bytes long,

这样设计，指令可以是任意长度

which makes the fetch cycle of the CPU a tad more complicated.

但会让读取阶段复杂一点点

Now, our example CPU and instruction set is hypothetical,

要说明的是，我们拿来举例的 CPU 和指令集都是假设的,

designed to illustrate key working principles.

是为了展示核心原理

So I want to leave you with a real CPU example.

所以我们来看个真的 CPU 例子.

In 1971, Intel released the 4004 processor.

1971年，英特尔发布了 4004 处理器.

It was the first CPU put all into a single chip

这是第一次把 CPU 做成一个芯片 \N 给后来的英特尔处理器打下了基础

and paved the path to the intel processors we know and love today.

这是第一次把 CPU 做成一个芯片 \N 给后来的英特尔处理器打下了基础

It supported 46 instructions, shown here.

它支持 46 个指令

Which was enough to build an entire working computer.

足够做一台能用的电脑

And it used many of the instructions we’ve talked about like JUMP ADD SUBTRACT and LOAD.

它用了很多我们说过的指令，比如 JUMP ADD SUB LOAD

It also uses 8-bit immediate values, like we just talked about, for things like JUMPs,

它也用 8 位的"立即值"来执行 JUMP, 以表示更多内存地址.

in order to address more memory.

它也用 8 位的"立即值"来执行 JUMP, 以表示更多内存地址.

And processors have come a long way since 1971.

处理器从 1971 年到现在发展巨大.

A modern computer processor, like an Intel Core i7, has thousands of different instructions

现代 CPU, 比如英特尔酷睿 i7, 有上千个指令和指令变种

and instruction variants, ranging from one to fifteen bytes long.

长度从1到15个字节.

For example, there’s over a dozens different opcodes just for variants of ADD!

举例，光 ADD 指令就有很多变种!

And this huge growth in instruction set size is due in large part to extra bells and whistles

指令越来越多，是因为给 CPU 设计了越来越多功能

that have been added to processor designs overtime, which we’ll talk about next episode.

下集我们会讲

See you next week!

下周见

Hi, I’m Carrie Anne and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

As we’ve discussed throughout the series, computers have come a long way from mechanical devices

随着本系列进展，我们知道计算机进步巨大

capable of maybe one calculation per second,

从 1 秒 1 次运算，到现在有千赫甚至兆赫的CPU

to CPUs running at kilohertz and megahertz speeds.

从 1 秒 1 次运算，到现在有千赫甚至兆赫的CPU

The device you’re watching this video on right now is almost certainly running at Gigahertz speeds

你现在看视频的设备八成也有 GHz 速度

- that’s billions of instructions executed every second.

1 秒十亿条指令

Which, trust me, is a lot of computation!

这是很大的计算量！

In the early days of electronic computing, processors were typically made faster by

早期计算机的提速方式是 减少晶体管的切换时间.

improving the switching time of the transistors inside the chip

早期计算机的提速方式是 减少晶体管的切换时间.

- the ones that make up all the logic gates, ALUs

晶体管组成了逻辑门，ALU 以及前几集的其他组件

and other stuff we’ve talked about over the past few episodes.

晶体管组成了逻辑门，ALU 以及前几集的其他组件

But just making transistors faster and more efficient only went so far, so processor designers

但这种提速方法最终会碰到瓶颈，所以处理器厂商

have developed various techniques to boost performance allowing not only simple instructions

发明各种新技术来提升性能，不但让简单指令运行更快

to run fast, but also performing much more sophisticated operations.

也让它能进行更复杂的运算

Last episode, we created a small program for our CPU that allowed us to divide two numbers.

上集我们写了个做除法的程序，给 CPU 执行

We did this by doing many subtractions in a row... so, for example, 16 divided by 4

方法是做一连串减法，比如16除4 会变成

could be broken down into the smaller problem of 16 minus 4, minus 4, minus 4, minus 4.

16-4 -4 -4 -4

When we hit zero, or a negative number, we knew that we we’re done.

碰到 0 或负数才停下.

But this approach gobbles up a lot of clock cycles, and isn’t particularly efficient.

但这种方法要多个时钟周期，很低效

So most computer processors today have divide as one of the instructions

所以现代 CPU 直接在硬件层面设计了除法 \N 可以直接给 ALU 除法指令

that the ALU can perform in hardware.

所以现代 CPU 直接在硬件层面设计了除法 \N 可以直接给 ALU 除法指令

Of course, this extra circuitry makes the ALU bigger and more complicated to design,

这让 ALU 更大也更复杂一些

but also more capable - a complexity-for-speed tradeoff that

但也更厉害 - \N 复杂度 vs 速度 的平衡在计算机发展史上经常出现

has been made many times in computing history.

但也更厉害 - \N 复杂度 vs 速度 的平衡在计算机发展史上经常出现

For instance, modern computer processors now have special circuits for things like

举例，现代处理器有专门电路来处理 \N 图形操作, 解码压缩视频, 加密文档 等等

graphics operations, decoding compressed video, and encrypting files

举例，现代处理器有专门电路来处理 \N 图形操作, 解码压缩视频, 加密文档 等等

all of which are operations that would take many many many clock cycles to perform with standard operations.

如果用标准操作来实现，要很多个时钟周期.

You may have even heard of processors with MMX, 3DNow!, or SSE.

你可能听过某些处理器有 MMX, 3DNOW, SEE

These are processors with additional, fancy circuits that allow them to

它们有额外电路做更复杂的操作

execute additional fancy instructions - for things like gaming and encryption.

用于游戏和加密等场景

These extensions to the instruction set have grown, and grown over time, and once people

指令不断增加，人们一旦习惯了它的便利就很难删掉

have written programs to take advantage of them, it’s hard to remove them.

指令不断增加，人们一旦习惯了它的便利就很难删掉

So instruction sets tend to keep getting larger and larger keeping all the old opcodes around for backwards compatibility.

所以为了兼容旧指令集，指令数量越来越多

The Intel 4004, the first truly integrated CPU, had 46 instructions

英特尔 4004，第一个集成CPU，有 46 条指令

- which was enough to build a fully functional computer.

足够做一台能用的计算机

But a modern computer processor has thousands of different instructions,

但现代处理器有上千条指令，有各种巧妙复杂的电路

which utilize all sorts of clever and complex internal circuitry.

但现代处理器有上千条指令，有各种巧妙复杂的电路

Now, high clock speeds and fancy instruction sets lead to another problem

超高的时钟速度带来另一个问题

- getting data in and out of the CPU quickly enough.

- 如何快速传递数据给 CPU

It’s like having a powerful steam locomotive, but no way to shovel in coal fast enough.

就像有强大的蒸汽机 但无法快速加煤

In this case, the bottleneck is RAM.

RAM 成了瓶颈

RAM is typically a memory module that lies outside the CPU.

RAM 是 CPU 之外的独立组件

This means that data has to be transmitted to and from RAM along sets of data wires,

意味着数据要用线来传递，叫"总线"

called a bus.

意味着数据要用线来传递，叫"总线"

This bus might only be a few centimeters long,

总线可能只有几厘米

and remember those electrical signals are traveling near the speed of light,

别忘了电信号的传输接近光速

but when you are operating at gigahertz speeds

但 CPU 每秒可以处理上亿条指令

– that’s billionths of a second – even this small delay starts to become problematic.

很小的延迟也会造成问题

It also takes time for RAM itself to lookup the address, retrieve the data

RAM 还需要时间找地址 \N 取数据，配置，输出数据

and configure itself for output.

RAM 还需要时间找地址 \N 取数据，配置，输出数据

So a “load from RAM” instruction might take dozens of clock cycles to complete, and during

一条"从内存读数据"的指令可能要多个时钟周期

this time the processor is just sitting there idly waiting for the data.

CPU 空等数据

One solution is to put a little piece of RAM right on the CPU -- called a cache.

解决延迟的方法之一是 \N 给 CPU 加一点 RAM - 叫"缓存"

There isn’t a lot of space on a processor’s chip,

因为处理器里空间不大，所以缓存一般只有 KB 或 MB

so most caches are just kilobytes or maybe megabytes in size,

因为处理器里空间不大，所以缓存一般只有 KB 或 MB

where RAM is usually gigabytes.

而 RAM 都是 GB 起步

Having a cache speeds things up in a clever way.

缓存提高了速度

When the CPU requests a memory location from RAM, the RAM can transmit

CPU 从 RAM 拿数据时 \N RAM 不用传一个，可以传一批

not just one single value, but a whole block of data.

CPU 从 RAM 拿数据时 \N RAM 不用传一个，可以传一批

This takes only a little bit more time,

虽然花的时间久一点，但数据可以存在缓存

but it allows this data block to be saved into the cache.

虽然花的时间久一点，但数据可以存在缓存

This tends to be really useful because computer data is often arranged and processed sequentially.

这很实用，因为数据常常是一个个按顺序处理

For example, let say the processor is totalling up daily sales for a restaurant.

举个例子，算餐厅的当日收入

It starts by fetching the first transaction from RAM at memory location 100.

先取 RAM 地址 100 的交易额

The RAM, instead of sending back just that one value, sends a block of data, from memory

RAM 与其只给1个值，直接给一批值

location 100 through 200, which are then all copied into the cache.

把地址100到200都复制到缓存

Now, when the processor requests the next transaction to add to its running total, the

当处理器要下一个交易额时

value at address 101, the cache will say “Oh, I’ve already got that value right here,

地址 101，缓存会说："我已经有了，现在就给你"

so I can give it to you right away!”

地址 101，缓存会说："我已经有了，现在就给你"

And there’s no need to go all the way to RAM.

不用去 RAM 取数据

Because the cache is so close to the processor,

因为缓存离 CPU 近, 一个时钟周期就能给数据 - CPU 不用空等！

it can typically provide the data in a single clock cycle -- no waiting required.

因为缓存离 CPU 近, 一个时钟周期就能给数据 - CPU 不用空等！

This speeds things up tremendously over having to go back and forth to RAM every single time.

比反复去 RAM 拿数据快得多

When data requested in RAM is already stored in the cache like this it’s called a

如果想要的数据已经在缓存，叫 缓存命中

cache hit,

如果想要的数据已经在缓存，叫 缓存命中

and if the data requested isn’t in the cache, so you have to go to RAM, it’s a called

如果想要的数据不在缓存，叫 缓存未命中

a cache miss.

如果想要的数据不在缓存，叫 缓存未命中

The cache can also be used like a scratch space,

缓存也可以当临时空间，存一些中间值，适合长/复杂的运算

storing intermediate values when performing a longer, or more complicated calculation.

缓存也可以当临时空间，存一些中间值，适合长/复杂的运算

Continuing our restaurant example, let’s say the processor has finished totalling up

继续餐馆的例子，假设 CPU 算完了一天的销售额

all of the sales for the day, and wants to store the result in memory address 150.

想把结果存到地址 150

Like before, instead of going back all the way to RAM to save that value,

就像之前，数据不是直接存到 RAM

it can be stored in cached copy, which is faster to save to,

而是存在缓存，这样不但存起来快一些

and also faster to access later if more calculations are needed.

如果还要接着算，取值也快一些

But this introduces an interesting problem -

但这样带来了一个有趣的问题

- the cache’s copy of the data is now different to the real version stored in RAM.

缓存和 RAM 不一致了.

This mismatch has to be recorded, so that at some point everything can get synced up.

这种不一致必须记录下来，之后要同步

For this purpose, the cache has a special flag for each block of memory it stores, called

因此缓存里每块空间 有一个特殊标记

the dirty bit

叫 "脏位"

-- which might just be the best term computer scientists have ever invented.

- 这可能是计算机科学家取的最贴切的名字

Most often this synchronization happens when the cache is full,

同步一般发生在 当缓存满了而 CPU 又要缓存时

but a new block of memory is being requested by the processor.

同步一般发生在 当缓存满了而 CPU 又要缓存时

Before the cache erases the old block to free up space, it checks its dirty bit,

在清理缓存腾出空间之前，会先检查 "脏位"

and if it’s dirty, the old block of data is written back to RAM before loading in the new block.

如果是"脏"的, 在加载新内容之前, 会把数据写回 RAM

Another trick to boost cpu performance is called instruction pipelining.

另一种提升性能的方法叫 "指令流水线"

Imagine you have to wash an entire hotel’s worth of sheets,

想象下你要洗一整个酒店的床单

but you’ve only got one washing machine and one dryer.

但只有 1 个洗衣机, 1 个干燥机

One option is to do it all sequentially: put a batch of sheets in the washer

选择1：按顺序来，放洗衣机等 30 分钟洗完

and wait 30 minutes for it to finish.

选择1：按顺序来，放洗衣机等 30 分钟洗完

Then take the wet sheets out and put them in the dryer and wait another 30 minutes for that to finish.

然后拿出湿床单，放进干燥机等 30 分钟烘干

This allows you to do one batch of sheets every hour.

这样1小时洗一批

Side note: if you have a dryer that can dry a load of laundry in 30 minutes,

另外一说：如果你有 30 分钟就能烘干的干燥机

Please tell me the brand and model in the comments, because I’m living with 90 minute dry times, minimum.

请留言告诉我是什么牌子，我的至少要 90 分钟.

But, even with this magic clothes dryer,

即使有这样的神奇干燥机, \N 我们可以用"并行处理"进一步提高效率

you can speed things up even more if you parallelize your operation.

即使有这样的神奇干燥机, \N 我们可以用"并行处理"进一步提高效率

As before, you start off putting one batch of sheets in the washer.

就像之前，先放一批床单到洗衣机

You wait 30 minutes for it to finish.

等 30 分钟洗完

Then you take the wet sheets out and put them in the dryer.

然后把湿床单放进干燥机

But this time, instead of just waiting 30 minutes for the dryer to finish,

但这次，与其干等 30 分钟烘干，\N 可以放另一批进洗衣机

you simultaneously start another load in the washing machine.

但这次，与其干等 30 分钟烘干，\N 可以放另一批进洗衣机

Now you’ve got both machines going at once.

让两台机器同时工作

Wait 30 minutes, and one batch is now done, one batch is half done,

30 分钟后，一批床单完成, 另一批完成一半

and another is ready to go in.

另一批准备开始

This effectively doubles your throughput.

效率x2！

Processor designs can apply the same idea.

处理器也可以这样设计

In episode 7, our example processor performed the fetch-decode-execute cycle sequentially

第7集，我们演示了 CPU 按序处理

and in a continuous loop: Fetch-decode-execute, fetch-decode-execute, fetch-decode-execute, and so on

取指 → 解码 → 执行, 不断重复

This meant our design required three clock cycles to execute one instruction.

这种设计，三个时钟周期执行 1 条指令

But each of these stages uses a different part of the CPU,

但因为每个阶段用的是 CPU 的不同部分

meaning there is an opportunity to parallelize!

意味着可以并行处理！

While one instruction is getting executed, the next instruction could be getting decoded,

"执行"一个指令时，同时"解码"下一个指令

and the instruction beyond that fetched from memory.

"读取"下下个指令

All of these separate processes can overlap

不同任务重叠进行，同时用上 CPU 里所有部分.

so that all parts of the CPU are active at any given time.

不同任务重叠进行，同时用上 CPU 里所有部分.

In this pipelined design, an instruction is executed every single clock cycle

这样的流水线 每个时钟周期执行1个指令

which triples the throughput.

吞吐量 x 3

But just like with caching this can lead to some tricky problems.

和缓存一样，这也会带来一些问题

A big hazard is a dependency in the instructions.

第一个问题是 指令之间的依赖关系

For example, you might fetch something that the currently executing instruction is just about to modify,

举个例子，你在读某个数据 \N 而正在执行的指令会改这个数据

which means you’ll end up with the old value in the pipeline.

也就是说拿的是旧数据

To compensate for this, pipelined processors have to look ahead for data dependencies,

因此流水线处理器 要先弄清数据依赖性

and if necessary, stall their pipelines to avoid problems.

必要时停止流水线，避免出问题

High end processors, like those found in laptops and smartphones,

高端 CPU，比如笔记本和手机里那种

go one step further and can dynamically reorder instructions with dependencies

会更进一步，动态排序 有依赖关系的指令

in order to minimize stalls and keep the pipeline moving,

最小化流水线的停工时间

which is called out-of-order execution.

这叫 "乱序执行"

As you might imagine, the circuits that figure this all out are incredibly complicated.

和你猜的一样，这种电路非常复杂

Nonetheless, pipelining is tremendously effective and almost all processors implement it today.

但因为非常高效，几乎所有现代处理器都有流水线

Another big hazard are conditional jump instructions -- we talked about one example, a JUMP NEGATIVE,last episode.

第二个问题是 "条件跳转"，比如上集的 JUMP NEGATIVE

These instructions can change the execution flow of a program depending on a value.

这些指令会改变程序的执行流

A simple pipelined processor will perform a long stall when it sees a jump instruction,

简单的流水线处理器，看到 JUMP 指令会停一会儿 \N 等待条件值确定下来

waiting for the value to be finalized.

简单的流水线处理器，看到 JUMP 指令会停一会儿 \N 等待条件值确定下来

Only once the jump outcome is known, does the processor start refilling its pipeline.

一旦 JUMP 的结果出了，处理器就继续流水线

But, this can produce long delays, so high-end processors have some tricks to deal with this problem too.

因为空等会造成延迟，所以高端处理器会用一些技巧

Imagine an upcoming jump instruction as a fork in a road - a branch.

可以把 JUMP 想成是 "岔路口"

Advanced CPUs guess which way they are going to go, and start filling their pipeline with

高端 CPU 会猜哪条路的可能性大一些

instructions based off that guess – a technique called speculative execution.

然后提前把指令放进流水线，这叫 "推测执行"

When the jump instruction is finally resolved, if the CPU guessed correctly,

当 JUMP 的结果出了，如果 CPU 猜对了

then the pipeline is already full of the correct instructions and it can motor along without delay.

流水线已经塞满正确指令，可以马上运行

However, if the CPU guessed wrong, it has to discard all its speculative results and

如果 CPU 猜错了，就要清空流水线

perform a pipeline flush - sort of like when you miss a turn and have to do a u-turn to

就像走错路掉头

get back on route, and stop your GPS’s insistent shouting.

让 GPS 不要再！叫！了！

To minimize the effects of these flushes, CPU manufacturers have developed sophisticated

为了尽可能减少清空流水线的次数，CPU 厂商开发了复杂的方法

ways to guess which way branches will go, called branch prediction.

来猜测哪条分支更有可能，叫"分支预测"

Instead of being a 50/50 guess, today’s processors can often guess with over 90% accuracy!

现代 CPU 的正确率超过 90%

In an ideal case, pipelining lets you complete one instruction every single clock cycle,

理想情况下，流水线一个时钟周期完成 1 个指令

but then superscalar processors came along

然后"超标量处理器"出现了，一个时钟周期完成多个指令

which can execute more than one instruction per clock cycle.

然后"超标量处理器"出现了，一个时钟周期完成多个指令

During the execute phase even in a pipelined design,

即便有流水线设计，在指令执行阶段

whole areas of the processor might be totally idle.

处理器里有些区域还是可能会空闲

For example, while executing an instruction that fetches a value from memory,

比如，执行一个 "从内存取值" 指令期间

the ALU is just going to be sitting there, not doing a thing.

ALU 会闲置

So why not fetch-and-decode several instructions at once, and whenever possible, execute instructions

所以一次性处理多条指令（取指令+解码） 会更好.

that require different parts of the CPU all at the same time

如果多条指令要 ALU 的不同部分，就多条同时执行

But we can take this one step further and add duplicate circuitry

我们可以再进一步，加多几个相同的电路 \N 执行出现频次很高的指令

for popular instructions.

我们可以再进一步，加多几个相同的电路 \N 执行出现频次很高的指令

For example, many processors will have four, eight or more identical ALUs,

举例，很多 CPU 有四个, 八个甚至更多 完全相同的ALU

so they can execute many mathematical instructions all in parallel!

可以同时执行多个数学运算

Ok, the techniques we’ve discussed so far primarily optimize the execution throughput

好了，目前说过的方法，都是优化 1 个指令流的吞吐量

of a single stream of instructions,

好了，目前说过的方法，都是优化 1 个指令流的吞吐量

but another way to increase performance is to run several streams of instructions at once

另一个提升性能的方法是 同时运行多个指令流

with multi-core processors.

用多核处理器

You might have heard of dual core or quad core processors.

你应该听过双核或四核处理器

This means there are multiple independent processing units inside of a single CPU chip.

意思是一个 CPU 芯片里，有多个独立处理单元

In many ways, this is very much like having multiple separate CPUs,

很像是有多个独立 CPU

but because they’re tightly integrated, they can share some resources,

但因为它们整合紧密，可以共享一些资源

like cache, allowing the cores to work together on shared computations.

比如缓存，使得多核可以合作运算

But, when more cores just isn’t enough, you can build computers with multiple independent CPUs!

但多核不够时，可以用多个 CPU

High end computers, like the servers streaming this video from YouTube’s datacenter, often

高端计算机，比如现在给你传视频的 Youtube 服务器

need the extra horsepower to keep it silky smooth for the hundreds of people watching simultaneously.

需要更多马力，让上百人能同时流畅观看

Two- and four-processor configuration are the most common right now,

2个或4个CPU是最常见的

but every now and again even that much processing power isn’t enough.

但有时人们有更高的性能要求

So we humans get extra ambitious and build ourselves a supercomputer!

所以造了超级计算机！

If you’re looking to do some really monster calculations

如果要做怪兽级运算

– like simulating the formation of the universe - you’ll need some pretty serious compute power.

比如模拟宇宙形成，你需要强大的计算能力

A few extra processors in a desktop computer just isn’t going to cut it.

给普通台式机加几个 CPU 没什么用

You’re going to need a lot of processors.

你需要很多处理器！

No.. no... even more than that.

不…不…还要更多

A lot more!

更多

When this video was made, the world’s fastest computer was located in

截止至视频发布，世上最快的计算机在

The National Supercomputing Center in Wuxi, China.

中国无锡的国家超算中心

The Sunway TaihuLight contains a brain-melting 40,960 CPUs, each with 256 cores!

神威·太湖之光有 40960 个CPU，每个 CPU 有 256 个核心

Thats over ten million cores in total... and each one of those cores runs at 1.45 gigahertz.

总共超过1千万个核心，每个核心的频率是 1.45GHz

In total, this machine can process 93 Quadrillion -- that’s 93 million-billions

每秒可以进行 9.3 亿亿次浮点数运算

floating point math operations per second, knows as FLOPS.

也叫 每秒浮点运算次数 (FLOPS)

And trust me, that’s a lot of FLOPS!!

相信我 这个速度很可怕

No word on whether it can run Crysis at max settings, but I suspect it might.

没人试过跑最高画质的《孤岛危机》但我估计没问题

So long story short, not only have computer processors gotten a lot faster over the years,

长话短说，这些年处理器不但大大提高了速度

but also a lot more sophisticated, employing all sorts of clever tricks to squeeze out

而且也变得更复杂，用各种技巧

more and more computation per clock cycle.

榨干每个时钟周期 做尽可能多运算

Our job is to wield that incredible processing power to do cool and useful things.

我们的任务是利用这些运算能力，做又酷又实用的事

That’s the essence of programming, which we’ll start discussing next episode.

编程就是为了这个，我们下集说

See you next week.

下周见

Hi, I'm Carrie Anne and welcome to Crash Course Computer Science.

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Over the last few episodes,

前几集我们把重点放在计算机的原理

We've talked a lot about the mechanics of how computers work.

前几集我们把重点放在计算机的原理

How they use complex circuits to save and retrieve values from memory,

怎么从内存读写数据，执行操作

and perform operations on those values

怎么从内存读写数据，执行操作

like adding two numbers together.

比如把两个数字加在一起

We've even briefly talked about sequences of operations,

还简单讲了下指令的执行，也就是计算机程序

which is a computer program

还简单讲了下指令的执行，也就是计算机程序

What we haven't talked about is how a program gets into a computer?

但我们还没讲的是：程序如何"进入"计算机

You might remember in episode 7 and 8 ,

你应该记得在第 7, 8 集，我们一步步讲了例子程序

we step through some simple example programs for the CPU that we had created

你应该记得在第 7, 8 集，我们一步步讲了例子程序

For simplicity, we just waved our hands and said that the program was already magically in memory

当时为了简单，我们假设程序已经魔法般在内存里了

But in reality, programs have to be loaded into a computer's memory.

但事实是，程序需要加载进内存

It's not magic. It's computer science

这不是魔法，是计算机科学！

The need to program machines existed way before the development of computers.

给机器编程这个需求，早在计算机出现之前就有了

The most famous example of this was in textile manufacturing

最著名的例子来自纺织业

If you just wanted to weave a big red tablecloth

如果你只想织一块红色大桌布

You could simply feed red thread into a loom and let it run

可以直接放红线进织布机

What about if you wanted the cloth to have a pattern like stripes or plaid?

但如果想要图案怎么办? 比如条纹或者方格

Workers would have to periodically reconfigure the loom as dictated by the pattern,

工人要每隔一会儿 调整一次织布机

but this was labor intensive which made patterned fabrics expensive.

因为非常消耗劳动力，所以图案纺织品很贵

The presence or absence of a hole in the card determined if a specific thread was held high or low in the loom

特定位置有没有穿孔，决定了线是高是低

Such as the cross thread, called the weft, passed above or below the thread

横线是从上/从下穿过

To vary the pattern across rows these punch cards were arranged in long chains

为了让每行图案不同，纸卡连成长条

Forming a sequence of commands for the loom.

形成连续指令

Sound familiar?

听起来很熟？

Many consider Jacquard loom to be one of the earliest forms of programming.

很多人认为雅卡尔织布机是最早的编程

Punched cards, turned out to be a cheap, reliable, fairly human-readable way to store data.

事实证明 穿孔纸卡便宜、可靠、也易懂

Nearly a century later,

近一个世纪后

punch cards were used to help tabulate the 1890 US census

穿孔纸卡用于 1890 年美国人口普查

which we talked about in episode 1

我们在第一集提过

Each card held an individual person's data.

一张卡存一个人的信息

things like race

比如种族

marital status

婚姻状况

number of children

子女数量

country of birth and so on

出生国家 等等

for each demographic question

针对每个问题，人口普查工作者会在对应位置打孔

a census worker would punch out a hole of the appropriate position

针对每个问题，人口普查工作者会在对应位置打孔

when a card was fed into the tabulating machine

当卡片插入汇总机

a hole would cause the running total for that specific answer to be increased by one

孔会让对应总和值+1

in this way you could afeed the entire counties worth of people

可以插入整个国家人口的卡片

and at the end you'd have running totals for all of the questions that you ask

在结束后得到各个总值

It is important to note here that early tabulating machines were not truly computers

值得注意的是，早期汇总机不算计算机

as they can only do one thing-tabulate

因为它们只做一件事 - 汇总数据

their operation was fixed and not programmable

操作是固定的，不能编程

punched cards stored data, but not a program

穿孔纸卡存的是数据，不是程序.

over the next 60 years, these business machines grew in capability

之后60年，这些机器被加强，可以做减、乘、除

Adding features to subtract multiply divide

之后60年，这些机器被加强，可以做减、乘、除

and even make simple decisions about when to perform certain operations.

甚至可以做一些小决定，决定何时执行某指令

To trigger these functions appropriately

为了正确执行不同计算，程序员需要某种控制面板

so that different calculations could be performed, a programmer accessed a control panel

为了正确执行不同计算，程序员需要某种控制面板

this panel was full of little sockets into which a programmer would plug cables

面板有很多小插孔，程序员可以插电线

to pass values and signals between different parts of the machine

让机器的不同部分 互相传数据和信号

for this reason they were also called plug boards

因此也叫 "插线板"

Unfortunately this meant having to rewire the machine each time a different program needed to be run

不幸的是, 这意味着 运行不同程序要重新接线

And so by the 1920s these plug boards were made swappable

所以到 1920 年代，控制面板变成了可拔插

This not only made programming a lot more comfortable

让编程更方便

but also allowed for different programs be plugged into a machine

可以给机器插入不同程序

For example one board might be wired to calculate sales tax

比如，一个插线板算销售税，另一个算工资单

While another helps with payroll

比如，一个插线板算销售税，另一个算工资单

But plug boards were fiendishly complicated to program

但给插线板编程很复杂

This tangle of wires is a program for calculating a profit loss summary using an IBM 402 accounting machine

图中乱成一团的线 负责算盈亏总额 \N 用于 IBM 402 核算机

which were popular in the 1940s

在 1940 年代这样做很流行

And this style of plug board programming wasn't unique through electromechanical computers

用插线板编程 不只在机电计算机流行

The world's first general-purpose electronic computer, the ENIAC, completed in 1946

世上第一台通用电子计算机，ENIAC，完成于 1946 年

used a ton of them

用了一大堆插线板

Even after a program had been completely figured out on paper

程序在纸上设计好之后

Physically wiring up the ENIAC and getting the program to run could take upwards of three weeks

给 ENIAC 连线，最多可能花三个星期

Given the enormous cost of these early computers, weeks of downtime simply to switch programs was unacceptable

因为早期计算机非常昂贵 \N 停机几个星期只为换程序 完全无法接受

and the new faster more flexible way to program machines was badly needed

人们急需更快、更灵活的新方式来编程

Fortunately by the late 1940s and into the 50s

幸运的是，到 1940 年代晚期 1950 年代初

electronic memory was becoming feasible

内存变得可行

As costs fell, memory size grew, instead of storing a program as a physical plug board of wires

价格下降, 容量上升. 与其把程序存在插线板

it became possible to store a program entirely in a computer's memory

存在内存变得可行

where it could be easily changed by programmers and quickly accessed by the CPU

这样程序易于修改、方便 CPU 快速读取

these machines were called Stored-program Computers

这类机器叫 "存储程序计算机"

With enough computer memory you could store not only the program you wanted to run

如果内存足够，不仅可以存要运行的程序

but also any data your program would need

还可以存程序需要的数据

including new values it created along the way

包括程序运行时产生的新数据

Unifying the program and data into a single shared memory is called the Von Neumann Architecture

程序和数据都存在一个地方，叫 "冯诺依曼结构"

named after John Von Neumann

命名自 约翰·冯·诺依曼

a prominent mathematician and physicist who worked on the Manhattan project and several early electronic computers

杰出的数学家和物理学家 \N 参与了曼哈顿计划和早期电子计算机项目

and once said I am thinking about something much more important than Bombs

他曾说：我在思考比炸弹重要得多的东西

I'm thinking about computers

计算机

The hallmarks of a Von Neumann computer are a processing unit containing an arithmetic logic unit

冯诺依曼计算机的标志是，一个处理器(有算术逻辑单元)+

data registers and instruction register and instruction address register

数据寄存器+指令寄存器+指令地址寄存器

And finally a memory to store both data and instructions

+内存（负责存数据和指令)

Hopefully this sounds familiar

希望这听起来很耳熟

Because we actually built a Von Neumann computer in episode 7

因为第7集我们造了一个冯诺依曼计算机

The very first Von Neumann Architecture Stored-program computer

第一台冯诺依曼架构的"储存程序计算机"

was constructed in 1948 by the University of Manchester, nicknamed Baby.

由曼彻斯特大学于 1948 年建造完成，绰号"宝宝"

and even the computer you are watching this video right now

甚至你现在看视频的计算机，也在用一样的架构

uses the same architecture

甚至你现在看视频的计算机，也在用一样的架构

Now electronic computer memory is great and all

虽然有内存很棒

but you still have to load the program and data into the computer before it can run

但程序和数据 依然需要某种方式输入计算机

and for this reason punch cards were used

所以用穿孔纸卡

Let's get to the Thought bubbles

让我们进入 思维泡泡

Well into the 1980s almost all computers have a punch card reader

到1980年代，几乎所有的计算机都有穿孔纸卡读取器

which could suck in a single punch card at a time

可以吸入一张卡片，把卡片内容写进内存

and write the contents of the card into the computer's memory

可以吸入一张卡片，把卡片内容写进内存

If you load it in a stack of punch cards,

如果放了一叠卡片，读取器会一个个写进内存

the reader would load them all into memory sequentially as a big block

如果放了一叠卡片，读取器会一个个写进内存

once the program and data were in memory, the computer would be told to execute it

一旦程序和数据写入完毕，电脑会开始执行

Of course even simple computer programs might have hundreds of instructions

即便简单程序也有几百条指令，要用一叠纸卡来存

which meant that programs were stored as stacks of punch cards

即便简单程序也有几百条指令，要用一叠纸卡来存

So if you ever have the misfortune of accidentally dropping your program on the floor

如果不小心摔倒弄撒了

it could take you hours days or even weeks to put the code back in the right order

要花上几小时、几天、甚至几周来整理

A common trick was to draw a diagonal line on the side of the card stack called striping,

有个小技巧是 在卡片侧面画对角线

so you'd have at least some clue how to get it back into the right order

如果弄散了，整理起来会方便很多

The largest program ever punched into punch cards was the US Air Force's SAGE air defense system, completed in 1955.

用纸卡的最大型程序 \N 是美国空军的 SAGE 防空系统，于 1955 年完成

and its peak, the project is said to have employed 20% of the world's programmers

据称顶峰时期 雇佣了世上 20% 程序员

Its main control program was stored on a whopping 62,500 punch cards

主控制程序用了 62500 张穿孔纸卡

which is equivalent to roughly 5 megabytes of data

等同于大约 5MB 的数据

Pretty underwhelming by today's standards

以如今的标准，不值一提

And punch cards weren't only useful for getting data into computers

穿孔纸卡不仅可以往计算机放数据

but also getting data out of them

还可以取出数据

At the end of a program results could be written out of computer memory and onto punch cards by, well, punching cards

程序运行到最后，结果可以输到纸卡上，方式嘛，当然是打孔

then this data could be analyzed by humans or loaded into a second program for additional computation

然后人可以分析结果，或者再次放进计算机，做进一步计算

Thanks, thought-bubble

谢了 思维泡泡

A close cousin to punch cards was punched paper tape

穿孔纸卡 的亲戚是纸带

Which is basically the same idea, but continuous instead of being on individual cards

基本是一回事，只不过更连续，不是一张张卡.

And of course we haven't talked about Hard Drives, CD-ROMs, DVDs, USB-Thumb drives and other similar goodies

当然我们还没提硬盘, 只读光盘, DVD, U盘等等

We'll get to those more advanced types of data storage in a future episode

以后我们会讲这些更先进的存储方法

Finally in addition to plug boards and punch paper

最后，除了插线板和穿孔纸卡

there was another common way to program and control computers in pre-1980

在 1980 年代前，还有一种常见编程方式

Panel programming

面板编程

Rather than having to physically plug in cables to activate certain functions

与其插一堆线到插线板

this could also be done with huge panels full of switches and buttons

可以用一大堆开关和按钮，做到一样的效果

And there were indicator lights to display the status of various functions and values in memory

面板上有指示灯，代表各种函数的状态和内存中的值

Computers of the 50s and 60s often featured huge control consoles that look like this

50和60年代的计算机，一般都有这样巨大的控制台

Although it was rare to input a whole program using just switches,it was possible

很少有人只用开关来输入一整个程序，但技术上是可行的

And early home computers made for the hobbyist market use switches extensively

早期针对计算机爱好者的家用计算机，大量使用了开关

because most home users couldn't afford expensive peripherals like punch card readers

因为大多数家庭用户负担不起昂贵的外围设备 \N 比如穿孔纸卡读取器

The first commercially successful home computer was the Altair 8800

第一款取得商业成功的家用计算机是 Altair 8800

which sold in two versions: Pre-assembled and the Kit

有两种版本可以买: \N 1. 预先装好的整机 \N 2. 需要组装的组件

the Kit which was popular with amateur computing enthusiasts,

计算机爱好者 喜欢买组件版

sold for the then unprecedented low price are around $400 in 1975

售价极低，在 1975 年卖 400 美元左右

Or about $2,000 in 2017

相当于 2017 年的 2000 美元

To program the 8800, you'd literally toggle the switches on the front panel

为了给 8800 编程，你要拨动面板上的开关

to enter the binary op-codes for the instruction you wanted

输入二进制操作码

Then you press the deposit button to write that value into memory

然后按 "存储键" 把值存入内存

Then in the next location in memory you toggle the switches again

然后会到下一个内存位置 \N 你可以再次拨开关，写下一个指令

for your next instruction deposit it and so on

重复这样做

When you finally entered your whole program into memory

把整个程序都写入内存之后

you would toggle the switches moves back to memory address 0

可以推动开关，回到内存地址0

press the run button and watch the little lights blink

然后按运行按钮，灯会闪烁

That was home computing in 1975, Wow.

这就是 1975 年的家用计算机, 哇.

Whether it was plug board, switches or punched paper

不管是插线板、开关或穿孔纸卡

Programming these early computers was the realm of experts

早期编程都是专家活

either professionals who did this for living or technology enthusiasts

不管是全职还是技术控，都要非常了解底层硬件

you needed intimate knowledge of the underlying hardware,

不管是全职还是技术控，都要非常了解底层硬件

so things like processor op-codes and register wits, to write programs

比如 操作码, 寄存器等, 才能写程序

This meant programming was hard and tedious and even professional engineers

所以编程很难，很烦

and scientists struggled to take full advantage of what computing could offer

哪怕工程师和科学家都无法 完全发挥计算机的能力

What was needed was a simpler way to tell computers what to do,

我们需要一种更简单方式 告诉计算机要做什么

a simpler way to write programs

一种更简单的编程方式

And that brings us to programming languages, which we'll talk about next episode

这带领我们到下一个话题 - 编程语言, 我们下集会讲

See you next week

下周见

This episode is brought to you by CuriosityStream.

本集由 CuriosityStream 赞助播出

Hi, I'm Carrie Anne and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

So far, for most of this series, we've focused on hardware

之前我们把重点放在硬件 - 组成计算机的物理组件

-- the physical components of computing --

之前我们把重点放在硬件 - 组成计算机的物理组件

things like: electricity and circuits, registers and RAM, ALUs and CPUs.

比如电，电路，寄存器，RAM，ALU，CPU

But programming at the hardware level is cumbersome and inflexible,

但在硬件层面编程非常麻烦

so programmers wanted a more versatile way to program computers

所以程序员想要一种更通用的方法编程

- what you might call a "softer" medium.

- 一种"更软的"媒介

That's right, we're going to talk about Software!

没错，我们要讲软件！

In episode 8, we walked through a simple program for the CPU we designed.

第 8 集我们一步步讲了一个简单程序

The very first instruction to be executed, the one at memory address 0, was 0010 1110.

第一条指令在内存地址 0：0010 1110

As we discussed, the first four bits of an instruction is the operation code,

之前说过，前 4 位是操作码

or OPCODE for short.

简称 OPCODE

On our hypothetical CPU, 0010 indicated a LOAD\_A instruction

对于这个假设 CPU，0010 代表 LOAD\_A 指令

-- which moves a value from memory into Register A.

- 把值从内存复制到寄存器 A

The second set of four bits defines the memory location,

后 4 位是内存地址，1110 是十进制的 14

in this case, 1110, which is 14 in decimal.

后 4 位是内存地址，1110 是十进制的 14

So what these eight numbers really mean is "LOAD Address 14 into Register A".

所以这 8 位表达的意思是 \N "读内存地址 14，放入寄存器 A"

We're just using two different languages.

只是用了两种不同语言

You can think of it like English and Morse Code.

可以想成是英语和摩尔斯码的区别

"Hello" and ".... . .-.. .-.. ---" mean the same thing -- hello! --

"你好" 和 ".... . .-.. .-.. ---" 是一个意思：你好

they're just encoded differently.

只是编码方式不同

English and Morse Code also have different levels of complexity.

英语和摩尔斯码的复杂度也不同

English has 26 different letters in its alphabet and way more possible sounds.

英文有 26 个字母以及各种发音

Morse only has dots and dashes.

摩尔斯码只有"点"和"线"

But, they can convey the same information, and computer languages are similar.

但它们可以传达相同的信息，计算机语言也类似.

As we've seen, computer hardware can only handle raw, binary instructions.

计算机能处理二进制，二进制是处理器的"母语"

This is the "language" computer processors natively speak.

计算机能处理二进制，二进制是处理器的"母语"

In fact, it's the only language they're able to speak.

事实上，它们\*只能\*理解二进制

It's called Machine Language or Machine Code.

这叫"机器语言"或"机器码"

In the early days of computing, people had to write entire programs in machine code.

在计算机早期阶段，必须用机器码写程序

More specifically, they'd first write a high-level version of a program on paper, in English,

具体来讲，会先在纸上用英语写一个"高层次版"

For example "retrieve the next sale from memory,

举例："从内存取下一个销售额，

then add this to the running total for the day, week and year,

然后加到天、周、年的总和

then calculate any tax to be added"

然后算税"

...and so on.

等等...

An informal, high-level description of a program like this is called Pseudo-Code.

这种对程序的高层次描述，叫 "伪代码"

Then, when the program was all figured out on paper,

在纸上写好后

they'd painstakingly expand and translate it into binary machine code by hand,

用"操作码表"把伪代码转成二进制机器码

using things like opcode tables.

用"操作码表"把伪代码转成二进制机器码

After the translation was complete, the program could be fed into the computer and run.

翻译完成后，程序可以喂入计算机并运行

As you might imagine, people quickly got fed up with this process.

你可能猜到了，很快人们就厌烦了

So, by the late 1940s and into the 50s,

所以在 1940~1950 年代

programmers had developed slightly higher-level languages that were more human-readable.

程序员开发出一种新语言， 更可读 更高层次

Opcodes were given simple names, called mnemonics,

每个操作码分配一个简单名字，叫"助记符"

which were followed by operands, to form instructions.

"助记符"后面紧跟数据，形成完整指令

So instead of having to write instructions as a bunch of 1's and 0's,

与其用 1 和 0 写代码，程序员可以写"LOAD\_A 14"

programmers could write something like "LOAD\_A 14".

与其用 1 和 0 写代码，程序员可以写"LOAD\_A 14"

We used this mnemonic in Episode 8 because it's so much easier to understand!

我们在第 8 集用过这个助记符，因为容易理解得多！

Of course, a CPU has no idea what "LOAD\_A 14" is.

当然，CPU 不知道 LOAD\_A 14 是什么

It doesn't understand text-based language, only binary.

它不能理解文字，只能理解二进制

And so programmers came up with a clever trick.

所以程序员想了一个技巧，写二进制程序来帮忙

They created reusable helper programs, in binary,

所以程序员想了一个技巧，写二进制程序来帮忙

that read in text-based instructions,

它可以读懂文字指令，自动转成二进制指令

and assemble them into the corresponding binary instructions automatically.

它可以读懂文字指令，自动转成二进制指令

This program is called

这种程序叫

-- you guessed it --

你可能猜到了

an Assembler.

汇编器

It reads in a program written in an Assembly Language

汇编器读取用"汇编语言"写的程序，然后转成"机器码"

and converts it to native machine code.

汇编器读取用"汇编语言"写的程序，然后转成"机器码"

"LOAD\_A 14" is one example of an assembly instruction.

"LOAD\_A 14" 是一个汇编指令的例子

Over time, Assemblers gained new features that made programming even easier.

随着时间推移，汇编器有越来越多功能，让编程更容易

One nifty feature is automatically figuring out JUMP addresses.

其中一个功能是自动分析 JUMP 地址

This was an example program I used in episode 8:

这里有一个第8集用过的例子：

Notice how our JUMP NEGATIVE instruction jumps to address 5,

注意, JUMP NEGATIVE 指令跳到地址 5

and our regular JUMP goes to address 2.

JUMP 指令跳到地址 2

The problem is, if we add more code to the beginning of this program,

问题是，如果在程序开头多加一些代码

all of the addresses would change.

所有地址都会变

That's a huge pain if you ever want to update your program!

更新程序会很痛苦！

And so an assembler does away with raw jump addresses,

所以汇编器不用固定跳转地址

and lets you insert little labels that can be jumped to.

而是让你插入可跳转的标签

When this program is passed into the assembler,

当程序被传入汇编器，汇编器会自己搞定跳转地址

it does the work of figuring out all of the jump addresses.

当程序被传入汇编器，汇编器会自己搞定跳转地址

Now the programmer can focus more on programming

程序员可以专心编程，不用管底层细节

and less on the underlying mechanics under the hood

程序员可以专心编程，不用管底层细节

enabling more sophisticated things to be built by hiding unnecessary complexity.

隐藏不必要细节来做更复杂的工作

As we've done many times in this series,

我们又提升了一层抽象

we're once again moving up another level of abstraction.

我们又提升了一层抽象

However, even with nifty assembler features like auto-linking JUMPs to labels,

然而，即使汇编器有这些厉害功能，比如自动跳转

Assembly Languages are still a thin veneer over machine code.

汇编只是修饰了一下机器码

In general, each assembly language instruction converts directly

一般来说，一条汇编指令对应一条机器指令

to a corresponding machine instruction - a one-to-one mapping -

一般来说，一条汇编指令对应一条机器指令

so it's inherently tied to the underlying hardware.

所以汇编码和底层硬件的连接很紧密

And the assembler still forces programmers to think about

汇编器仍然强迫程序员思考 用什么寄存器和内存地址

which registers and memory locations they will use.

汇编器仍然强迫程序员思考 用什么寄存器和内存地址

If you suddenly needed an extra value,

如果你突然要一个额外的数，可能要改很多代码

you might have to change a lot of code to fit it in.

如果你突然要一个额外的数，可能要改很多代码

Let's go to the Thought Bubble.

让我们进入思考泡泡

This problem did not escape Dr. Grace Hopper.

葛丽丝·霍普博士 也遇到了这个问题

As a US naval officer, she was one of the first programmers on the Harvard Mark 1 computer,

作为美国海军军官，她是哈佛1号计算机的首批程序员之一

which we talked about in Episode 2.

这台机器我们在第 2 集提过

This was a colossal, electro-mechanical beast

这台巨大机电野兽在 1944 年战时建造完成，帮助盟军作战

completed in 1944 as part of the allied war effort.

这台巨大机电野兽在 1944 年战时建造完成，帮助盟军作战

Programs were stored and fed into the computer on punched paper tape.

程序写在打孔纸带上，放进计算机执行

By the way, as you can see,

顺便一说，如果程序里有漏洞

they "patched" some bugs in this program

顺便一说，如果程序里有漏洞

by literally putting patches of paper over the holes on the punch tape.

真的就 直接用胶带来补"漏洞"

The Mark 1's instruction set was so primitive,

Mark 1 的指令集非常原始，甚至没有 JUMP 指令

there weren't even JUMP instructions.

Mark 1 的指令集非常原始，甚至没有 JUMP 指令

To create code that repeated the same operation multiple times,

如果代码要跑不止一次

you'd tape the two ends of the punched tape together, creating a physical loop.

得把带子的两端连起来 做成循环

In other words, programming the Mark 1 was kind of a nightmare!

换句话说，给 Mark 1 编程简直是噩梦！

After the war, Hopper continued to work at the forefront of computing.

战后，霍普继续在计算机前沿工作

To unleash the potential of computers,

为了释放电脑的潜力

she designed a high-level programming language called "Arithmetic Language Version 0",

她设计了一个高级编程语言，叫"算术语言版本 0"

or A-0 for short.

简称"A-0"

Assembly languages have direct, one-to-one mapping to machine instructions.

汇编与机器指令是一一对应的

But, a single line of a high-level programming language

但一行高级编程语言 可能会转成几十条二进制指令

might result in dozens of instructions being executed by the CPU.

但一行高级编程语言 可能会转成几十条二进制指令

To perform this complex translation, Hopper built the first compiler in 1952.

为了做到这种复杂转换 \N Hopper 在 1952 年创造了第一个编译器

This is a specialized program

编译器专门把高级语言 转成低级语言

that transforms "source" code written in a programming language into a low-level language,

编译器专门把高级语言 转成低级语言

like assembly or the binary "machine code" that the CPU can directly process.

比如汇编或机器码（CPU 可以直接执行机器码）

Thanks, Thought Bubble.

谢了 思想泡泡

So, despite the promise of easier programming,

尽管"使编程更简单"很诱人

many people were skeptical of Hopper's idea.

但很多人对霍普的点子持怀疑态度

She once said, "I had a running compiler and nobody would touch it.

她曾说"我有能用的编译器，但没人愿意用

they carefully told me, computers could only do arithmetic;

他们告诉我计算机只能做算术,不能运行程序"

they could not do programs."

他们告诉我计算机只能做算术,不能运行程序"

But the idea was a good one,

但这个点子是好的

and soon many efforts were underway to craft new programming languages

不久，很多人尝试创造新编程语言

-- today there are hundreds!

- 如今有上百种语言！

Sadly, there are no surviving examples of A-0 code,

可惜的是，没有任何 A-0 的代码遗留下来

so we'll use Python, a modern programming language, as an example.

所以我们用 Python 举例（一门现代编程语言）

Let's say we want to add two numbers and save that value.

假设我们想相加两个数字，保存结果

Remember, in assembly code,

记住，如果用汇编代码

we had to fetch values from memory, deal with registers, and other low-level details.

我们得从内存取值，和寄存器打交道，以及其他底层细节

But this same program can be written in python like so:

但同样的程序可以用 Python 这样写：

Notice how there are no registers or memory locations to deal with

不用管寄存器或内存位置

-- the compiler takes care of that stuff, abstracting away a lot of low-level and unnecessary complexity.

- 编译器会搞定这些细节，不用管底层细节

The programmer just creates abstractions for needed memory locations, known as variables,

程序员只需要创建 代表内存地址的抽象，叫"变量"

and gives them names.

给变量取名字

So now we can just take our two numbers, store them in variables we give names to

现在可以把两个数 存在变量里

-- in this case, I picked a and b but those variables could be anything -

这里取名 A 和 B, 实际编程时你可以随便取名

and then add those together, saving the result in c, another variable I created.

然后相加两个数，把结果存在变量 C

It might be that the compiler assigns Register A under the hood to store the value in a,

底层操作时，编译器可能把变量 A 存在寄存器 A

but I don't need to know about it!

但我不需要知道这些！

Out of sight, out of mind!

眼不见心不烦

It was an important historical milestone,

这是个重要历史里程碑

but A-0 and its later variants weren't widely used.

但 A-0 和之后的版本没有广泛使用

FORTRAN, derived from "Formula Translation",

FORTRAN，名字来自 "公式翻译"

was released by IBM a few years later, in 1957,

这门语言数年后由 IBM 在 1957 年发布

and came to dominate early computer programming.

主宰了早期计算机编程

John Backus, the FORTRAN project director,

FORTRAN 项目总监 John Backus 说过

said: "Much of my work has come from being lazy.

"我做的大部分工作都是因为懒

I didn't like writing programs,

我不喜欢写程序

and so ... I started work on a programming system to make it easier to write programs."

所以我写这门语言，让编程更容易"

You know, typical lazy person.

你懂的，典型的"懒人"

They're always creating their own programming systems.

（白眼）创造自己的编程语言

Anyway, on average, programs written in FORTRAN

平均来说，FORTRAN 写的程序

were 20 times shorter than equivalent handwritten assembly code.

比等同的手写汇编代码短 20 倍

Then the FORTRAN Compiler would translate and expand that into native machine code.

然后 FORTRAN 编译器会把代码转成机器码

The community was skeptical that the performance would be as good as hand written code,

人们怀疑性能是否比得上手写代码

but the fact that programmers could write more code more quickly,

但因为能让程序员写程序更快，所以成了一个更经济的选择

made it an easy choice economically:

但因为能让程序员写程序更快，所以成了一个更经济的选择

trading a small increase in computation time for a significant decrease in programmer time.

运行速度慢一点点，编程速度大大加快

Of course, IBM was in the business of selling computers,

当时 IBM 在卖计算机

and so initially, FORTRAN code could only be compiled and run on IBM computers.

因此最初 FORTRAN 代码只能跑在 IBM 计算机上

And most programing languages and compilers of the 1950s

1950 年代大多数编程语言和编译器

could only run on a single type of computer.

只能运行在一种计算机上

So, if you upgraded your computer,

如果升级电脑

you'd often have to re-write all the code too!

可能要重写所有代码！

In response, computer experts from industry,

因此工业界，学术界，政府的计算机专家 \N 在 1959 年组建了一个联盟

academia and government formed a consortium in 1959

因此工业界，学术界，政府的计算机专家 \N 在 1959 年组建了一个联盟

-- the Committee on Data Systems Languages, advised by our friend Grace Hopper --

- 数据系统语言委员会，Grace Hopper 担任顾问

to guide the development of a common programming language

开发一种通用编程语言，可以在不同机器上通用

that could be used across different machines.

开发一种通用编程语言，可以在不同机器上通用

The result was the high-level, easy to use,

最后诞生了一门高级，易于使用，

Common Business-Oriented Language, or COBOL for short.

"普通面向商业语言"，简称 COBOL

To deal with different underlying hardware,

为了兼容不同底层硬件

each computing architecture needed its own COBOL compiler.

每个计算架构需要一个 COBOL 编译器

But critically, these compilers could all accept the same COBOL source code,

最重​​要的是，这些编译器都可以接收相同 COBOL 代码

no matter what computer it was run on.

不管是什么电脑

This notion is called write once, run anywhere.

这叫"一次编写，到处运行"

It's true of most programming languages today,

如今大多数编程语言都是这样

a benefit of moving away from assembly and machine code,

不必接触 CPU 特有的汇编码和机器码

which is still CPU specific.

不必接触 CPU 特有的汇编码和机器码

The biggest impact of all this was reducing computing's barrier to entry.

减小了使用门槛

Before high level programming languages existed,

在高级编程语言出现之前

it was a realm exclusive to computer experts and enthusiasts.

编程只是计算机专家和爱好者才会做的事

And it was often their full time profession.

而且通常是主职

But now, scientists, engineers, doctors, economists, teachers,

但现在，科学家，工程师，医生，经济学家，教师

and many others could incorporate computation into their work .

等等，都可以把计算机用于工作

Thanks to these languages,

感谢这些语言

computing went from a cumbersome and esoteric discipline

计算机科学从深奥学科 变成了大众化工具

to a general purpose and accessible tool.

计算机科学从深奥学科 变成了大众化工具

At the same time, abstraction in programming allowed those computer experts

同时，编程的抽象也让计算机专家

- now "professional programmers" -

现在叫"专业程序员"

to create increasingly sophisticated programs,

制作更复杂的程序

which would have taken millions, tens of millions, or even more lines of assembly code.

如果用汇编写可能要上百万行

Now, this history didn't end in 1959.

当然，计算机的历史没有在 1959 年结束

In fact, a golden era in programming language design jump started,

编程语言设计的黄金时代才刚刚开始

evolving in lockstep with dramatic advances in computer hardware.

和硬件一起飞速发展

In the 1960s, we had languages like ALGOL, LISP and BASIC.

在 1960 年代，有 ALGOL, LISP 和 BASIC 等语言

In the 70's: Pascal, C and Smalltalk were released.

70年代有：Pascal，C 和 Smalltalk

The 80s gave us C++, Objective-C, and Perl.

80年代有：C++，Objective-C 和 Perl

And the 90's: python, ruby, and Java.

90年代有：Python，Ruby 和 Java

And the new millennium has seen the rise of Swift, C#, and Go

新千年 Swift, C#, Go 在崛起

- not to be confused with Let it Go and Pokemon Go.

不要把 Go 和\N 《冰雪奇缘》的 Let it Go 和游戏 Pokemon Go 弄混

Anyway, some of these might sound familiar

有些语言你可能听起来耳熟 - 很多现在还存在

-- many are still around today.

有些语言你可能听起来耳熟 - 很多现在还存在

It's extremely likely that the web browser you're using right now

你现在用的浏览器很可能是 C++ 或 Objective-C 写的

was written in C++ or Objective-C.

你现在用的浏览器很可能是 C++ 或 Objective-C 写的

That list I just gave is the tip of the iceberg.

我刚才说的编程语言名字 只是冰山一角

And languages with fancy, new features are proposed all the time.

新的编程语言在不断诞生

Each new language attempts to leverage new and clever abstractions

新语言想用更聪明的抽象

to make some aspect of programming easier or more powerful,

让某些方面更容易或更强大

or take advantage of emerging technologies and platforms,

或利用新技术和新平台带来的优势

so that more people can do more amazing things, more quickly.

让更多人能快速做出美妙的事情

Many consider the holy grail of programming to be the use of "plain ol' English",

许多人认为编程的"圣杯"是直接用英文

where you can literally just speak what you want the computer to do,

直接对计算机说话，然后它会理解并执行

it figures it out, and executes it.

直接对计算机说话，然后它会理解并执行

This kind of intelligent system is science fiction for now.

这种智能系统目前只存在于科幻小说

And fans of 2001: A Space Odyssey may be okay with that.

"2001：太空漫游" 的粉丝可能没什么意见

Now that you know all about programming languages,

现在你理解了编程语言,

we're going to deep dive for the next couple of episodes,

接下来几集 我们会深入了解

and we'll continue to build your understanding

接下来几集 我们会深入了解

of how programming languages, and the software they create,

编程语言和用语言写的软件

are used to do cool and unbelievable things.

是怎么做到那些酷事

See you next week.

下周见

（给 Curiosity Stream 打广告）

Hi, I’m Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Last episode we discussed how writing programs in native machine code,

上集讲到用机器码写程序,

and having to contend with so many low level details, was a huge impediment to writing complex programs.

还要处理那么多底层细节 对写大型程序是个巨大障碍

To abstract away many of these low-level details, Programming Languages were developed that

为了脱离底层细节，开发了编程语言

let programmers concentrate on solving a problem with computation, and less on nitty gritty hardware details.

让程序员专心解决问题，不用管硬件细节

So today, we’re going to continue that discussion, and introduce some fundamental building blocks

今天我们讨论 大多数编程语言都有的基本元素

that almost all programming languages provide.

今天我们讨论 大多数编程语言都有的基本元素

Just like spoken languages, programming languages have statements.

就像口语一样，编程语言有"语句"

These are individual complete thoughts, like "I want tea" or "it is raining".

语句表达单个完整思想，比如"我想要茶"或者"在下雨"

By using different words, we can change the meaning;

用不同词汇可以代表不同含义 \N 比如"我想要茶"变成"我想要独角兽"

for example, "I want tea" to "I want unicorns".

用不同词汇可以代表不同含义 \N 比如"我想要茶"变成"我想要独角兽"

But we can’t change "I want tea" to "I want raining" - that doesn’t make grammatical sense.

但没法把"我想要茶"改成"我想要雨"- 语法毫无意义

The set of rules that govern the structure and composition of statements in a language

规定句子结构的一系列规则 叫语法

is called syntax.

规定句子结构的一系列规则 叫语法

The English language has syntax, and so do all programming languages.

英语有语法，所有编程语言也都有语法

"a = 5" is a programming language statement.

a=5 是一个编程语言语句

In this case, the statement says a variable named A has the number 5 stored in it.

意思是创建一个叫 a 的变量，把数字 5 放里面.

This is called an assignment statement because we're assigning a value to a variable.

这叫"赋值语句"，把一个值赋给一个变量

To express more complex things, we need a series of statements,

为了表达更复杂的含义，需要更多语句

like "A is 5, B is 10, C equals A plus B"

比如 \Na=5 \N b=10 \Nc=a+b

This program tells the computer to set variable ‘A’ equal to 5, variable ‘B’ to 10,

意思是，变量 a 设为5，变量 b 设为10

and finally to add ‘A’ and ‘B’ together, and put that result, which is 15, into -- you guessed it -- variable C.

把 a 和 b 加起来，把结果 15 放进变量 c

Note that we can call variables whatever we want.

注意，变量名可以随意取

Instead of A, B and C, it could be apples, pears, and fruits.

除了 a b c，也可以叫苹果、梨、水果

The computer doesn’t care, as long as variables are uniquely named.

计算机不在乎你取什么名，只要不重名就行

But it’s probably best practice to name them things that make sense

当然取名最好还是有点意义，方便别人读懂

in case someone else is trying to understand your code.

当然取名最好还是有点意义，方便别人读懂

A program, which is a list of instructions, is a bit like a recipe:

程序由一个个指令组成，有点像菜谱：

boil water, add noodles, wait 10 minutes, drain and enjoy.

烧水、加面，等10分钟，捞出来就可以吃了

In the same way, the program starts at the first statement

程序也是这样，从第一条语句开始

and runs down one at a time until it hits the end.

一句一句运行到结尾

So far, we’ve added two numbers together.

刚才我们只是把两个数字加在一起

Boring.

无聊

Let’s make a video game instead!

我们来做一款游戏吧

Of course, it’s way too early to think about coding an entire game,

当然，现在这个学习阶段 \N来编写一整个游戏还太早了

so instead, we’ll use our example to write little snippets of code

所以我们只写一小段一小段的代码

that cover some programming fundamentals.

来讲解一些基础知识

Imagine we’re building an old-school arcade game where Grace Hopper has to capture bugs

假设我们在写一款老派街机游戏：Grace Hopper 拍虫子

before they get into the Harvard Mark 1 and crash the computer!

阻止虫子飞进计算机造成故障

On every level, the number of bugs increases.

关卡越高，虫子越多

Grace has to catch them before they wear out any relays in the machine.

Grace 要在虫子损坏继电器之前 抓住虫子

Fortunately, she has a few extra relays for repairs.

好消息是 她有几个备用继电器

To get started, we’ll need to keep track of a bunch of values that are important for gameplay

开始编写时，我们需要一些值 来保存游戏数据

like what level the player is on, the score, the number of bugs remaining,

比如当前关卡数、分数、剩余虫子数、

as well as the number of spare relays in Grace’s inventory.

Grace 还剩几个备用继电器

So, we must "initialize" our variables, that is, set their initial value:

所以我们要"初始化"变量 \N "初始化"的意思是设置最开始的值.

"level equals 1, score equals 0, bugs equals 5, spare relays equals 4, and player name equals "Andre".

关卡=1 分数=0 虫子数=5 \N 备用继电器=4 玩家名=Andre

To create an interactive game, we need to control the flow of the program

为了做成交互式游戏，程序的执行顺序要更灵活

beyond just running from top to bottom.

不只是从上到下执行

To do this, we use Control Flow Statements.

因此用 "控制流语句"

There are several types, but If Statements are the most common.

控制流语句有好几种，最常见的是 if 语句

You can think of them as "If X is true, then do Y".

可以想成是 "如果 X 为真，那么执行 Y"

An English language example is: "If I am tired, then get tea"

用英语举例就是 "如果累了, 就去喝茶"

So if "I am tired" is a true statement, then I will go get tea

如果 "累了" 为真，就去喝茶

If "I am tired" is false, then I will not go get tea.

如果 "累了" 为假，就不喝茶

An IF statement is like a fork in the road.

if 语句就像岔路口

Which path you take is conditional on whether the expression is true or false

走哪条路 取决于 "表达式" 的真假，

so these expressions are called Conditional Statements.

因此这些表达式又叫 "条件语句"

In most programming languages, an if statement looks something like

在大多数编程语言中，if 语句看起来像这样：

"If, expression, then, some code, then end the if statement".

if [条件], then [一些代码]，结束 if 语句.

For example, if "level" is 1, then we set the score to zero, because the player is just starting.

比如，if [第一关]，then [分数设为0] \N 因为玩家才刚开始游戏

We also set the number of bugs to 1, to keep it easy for now.

同时把虫子数设为 1，让游戏简单些

Notice the lines of code that are conditional on the if-statement are nested between the

注意, 依赖于 if 条件的代码，要放在 IF 和 END IF 之间

IF and END IF.

注意, 依赖于 if 条件的代码，要放在 IF 和 END IF 之间

Of course, we can change the conditional expression to whatever we want to test, like

当然，条件表达式 可以改成别的，比如：

"is score greater than 10" or "is bugs less than 1".

"分数 >10" 或者 "虫子数 <1"

And If-Statements can be combined with an ELSE statement, which acts as a catch-all if the expression is false.

if 还可以和 else 结合使用 \N 条件为假会执行 else 里的代码

If the level is not 1, the code inside the ELSE block will be executed instead, and the

如果不是第1关，else 里的指令就会被执行

number of bugs that Grace has to battle is set to 3 times the level number.

Grace 要抓的虫子数，是当前关卡数 \* 3

So on level 2, it would be six bugs, and on level 3 there’s 9, and so on.

所以第 2 关有 6 个虫子，第 3 关有 9 个虫子，以此类推

Score isn’t modified in the ELSE block, so Grace gets to keep any points earned.

else 中没有改分数，所以 Grace 的分数不会变

Here are some examples of if-then-else statements from some popular programming languages

这里列了一些热门编程语言 if-then-else 的具体语法

-- you can see the syntax varies a little, but the underlying structure is roughly the same.

具体语法略有不同，但主体结构一样

If-statements are executed once, a conditional path is chosen, and the program moves on.

if 语句 根据条件执行一次

To repeat some statements many times, we need to create a conditional loop.

如果希望根据条件执行多次，需要"条件循环"

One way is a while statement, also called a while loop.

比如 while 语句，也叫 "while 循环"

As you might have guessed, this loops a piece of code "while" a condition is true.

当 while 条件为真，代码会重复执行

Regardless of the programming language, they look something like this:

不管是哪种编程语言，结构都是这样

In our game, let’s say at certain points, a friendly colleague restocks Grace with relays!

假设到达一定分数会冒出一个同事，给 Grace 补充继电器

Hooray!

棒极了！

To animate him replenishing our stock back up to a maximum of 4, we can use a while loop.

把继电器补满到最大数 4 个 \N 我们可以用 while 语句来做

Let’s walk through this code.

来过一遍代码

First we’ll assume that Grace only has 1 tube left when her colleague enters.

假设同事入场时， Grace 只剩一个继电器

When we enter the while loop, the first thing the computer does is test its conditional…

当执行 while 循环，第一件事是检查条件

is relays less than 4?

继电器数量<4?

Well, relays is currently 1, so yes.

继电器数量现在是1，所以是真

Now we enter the loop!

进入循环！

Then, we hit the line of code: "relays equals relays plus 1".

碰到这一行：继电器数量=继电器数量+1

This is a bit confusing because the variable is using itself in an assignment statement,

看起来有点怪，变量的赋值用到了自己

so let's unpack it.

我们讲下这个

You always start by figuring out the right side of the equals sign first,

总是从等号右边开始，

so what does "relays plus 1" come out to be?

"继电器数量+1" 是多少？

Well, relays is currently the value 1, so 1 plus 1 equals 2.

当前值是1，所以 1+1=2

Then, this result gets saved back into the variable relays, writing over the old value,

结果存到"继电器数量"，覆盖旧的值

so now relays stores the value 2.

所以现在继电器数量是 2

We’ve hit the end of the while loop, which jumps the program back up.

现在到了结尾，跳回开始点

Just as before, we test the conditional to see if we’re going to enter the loop.

和之前一样，先判断条件，看要不要进入循环

Is relays less than 4?

继电器数量<4？

Well, yes, relays now equals 2, so we enter the loop again!

是，继电器数量是2，所以再次进入循环!

2 plus 1 equals 3.

2+1=3

so 3 is saved into relays.

3 存入"继电器数量"

Loop again.

回到开头

Is 3 less than 4?

3<4？

Yes it is!

是!

Into the loop again.

进入循环

3 plus 1 equals 4.

3+1=4

So we save 4 into relays.

4 存入"继电器数量"

Loop again.

回到开头

Is 4 less than 4?....

4<4?

No!

不！

So the condition is now false, and thus we exit the loop and move on to any remaining code

现在条件为假，退出循环，执行后面的代码

That’s how a while loop works!

while 循环就是这样运作的!

There’s also the common For Loop.

另一种常见的叫 "for 循环"

Instead of being a condition-controlled loop that can repeat forever until the condition is false

不判断条件，判断次数，会循环特定次数

a FOR loop is count-controlled; it repeats a specific number of times.

不判断条件，判断次数，会循环特定次数

They look something like this:

看起来像上图

Now, let’s put in some real values.

现在放些真正的值进去

This example loops 10 times, because we’ve specified that variable ‘i’

上图例子会循环10次，因为设了变量 i

starts at the value 1 and goes up to 10.

从 1 开始，一直到 10

The unique thing about a FOR loop is that each time it hits NEXT, it adds one to ‘i’.

for 的特点是，每次结束， i 会 +1

When ‘i’ equals 10, the computer knows it’s been looped 10 times, and the loop exits

当 i 等于10，就知道循环了10次，然后退出.

We can set the number to whatever we want -- 10, 42, or a billion -- it’s up to us.

我们可以用任何数字，10, 42, 10 亿

Let’s say we want to give the player a bonus at the end of each level

假设每关结束后 给玩家一些奖励分

for the number of vacuum relays they have left over.

奖励分多少取决于 继电器剩余数量

As the game gets harder, it takes more skill to have unused relays,

随着难度增加，剩下继电器会越来越难

so we want the bonus to go up exponentially based on the level.

因此奖励分会根据当前关卡数，指数级增长

We need to write a piece of code that calculates exponents -

我们要写一小段代码来算指数

that is, multiplying a number by itself a specific number of times.

指数是一个数乘自己，乘特定次数

A loop is perfect for this!

用循环来实现简直完美!

First lets initialize a new variable called "bonus" and set it to 1.

首先，创建一个叫"奖励分"的新变量，设为 1 （看上图）

Then, we create a FOR loop starting at 1, and looping up to the level number.

然后 for 循环，从 1 到 [当前关卡数]

Inside that loop, we multiply bonus times the number of relays,

[奖励分] x [继电器剩余数]，结果存入 [奖励分]

and save that new value back into bonus.

[奖励分] x [继电器剩余数]，结果存入 [奖励分]

For example, let’s say relays equals 2, and level equals 3.

比如继电器数是2，关卡数是3

So the FOR loop will loop three times, which means bonus is going to get multiplied by

for 会循环3次，奖励分会乘

relays... by relays... by relays.

继电器数量 x 继电器数量 x 继电器数量

Or in this case, times 2, times 2, times 2, which is a bonus of 8!

也就是1×2×2×2，奖励分是8，2的3次方

That’s 2 to the 3rd power!

也就是1×2×2×2，奖励分是8，2的3次方

This exponent code is useful, and we might want to use it in other parts of our code.

这个指数代码很实用，其他地方可能会用到

It’d be annoying to copy and paste this everywhere, and have to update the variable names each time.

如果每次想用就复制粘贴，会很麻烦，每次都要改变量名

Also, if we found a bug, we’d have to hunt around and update every place we used it.

如果代码发现问题，要补漏洞时 \N 要把每一个复制黏贴过的地方都找出来改

It also makes code more confusing to look at.

而且会让代码更难懂

Less is more!

少即是多！

What we want is a way to package up our exponent code so we can use it, get the result, and

我们想要某种方法，把代码"打包" \N 可以直接使用，得出结果，

not have to see all the internal complexity.

不用管内部复杂度.

We’re once again moving up a new level of abstraction!

这又提升了一层抽象！

To compartmentalize and hide complexity,

为了隐藏复杂度

programming languages can package pieces of code into named functions,

可以把代码打包成 "函数"

also called methods or subroutines in different programming languages.

也叫 "方法" 或 "子程序"\N（有些编程语言这么叫）

These functions can then be used by any other part of that program just by calling its name.

其他地方想用这个函数，直接写函数名就可以了

Let’s turn our exponent code into a function! First, we should name it.

现在我们把指数代码变成函数. 第一步，取名.

We can call it anything we want, like HappyUnicorn,

叫什么都行，比如"快乐独角兽"

but since our code calculates exponents, let’s call it exponent.

但因为是算指数, 直接叫"指数"合适一些

Also, instead of using specific variable names, like "relays" and "levels",

还有，与其用特定变量名，比如 "继电器" 和 "关卡数"

we specify generic variable names, like Base and Exp,

用更通用的名字，比如 底数(Base) 和 指数(Exp)

whose initial values are going to be "passed" into our function from some other part of the program.

Base 和 Exp 的初始值需要外部传入

The rest of our code is the same as before

剩余代码和之前一样

Now tucked into our function and with new variable names.

现在完成了，有函数名和新变量名.

Finally, we need to send the result of our exponent code back to the part of the program that requested it.

最后, 我们还需要把结果 交给使用这个函数的代码

For this, we use a RETURN statement, and specify that the value in ‘result’ be returned.

所以用 RETURN 语句，指明返回什么.

So our full function code looks like this:

完整版代码是这样

Now we can use this function anywhere in our program,

现在可以随意用这个函数

simply by calling its name and passing in two numbers.

只需要写出名字 然后传入2个数字 就可以了

For example, if we want to calculate 2 to the 44th power, we can just call "exponent 2 comma 44."

如果要算 2 的 44 次方，写 exponent(2,44)

and like 18 trillion comes back.

结果是 18 万亿左右

Behind the scenes, 2 and 44 get saved into variables Base and Exp inside the function,

幕后原理是，2 和 44 存进 Base 和 Exp

it does all its loops as necessary, and then the function returns with the result.

跑循环，然后返回结果

Let’s use our newly minted function to calculate a score bonus.

我们来用这个新函数 算奖励分

First, we initialize bonus to 0.

首先，奖励分初始化为 0

Then we check if the player has any remaining relays with an if-statement.

然后用 if 语句，看剩不剩继电器（看上图的 > 0）

If they do, we call our exponent function, passing in relays and level,

如果还剩，用指数函数，传入 [继电器数] 和 [关卡数]

which calculates relays to the power of level, and returns the result, which we save into bonus.

它会算 [继电器数]的[关卡数]次方, 存入奖励分

This bonus calculating code might be useful later, so let’s wrap it up as a function too!

这段算奖励分的代码，之后可能还会用，也打包成一个函数

Yes, a function that calls a function!

没错，这个函数 (CalcBonus) \N 会调用另一个函数 (Exponent)

And then, wait for it…. we can use this function in an even more complex function.

还有！这个 CalcBonus 函数，可以用在其他更复杂的函数

Let’s write one that gets called everytime the player finishes a level.

我们来写一个函数, 每一关结束后都会调用

We’ll call it "LevelFinished"

叫 LevelFinished (关卡结束)

- it needs to know the number of relays left, what level it was, and the current score;

需要传入 [剩余继电器数] [关卡数] [当前分]

those values have to get passed in.

这些数据必须传入.

Inside our function, we’ll calculate the bonus, using our CalcBonus function,

里面用 CalcBonus 算奖励分，并加进总分

and add that to the running score.

里面用 CalcBonus 算奖励分，并加进总分

Also, if the current score is higher than the game’s high score,

还有，如果当前分 > 游戏最高分

we save the new high score and the players name.

把新高分和玩家名 存起来

Now we’re getting pretty fancy.

现在代码变得蛮"花哨"了

Functions are calling functions are calling functions!

函数调函数调函数

When we call a single line of code, like this the complexity is hidden.

我们写这样一行代码时，复杂度都隐藏起来了

We don’t see all the internal loops and variables,

不需要知道内部的循环和变量

we just see the result come back as if by magic…. a total score of 53.

只知道结果会像魔术一样返回，总分 53

But it’s not magic, it’s the power of abstraction!

但是这不是魔术，是抽象的力量

If you understand this example, then you understand the power of functions,

如果你理解了这个例子，就明白了函数的强大之处

and the entire essence of modern programming.

和现代编程的核心

It’s not feasible to write, for example, a web browser as one gigantically long list of statements.

比如浏览器这样的复杂程序，用一长串语句来写是不可能的

It would be millions of lines long and impossible to comprehend!

会有几百万行代码，没人能理解

So instead, software consists of thousands of smaller functions,

所以现代软件由上千个函数组成

each responsible for different features.

每个负责不同的事

In modern programming, it’s uncommon to see functions longer than around 100 lines of code

如今超过100行代码的函数很少见

because by then, there’s probably something that

如果多于 100 行，应该有东西可以拆出来做成一个函数

should be pulled out and made into its own function.

如果多于 100 行，应该有东西可以拆出来做成一个函数

Modularizing programs into functions not only allows a single programmer to write an entire app

模块化编程 不仅可以让单个程序员独立制作 App

but also allows teams of people to work efficiently on even bigger programs.

也让团队协作可以写更大型的程序

Different programmers can work on different functions,

不同程序员写不同函数

and if everyone makes sure their code works correctly,

只需要确保自己的代码工作正常

then when everything is put together, the whole program should work too!

把所有人的拼起来，整个程序也应该能正常运作！

And in the real world, programmers aren’t wasting time writing things like exponents.

现实中，程序员不会浪费时间写指数函数这种东西

Modern programming languages come with huge bundles of pre-written functions, called Libraries.

现代编程语言 有很多预先写好的函数集合，叫 "库"

These are written by expert coders, made efficient and rigorously tested, and then given to everyone.

由专业人员编写，不仅效率高，而且经过了仔细检查

There are libraries for almost everything, including networking, graphics, and sound

几乎做所有事情都有库，网络、图像、声音

-- topics we’ll discuss in future episodes.

我们之后会讲这些主题.

But before we get to those, we need to talk about Algorithms.

但在此之前，我们先讲算法

Intrigued?

好奇吗？

You should be.

你应该才是！

I’ll see you next week.

下周见

Hi, I'm Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Over the past two episodes, we got our first taste of programming in a high-level language,

前两集，我们"初尝"了高级编程语言\N （比如 Python 和 Java）

like Python or Java.

前两集，我们"初尝"了高级编程语言\N （比如 Python 和 Java）

We talked about different types of programming language statements

我们讨论了几种语句 - 赋值语句，if 语句，循环语句

- like assignments, ifs, and loops -

我们讨论了几种语句 - 赋值语句，if 语句，循环语句

as well as putting statements into functions that perform a computation,

以及把代码打包成 "函数"

like calculating an exponent.

比如算指数

Importantly, the function we wrote to calculate exponents is only one possible solution.

重要的是，之前写的指数函数 \N 只是无数解决方案的一种

There are other ways to write this function

还有其它方案

- using different statements in different orders -

- 用不同顺序写不同语句 也能得到一样结果

that achieve exactly the same numerical result.

- 用不同顺序写不同语句 也能得到一样结果

The difference between them is the algorithm,

不同的是 "算法"，意思是：解决问题的具体步骤

that is the specific steps used to complete the computation.

不同的是 "算法"，意思是：解决问题的具体步骤

Some algorithms are better than others even if they produce equal results.

即使结果一致，有些算法会更好

Generally, the fewer steps it takes to compute, the better it is,

一般来说，所需步骤越少越好

though sometimes we care about other factors, like how much memory it uses.

不过有时我们也会关心其他因素，比如占多少内存

The term algorithm comes from Persian polymath Muhammad ibn Musa al-Khwarizmi

"算法" 一词来自 波斯博识者 阿尔·花拉子密

who was one of the fathers of algebra more than a millennium ago.

1000 多年前的代数之父之一

The crafting of efficient algorithms

如何想出高效算法 - 是早在计算机出现前就有的问题

- a problem that existed long before modern computers -

如何想出高效算法 - 是早在计算机出现前就有的问题

led to a whole science surrounding computation,

诞生了专门研究计算的领域，然后发展成一门现代学科

which evolved into the modern discipline of...

诞生了专门研究计算的领域，然后发展成一门现代学科

you guessed it!

你猜对了！

Computer Science!

计算机科学！

One of the most storied algorithmic problems in all of computer science is sorting

记载最多的算法之一是"排序"

as in sorting names or sorting numbers.

比如给名字、数字排序

Computers sort all the time.

排序到处都是

Looking for the cheapest airfare,

找最便宜的机票

arranging your email by most recently sent,

按最新时间排邮件

or scrolling your contacts by last name

按姓氏排联系人

-- those all require sorting.

-这些都要排序

You might think

你可能想"排序看起来不怎么难… 能有几种算法呢?"

"sorting isn't so tough how many algorithms can there possibly be?"

你可能想"排序看起来不怎么难… 能有几种算法呢?"

The answer is: a lot.

答案是超多

Computer Scientists have spent decades inventing algorithms for sorting,

计算机科学家花了数十年发明各种排序算法

with cool names like Bubble Sort and Spaghetti Sort.

还起了酷酷的名字，"冒泡排序""意面排序"

Let's try sorting!

我们来试试排序！

Imagine we have a set of airfare prices to Indianapolis.

试想有一堆机票价格，都飞往 印第安纳波利斯 (美国地名)

We'll talk about how data like this is represented in memory next week,

数据具体怎么在内存中表示 下周再说

but for now, a series of items like this is called an array.

上图的这样一组数据 叫"数组"（Array）

Let's take a look at these numbers to help see how we might sort this programmatically.

来看看怎么排序

We'll start with a simple algorithm.

先从一种简单算法开始

First, let's scan down the array to find the smallest number.

先找到最小数，从最上面的 307 开始

Starting at the top with 307.

先找到最小数，从最上面的 307 开始

It's the only number we've seen, so it's also the smallest.

因为现在只看了这一个，所以它是最小数

The next is 239, that's smaller than 307,

下一个是 239，比 307 小

so it becomes our new smallest number.

所以新的最小数变成 239

Next is 214, our new smallest number.

下一个是 214 ，新的最小数

250 is not, neither is 384, 299, 223 or 312.

250 不是，384, 299, 223, 312 都不是

So we've finished scanning all numbers,

现在扫完了所有数字

and 214 is the smallest.

214 是最小的

To put this into ascending order,

为了升序排列（从小到大排序）

we swap 214 with the number in the top location.

把 214 和最上面的数字，交换位置

Great! We sorted one number!

好棒! 刚排序了一个数字！

Now we repeat the same procedure,

现在重复同样的过程

but instead of starting at the top, we can start one spot below.

这次不从最上面开始，从第 2 个数开始

First we see 239, which we save as our new smallest number.

先看到 239，我们当作是 "最小数"

Scanning the rest of the array, we find 223 is the next smallest,

扫描剩下的部分，发现 223 最小

so we swap this with the number in the second spot.

所以把它和第 2 位交换

Now we repeat again, starting from the third number down.

重复这个过程，从第 3 位数字开始

This time, we swap 239 with 307.

让 239 和 307 互换位置

This process continues until we get to the very last number,

重复直到最后一个数字

and voila, the array is sorted and you're ready to book that flight to Indianapolis!

瞧，数字排好了，可以买机票了！

The process we just walked through is one way

刚刚这种方法，或者说算法，

- or one algorithm - for sorting an array.

刚刚这种方法，或者说算法，

It's called Selection sort - and it's pretty basic.

叫 选择排序 - 非常基础的一种算法

Here's the pseudo-code.

以下是"伪代码"

This function can be used to sort 8, 80, or 80 million numbers

这个函数可以排序8个, 80个或8千万个数字

- and once you've written the function, you can use it over and over again.

函数写好了就可以重复使用

With this sort algorithm, we loop through each position in the array, from top to bottom,

这里用循环 遍历数组

and then for each of those positions,

每个数组位置都跑一遍循环，找最小数然后互换位置

we have to loop through the array to find the smallest number to swap.

每个数组位置都跑一遍循环，找最小数然后互换位置

You can see this in the code, where one FOR loop is nested inside of another FOR loop.

可以在代码中看到这一点 \N （一个 for 循环套另一个 for 循环）

This means, very roughly, that if we want to sort N items, we have to loop N times,

这意味着，大致来说，如果要排 N 个东西，要循环 N 次，

inside of which, we loop N times, for a grand total of roughly N times N loops, or N squared.

每次循环中再循环 N 次，共 N\*N, 或 N

This relationship of input size to the number of steps the algorithm takes to run

算法的 输入大小 和 运行步骤 之间的关系

characterizes the complexity of the Selection Sort algorithm.

叫算法的 复杂度

It gives you an approximation of how fast, or slow, an algorithm is going to be.

表示运行速度的量级

Computer Scientists write this order of growth in something known as - no joke -

计算机科学家们把算法复杂度叫 - 没开玩笑

"big O notation".

大 O 表示法

N squared is not particularly efficient.

算法复杂度 O(N ) 效率不高

Our example array had n = 8 items, and 8 squared is 64.

前面的例子有 8 个元素（n=8）, 8 = 64

If we increase the size of our array from 8 items to 80,

如果 8 个变 80 个

the running time is now 80 squared, which is 6,400.

运行时间变成 80 = 6400

So although our array only grew by 10 times - from 8 to 80 -

虽然大小只增长了 10 倍（8 到 80）

the running time increased by 100 times - from 64 to 6,400!

但运行时间增加了 100 倍！（64 到 6400 ）

This effect magnifies as the array gets larger.

随着数组增大，对效率的影响会越来越大

That's a big problem for a company like Google,

这对大公司来说是个问题，比如 谷歌

which has to sort arrays with millions or billions of entries.

要对几十亿条信息排序

So, you might ask,

作为未来的计算机科学家你可能会问：有没有更高效的排序算法？

as a burgeoning computer scientist, is there a more efficient sorting algorithm?

作为未来的计算机科学家你可能会问：有没有更高效的排序算法？

Let's go back to our old, unsorted array

回到未排序的数组

and try a different algorithm, merge sort.

试另一个算法 "归并排序"

The first thing merge sort does is check if the size of the array is greater than 1.

第一件事是检查数组大小是否 > 1

If it is, it splits the array into two halves.

如果是，就把数组分成两半

Since our array is size 8, it gets split into two arrays of size 4.

因为数组大小是 8，所以分成两个数组，大小是 4

These are still bigger than size 1, so they get split again, into arrays of size 2,

但依然大于 1，所以再分成大小是 2 的数组

and finally they split into 8 arrays with 1 item in each.

最后变成 8 个数组，每个大小为 1

Now we are ready to merge, which is how "merge sort" gets its name.

现在可以"归并"了，"归并排序"因此得名

Starting with the first two arrays, we read the first - and only - value in them,

从前两个数组开始，读第一个（也是唯一一个）值

in this case, 307 and 239.

307 和 239

239 is smaller, so we take that value first.

239 更小，所以放前面

The only number left is 307, so we put that value second.

剩下的唯一数字是 307 ，所以放第二位

We've successfully merged two arrays.

成功合并了两个数组

We now repeat this process for the remaining pairs, putting them each in sorted order.

重复这个过程，按序排列

Then the merge process repeats.

然后再归并一次

Again, we take the first two arrays, and we compare the first numbers in them.

同样，取前两个数组，比较第一个数

This time its 239 and 214.

239 和 214

214 is lowest, so we take that number first.

214 更小，放前面

Now we look again at the first two numbers in both arrays: 239 and 250.

再看两个数组里的第一个数：239 和 250

239 is lower, so we take that number next.

239 更小，所以放下一位

Now we look at the next two numbers: 307 and 250.

看剩下两个数：307 和 250

250 is lower, so we take that.

250 更小，所以放下一位

Finally, we're left with just 307, so that gets added last.

最后剩下 307 ，所以放最后

In every case, we start with two arrays,

每次都以 2 个数组开始

each individually sorted, and merge them into a larger sorted array.

然后合并成更大的有序数组

We repeat the exact same merging process for the two remaining arrays of size two.

我们把刚隐藏起来的，下面的数组也这样做

Now we have two sorted arrays of size 4.

现在有两个大小是 4 的有序数组

Just as before, we merge,

就像之前，比较两个数组的第一个数，取最小数

comparing the first two numbers in each array, and taking the lowest.

就像之前，比较两个数组的第一个数，取最小数

We repeat this until all the numbers are merged,

重复这个过程，直到完成

and then our array is fully sorted again!

就排好了！

The bad news is: no matter how many times we sort these,

但坏消息是：无论排多少次

you're still going to have to pay $214 to get to Indianapolis.

你还是得付 214 美元到 印第安纳波利斯

Anyway, the "Big O" computational complexity of merge sort is N times the Log of N.

总之，"归并排序"的算法复杂度是 O(n \* log n)

The N comes from the number of times we need to compare and merge items,

n 是需要 比较+合并 的次数

which is directly proportional to the number of items in the array.

和数组大小成正比

The Log N comes from the number of merge steps.

log N 是合并步骤的次数

In our example, we broke our array of 8 items into 4,

例子中把大小是 8 的数组，分成四个数组

then 2, and finally 1.

然后分成 2 个，最后分成 1 个

That's 3 splits.

分了 3 次

Splitting in half repeatedly like this has a logarithmic relationship with the number of items

重复切成两半，和数量成对数关系

- trust me!

相信我！

Log base 2 of 8 equals 3 splits.

Log 8=3

If we double the size of our array to 16 - that's twice as many items to sort -

如果数组大小变成 16 - 之前的两倍

it only increases the number of split steps by 1

也只要多分割 1 次

since log base 2 of 16 equals 4.

因为 Log 16=4

Even if we increase the size of the array more than a thousand times,

即使扩大一千倍

from 8 items to 8000 items, the number of split steps stays pretty low.

从8到8000，分割次数也不会增大多少

Log base 2 of 8000 is roughly 13.

log 8000≈13

That's more, but not much more than 3 -- about four times larger --

13 比 3 只是4倍多一点

and yet we're sorting a lot more numbers.

然而排序的元素多得多

For this reason, merge sort is much more efficient than selection sort.

因此"归并排序"比"选择排序"更有效率

And now I can put my ceramic cat collection in name order MUCH faster!

这下我收藏的陶瓷猫 可以更快排序了！

There are literally dozens of sorting algorithms we could review,

有好几十种排序算法，但没时间讲

but instead, I want to move on to my other favorite category of classic algorithmic problems:

所以我们来谈一个经典算法问题：

graph search!

图搜索

A graph is a network of nodes connected by lines.

"图" 是用线连起来的一堆 "节点"

You can think of it like a map, with cities and roads connecting them.

可以想成地图，每个节点是一个城市，线是公路

Routes between these cities take different amounts of time.

一个城市到另一个城市，花的时间不同

We can label each line with what is called a cost or weight.

可以用 成本(cost) 或 权重(weight) 来代称

In this case, it's weeks of travel.

代表要几个星期

Now let's say we want to find the fastest route

假设想找"高庭"到"凛冬城"的最快路线

for an army at Highgarden to reach the castle at Winterfell.

假设想找"高庭"到"凛冬城"的最快路线

The simplest approach would just be to try every single path exhaustively

最简单的方法是尝试每一条路

and calculate the total cost of each.

计算总成本

That's a brute force approach.

这是蛮力方法

We could have used a brute force approach in sorting,

假设用蛮力方法 来排序数组

by systematically trying every permutation of the array to check if it's sorted.

尝试每一种组合，看是否排好序

This would have an N factorial complexity

这样的时间复杂度是 O(n!)

- that is the number of nodes, times one less, times one less than that, and so on until 1.

n 是节点数，n! 是 n 乘 n-1 乘 n-2... 一直到 1

Which is way worse than even N squared.

比 O(n ) 还糟糕

But, we can be way more clever!

我们可以更聪明些！

The classic algorithmic solution to this graph problem was invented by

图搜索问题的经典算法 发明者是

one of the greatest minds in computer science practice and theory, Edsger Dijkstra

理论计算机科学的伟人 Edsger Dijkstra

so it's appropriately named Dijkstra's algorithm.

所以叫 "Dijkstra 算法"

We start in Highgarden with a cost of 0, which we mark inside the node.

从"高庭"开始，此时成本为0，把0标在节点里

For now, we mark all other cities with question marks

其他城市标成问号，因为不知道成本多少

- we don't know the cost of getting to them yet.

其他城市标成问号，因为不知道成本多少

Dijkstra's algorithm always starts with the node with lowest cost.

Dijkstra 算法总是从成本最低的节点开始

In this case, it only knows about one node, Highgarden, so it starts there.

目前只知道一个节点 "高庭", 所以从这里开始

It follows all paths from that node to all connecting nodes that are one step away,

跑到所有相邻节点，记录成本

and records the cost to get to each of them.

跑到所有相邻节点，记录成本

That completes one round of the algorithm.

完成了一轮算法

We haven't encountered Winterfell yet,

但还没到"凛冬城"

so we loop and run Dijkstra's algorithm again.

所以再跑一次 Dijkstra 算法

With Highgarden already checked,

"高庭" 已经知道了

the next lowest cost node is King's Landing.

下一个成本最低的节点，是 "君临城"

Just as before, we follow every unvisited line to any connecting cities.

就像之前，记录所有相邻节点的成本

The line to The Trident has a cost of 5.

到"三叉戟河"的成本是 5

However, we want to keep a running cost from Highgarden,

然而我们想记录的是，从"高庭"到这里的成本

so the total cost of getting to The Trident is 8 plus 5, which is 13 weeks.

所以"三叉戟河"的总成本是 8+5=13周

Now we follow the offroad path to Riverrun,

现在走另一条路到"奔流城"

which has a high cost of 25, for a total of 33.

成本高达 25 ，总成本 33

But we can see inside of Riverrun that we've already found a path with a lower cost of just 10.

但 "奔流城" 中最低成本是 10

So we disregard our new path, and stick with the previous, better path.

所以无视新数字，保留之前的成本 10

We've now explored every line from King's Landing and didn't find Winterfell, so we move on.

现在看了"君临城"的每一条路，还没到"凛冬城" 所以继续.

The next lowest cost node is Riverrun, at 10 weeks.

下一个成本最低的节点，是"奔流城"，要 10 周

First we check the path to The Trident, which has a total cost of 10 plus 2, or 12.

先看 "三叉戟河" 成本： 10+2=12

That's slightly better than the previous path we found, which had a cost of 13,

比之前的 13 好一点

so we update the path and cost to The Trident.

所以更新 "三叉戟河" 为 12

There is also a line from Riverrun to Pyke with a cost of 3.

"奔流城"到"派克城"成本是 3

10 plus 3 is 13, which beats the previous cost of 14,

10+3=13，之前是14

and so we update Pyke's path and cost as well.

所以更新 "派克城" 为 13

That's all paths from Riverrun checked. so you guessed it, Dijkstra's algorithm loops again.

"奔流城"出发的所有路径都走遍了， \N 你猜对了，再跑一次 Dijkstra 算法

The node with the next lowest cost is The Trident

下一个成本最低的节点，是"三叉戟河"

and the only line from The Trident that we haven't checked is a path to Winterfell!

从"三叉戟河"出发，唯一没看过的路，通往"凛冬城"！

It has a cost of 10,

成本是 10

plus we need to add in the cost of 12 it takes to get to The Trident,

加"三叉戟河"的成本 12

for a grand total cost of 22.

总成本 22

We check our last path, from Pyke to Winterfell, which sums to 31.

再看最后一条路，"派克城"到"凛冬城"，成本 31

Now we know the lowest total cost, and also the fastest route for the army to get there,

现在知道了最低成本路线，让军队最快到达,

which avoids King's Landing!

还绕过了"君临城"！

Dijkstra's original algorithm, conceived in 1956,

Dijkstra 算法的原始版本，构思于 1956 年

had a complexity of the number of nodes in the graph squared.

算法复杂度是 O(n )

And squared, as we already discussed, is never great,

前面说过这个效率不够好

because it means the algorithm can't scale to big problems

意味着输入不能很大

- like the entire road map of the United States.

比如美国的完整路线图

Fortunately, Dijkstra's algorithm was improved a few years later

幸运的是，Dijkstra 算法几年后得到改进

to take the number of nodes in the graph,

变成 O(n log n + l)

times the log of the number of nodes, PLUS the number of lines.

n 是节点数，l 是多少条线

Although this looks more complicated,

虽然看起来更复杂

it's actually quite a bit faster.

但实际更快一些

Plugging in our example graph, with 6 cities and 9 lines, proves it.

用之前的例子，可以证明更快 \N （6 个节点 9 条线）

Our algorithm drops from 36 loops to around 14.

从 36 减少到 14 左右

As with sorting,

就像排序，图搜索算法也有很多，有不同优缺点

there are innumerable graph search algorithms, with different pros and cons.

就像排序，图搜索算法也有很多，有不同优缺点

Every time you use a service like Google Maps to find directions,

每次用谷歌地图时

an algorithm much like Dijkstra's is running on servers to figure out the best route for you.

类似 Dijkstra 的算法就在服务器上运行，找最佳路线

Algorithms are everywhere

算法无处不在

and the modern world would not be possible without them.

现代世界离不开它们

We touched only the very tip of the algorithmic iceberg in this episode,

这集只触及了算法的冰山一角

but a central part of being a computer scientist

但成为计算机科学家的核心

is leveraging existing algorithms and writing new ones when needed,

是根据情况合理决定 用现有算法 还是自己写新算法

and I hope this little taste has intrigued you to SEARCH further.

希望这集的小例子能让你体会到这点

I'll see you next week.

下周见

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Last episode, we discussed a few example classic algorithms,

上集讲了一些经典算法

like sorting a list of numbers and finding the shortest path in a graph.

比如给数组排序，找图的最短路径

What we didn't talk much about,

而上集没讲的是

is how the data the algorithms ran on was stored in computer memory.

算法处理的数据 存在内存里的格式是什么

You don't want your data to be like John Green's college dorm room,

你肯定不想数据像 John Green 的大学宿舍一样乱 \N 到处都是食物，衣服和纸

with food, clothing and papers strewn everywhere.

你肯定不想数据像 John Green 的大学宿舍一样乱 \N 到处都是食物，衣服和纸

Instead, we want our data to be structured,

我们希望数据是结构化的，方便读取

so that it's organized, allowing things to be easily retrieved and read.

我们希望数据是结构化的，方便读取

For this, computer scientists use Data Structures!

因此计算机科学家发明了 "数据结构"！

We already introduced one basic data structure last episode,

上集已经介绍了一种基本数据结构：

Arrays, also called lists or Vectors in some languages.

数组（Array）\N 也叫列表（list）或向量（Vector）（在其它编程语言里）

These are a series of values stored in memory.

数组的值一个个连续存在内存里

So instead of just a single value being saved into a variable, like 'j equals 5',

所以不像之前，一个变量里只存一个值（比如 j = 5）

we can define a whole series of numbers, and save that into an array variable.

我们可以把多个值存在数组变量里

To be able to find a particular value in this array, we have to specify an index.

为了拿出数组中某个值，我们要指定一个下标（index）

Almost all programing languages start arrays at index 0,

大多数编程语言里，数组下标都从 0 开始

and use a square bracket syntax to denote array access.

用方括号 [ ] 代表访问数组

So, for example, if we want to add the values in the first and third spots of our array 'j',

如果想相加数组 J 的第一个和第三个元素

and save that into a variable 'a', we would write a line of code like this.

把结果存在变量 a，可以写上图这样一行代码

How an array is stored in memory is pretty straightforward.

数组存在内存里的方式 十分易懂

For simplicity, let's say that the compiler chose to store ours at memory location 1,000.

为了简单，假设编译器从内存地址 1000 开始存数组

The array contains 7 numbers, and these are stored one after another in memory, as seen here.

数组有7个数字，像上图一样按顺序存.

So when we write "j index of 0", the computer goes to memory location 1,000,

写 j[0]，会去内存地址 1000

with an offset of 0, and we get the value 5.

加 0 个偏移，得到地址 1000，拿值：5

If we wanted to retrieve "j index of 5", our program goes to memory location 1000,

如果写 j[5]，会去内存地址 1000

plus an offset of 5, which in this case, holds a value of 4.

加 5 个偏移，得到地址 1005，拿值： 4

It's easy to confuse the fifth number in the array with the number at index 5.

很容易混淆 "数组中第 5 个数" 和 "数组下标为 5 的数"

They are not the same.

它们不是一回事

Remember, the number at index 5 is the 6th number in the array

记住，下标 5 其实是数组中第 6 个数

because the first number is at index 0.

因为下标是从 0 开始算的

Arrays are extremely versatile data structures, used all the time,

数组的用途广泛

and so there are many functions that can handle them to do useful things.

所以几乎所有编程语言 都自带了很多函数来处理数组

For example, pretty much every programming language comes with a built-in sort function,

举例，数组排序函数很常见

where you just pass in your array, and it comes back sorted.

只需要传入数组，就会返回排序后的数组

So there's no need to write that algorithm from scratch.

不需要写排序算法

Very closely related are Strings, which are just arrays of characters,

数组的亲戚是 字符串 (string)

like letters, numbers, punctuation and other written symbols.

其实就是字母,数字,标点符号等 组成的数组

We talked about how computers store characters way back in Episode 4.

第 4 集讨论过计算机怎么存储字符

Most often, to save a string into memory, you just put it in quotes, like so.

写代码时 用引号括起来就行了 \N j = "STAN ROCKS"

Although it doesn't look like an array, it is.

虽然长的不像数组，但的确是数组

Behind the scenes, the memory looks like this.

幕后看起来像这样

Note that the string ends with a zero in memory.

注意，字符串在内存里以 0 结尾

It's not the character zero, but the binary value 0.

不是"字符0"，是"二进制值0" \N 这叫字符"null"，表示字符串结尾

This is called the null character, and denotes the end of the string in memory.

不是"字符0"，是"二进制值0" \N 这叫字符"null"，表示字符串结尾

This is important because if I call a function like "print quote",

这个字符非常重要，如果调用 print 函数

which writes the string to the screen,

print 在屏幕上输出字符串

it prints out each character in turn starting at the first memory location,

会从开始位置，逐个显示到屏幕

but it needs to know when to stop!

但得知道什么时候停下来！

Otherwise, it would print out every single thing in memory as text.

否则会把内存里所有东西 都显示出来

The zero tells string functions when to stop.

0 告诉函数何时停下

Because computers work with text so often,

因为计算机经常处理字符串，所以有很多函数专门处理字符串

there are many functions that specifically handle strings.

因为计算机经常处理字符串，所以有很多函数专门处理字符串

For example, many programming languages have a string concatenation function, or "strcat",

比如连接字符串的 strcat

which takes in two strings, and copies the second one to the end of the first.

strcat 接收两个字符串，把第二个放到第一个结尾.

We can use arrays for making one dimensional lists,

我们可以用数组做一维列表

but sometimes you want to manipulate data that is two dimensional,

但有时想操作二维数据

like a grid of numbers in a spreadsheet, or the pixels on your computer screen.

比如电子表格，或屏幕上的像素

For this, we need a Matrix.

那么需要 矩阵（Matrix）

You can think of a Matrix as an array of arrays!

可以把矩阵看成 数组的数组！

So a 3 by 3 matrix is really an array of size 3, with each index storing an array of size 3.

一个 3x3 矩阵就是一个长度为3的数组 \N 数组里每个元素都是一个长度为3的数组

We can initialize a matrix like so.

可以这样初始化.

In memory, this is packed together in order like this.

内存里是这样排列的

To access a value, you need to specify two indexes, like "J index of 2, then index of 1" -

为了拿一个值，需要两个下标，比如 j[2][1]

this tells the computer you're looking for the item in subarray 2 at position 1.

告诉计算机在找数组 2 里，位置是 1 的元素

And this would give us the value 12.

得到数字 12

The cool thing about matrices is we're not limited to 3 by 3

矩阵酷的地方是，不止能做 3x3 的矩阵

-- we can make them any size we want

任何维度都行

-- and we can also make them any number of dimensions we want.

任何维度都行

For example, we can create a five dimensional matrix and access it like this.

可以做一个5维矩阵，然后这样访问 \N a = j[2][0][18][18][3]

That's right, you now know how to access a five dimensional matrix

现在你知道了 怎么读一个 5 维矩阵

-- tell your friends!

快去告诉你的朋友！

So far, we've been storing individual numbers or letters into our arrays or matrices.

目前我们只存过单个数字/字符，存进数组或矩阵

But often it's useful to store a block of related variables together.

但有时, 把几个有关系的变量存在一起, 会很有用

Like, you might want to store a bank account number along with its balance.

比如银行账户号和余额

Groups of variables like these can be bundled together into a Struct.

多个变量打包在一起叫 结构体 (Struct)

Now we can create variables that aren't just single numbers,

现在多个不同类型数据，可以放在一起

but are compound data structures, able to store several pieces of data at once.

现在多个不同类型数据，可以放在一起

We can even make arrays of structs that we define,

甚至可以做一个数组，里面放很多结构体

which are automatically bundled together in memory.

这些数据在内存里 会自动打包在一起

If we access, for example, J index of 0, we get back the whole struct stored there,

如果写 j[0]，能拿到 j[0] 里的结构体

and we can pull the specific account number and balance data we want.

然后拿银行账户和余额

This array of structs, like any other array,

存结构体的数组，和其它数组一样

gets created at a fixed size that can't be enlarged to add more items.

创建时就有固定大小，不能动态增加大小

Also, arrays must be stored in order in memory,

还有，数组在内存中 按顺序存储

making it hard to add a new item to the middle.

在中间插入一个值很困难

But, the struct data structure can be used for

但结构体可以创造更复杂的数据结构，消除这些限制

building more complicated data structures that avoid these restrictions.

但结构体可以创造更复杂的数据结构，消除这些限制

Let's take a look at this struct that's called a "node".

我们来看一个结构体，叫 节点(node)

It stores a variable, like a number, and also a pointer.

它存一个变量 \N 一个指针（pointer）

A pointer is a special variable that points, hence the name, to a location in memory.

"指针" 是一种特殊变量，指向一个内存地址，因此得名.

Using this struct, we can create a linked list,

用 节点 可以做 链表（linked list）

which is a flexible data structure that can store many nodes.

链表是一种灵活数据结构，能存很多个 节点 (node)

It does this by having each node point to the next node in the list.

灵活性是通过每个节点 指向 下一个节点实现的

Let's imagine we have three node structs saved in memory, at locations 1000, 1002 and 1008.

假设有三个节点，在内存地址 1000，1002, 1008

They might be spaced apart because they were created at different times,

隔开的原因 可能是创建时间不同

and other data can sit between them.

它们之间有其他数据

So, you see that the first node contains the value 7, and the location 1008 in its "next" pointer.

可以看到第一个节点，值是 7，指向地址 1008

This means that the next node in the linked list is located at memory location 1008.

代表下一个节点，位于内存地址 1008

Looking down the linked list, to the next node,

现在来到下一个节点

we see it stores the value 112 and points to another node at location 1002.

值是 112，指向地址 1002

If we follow that, we find a node that contains the value 14

如果跟着它，会看到一个值为 14 的节点

and points back to the first node at location 1000.

这个节点 指回地址 1000，也就是第一个节点

So this linked list happened to be circular,

这叫 循环链表

but it could also have been terminated by using a next pointer value of 0

但链表也可以是非循环的，最后一个指针是 0

-- the null value -- which would indicate we've reached the end of the list.

"null"，代表链表尽头

When programmers use linked lists,

当程序员用链表时

they rarely look at the memory values stored in the next pointers.

很少看指针具体指向哪里

Instead, they can use an abstraction of a linked list, that looks like this,

而是用链表的抽象模型，就像上图

which is much easier to conceptualize.

更容易看懂

Unlike an array, whose size has to be pre-defined,

数组大小需要预先定好

linked lists can be dynamically extended or shortened.

链表大小可以动态增减

For example, we can allocate a new node in memory,

可以创建一个新节点，通过改变指针值，把新节点插入链表

and insert it into this list, just by changing the next pointers.

可以创建一个新节点，通过改变指针值，把新节点插入链表

Linked Lists can also easily be re-ordered, trimmed, split, reversed, and so on.

链表也很容易重新排序，两端缩减，分割，倒序等

Which is pretty nifty!

超方便！

And pretty useful for algorithms like sorting, which we talked about last week.

链表也适合上集的排序算法

Owing to this flexibility, many more-complex data structures are built on top of linked lists

因为灵活，很多复杂数据结构 都用链表

The most famous and universal are queues and stacks.

最出名的是 队列（queue）和 栈（stack）

A queue - like the line at your post office - goes in order of arrival.

"队列" 就像邮局排队，谁先来就排前面

The person who has been waiting the longest, gets served first.

"队列" 就像邮局排队，谁先来就排前面

No matter how frustrating it is that all you want to do is buy stamps

虽然你可能只想买邮票，而前面的人要寄 23 个包裹

and the person in front of you seems to be mailing 23 packages.

虽然你可能只想买邮票，而前面的人要寄 23 个包裹

But, regardless, this behavior is called First-In First-Out, or FIFO.

这叫 先进先出（FIFO）

That's the first part.

我指队列，不是指那 23 个包裹

Not the 23 packages thing.

我指队列，不是指那 23 个包裹

Imagine we have a pointer, named "post office queue", that points to the first node in our linked list.

想象有个指针叫"邮局队列"，指向链表第一个节点

Once we're done serving Hank, we can read Hank's next pointer,

第一个节点是 Hank，服务完 Hank 之后 \N 读取 Hank 的指针

and update our "post office queue" pointer to the next person in the line.

把"邮局队列"指向下一个人

We've successfully dequeued Hank -- he's gone, done, finished.

这样就把 Hank "出队"（dequeue）了

If we want to enqueue someone, that is, add them to the line,

如果我们想把某人"入队"（enqueue） \N 意思是加到队列里

we have to traverse down the linked list until we hit the end,

要遍历整个链表到结尾

and then change that next pointer to point to the new person.

然后把结尾的指针，指向新人（Nick）

With just a small change, we can use linked lists as stacks, which are LIFO…

只要稍作修改，就能用链表做 栈，\N 栈是后进先出(LIFO)

Last-In First-Out.

只要稍作修改，就能用链表做 栈，\N 栈是后进先出(LIFO)

You can think of this like a stack of pancakes...

可以把"栈"想成一堆松饼

as you make them, you add them to the top of stack.

做好一个新松饼，就堆在之前上面

And when you want to eat one, you take them from the top of the stack.

吃的时候，是从最上面开始

Delicious!

美味！

Instead of enqueueing and dequeuing,

栈就不叫"入队""出队"了

data is pushed onto the stack and popped from the stacks.

叫"入栈"（push） "出栈"（pop）

Yep, those are the official terms!

对，这些是正确术语！

If we update our node struct to contain not just one, but two pointers,

如果节点改一下，改成 2 个指针

we can build trees,

就能做 树（tree）

another data structure that's used in many algorithms.

很多算法用了 "树" 这种数据结构

Again, programmers rarely look at the values of these pointers,

同样，程序员很少看指针的具体值

and instead conceptualize trees like this: The top most node is called the root.

而是把"树"抽象成这样：最高的节点叫"根节点"（root）

And any nodes that hang from other nodes are called children nodes.

"根节点"下的所有节点 都叫"子节点"（children）

As you might expect, nodes above children are called parent nodes.

任何子节点的直属上层节点，叫"母节点"（parent node）

Does this example imply that Thomas Jefferson is the parent of Aaron Burr?

这个例子能说明 托马斯·杰斐逊 是 阿龙·伯尔 的父亲吗？

I'll leave that to your fanfiction to decide.

我让你们的同人文来决定

And finally, any nodes that have no children

没有任何"子节点"的节点

-- where the tree ends -- are called Leaf Nodes.

也就是"树"结束的地方，叫"叶节点"（leaf）

In our example, nodes can have up to two children,

在这里的例子中，节点最多只可以有 2 个子节点

and for that reason, this particular data structure is called a binary tree.

因此叫 二叉树（binary tree）

But you could just as easily have trees with three, four or any number of children

但你可以随便改，弄成 3个，4个，或更多

by modifying the data structure accordingly.

但你可以随便改，弄成 3个，4个，或更多

You can even have tree nodes that use linked lists to store all the nodes they point to.

甚至节点 可以用链表存所有子节点

An important property of trees - both in real life and in data structures - is that

"树"的一个重要性质是（不管现实中还是数据结构中）

there's a one-way path from roots to leaves.

"根"到"叶"是 单向 的

It'd be weird if roots connected to leaves, that connected to roots.

如果根连到叶，叶连到根 就很奇怪

For data that links arbitrarily, that include things like loops,

如果数据随意连接，包括循环

we can use a graph data structure instead.

可以用"图"表示

Remember our graph from last episode of cities connected by roads?

还记得上集用路连接城市的"图"吗？

This can be stored as nodes with many pointers, very much like a tree,

这种结构 可以用有多个指针的节点表示

but there is no notion of roots and leaves, and children and parents…

因此没有 根 叶 子节点 父节点 这些概念

Anything can point to anything!

可以随意指向！

So that's a whirlwind overview

以上概述了计算机科学中，最主要的一些数据结构

of pretty much all of the fundamental data structures used in computer science.

以上概述了计算机科学中，最主要的一些数据结构

On top of these basic building blocks,

这些基本结构之上，程序员做了各种新变体，有不同性质.

programmers have built all sorts of clever variants, with slightly different properties

这些基本结构之上，程序员做了各种新变体，有不同性质.

-- data structures like red-black trees and heaps, which we don't have time to cover.

比如"红黑树"和"堆"，我们没时间讲

These different data structures have properties that are useful for particular computations.

不同数据结构适用于不同场景

The right choice of data structure can make your job a lot easier,

选择正确数据结构会让工作更简单

so it pays off to think about how you want to structure your data before you jump in.

所以花时间考虑用什么数据结构是值得的

Fortunately, most programming languages come with

幸运的是，大多数编程语言自带了预先做好的数据结构

libraries packed full of ready-made data structures.

幸运的是，大多数编程语言自带了预先做好的数据结构

For example, C++ has its Standard Template Library, and Java has the Java Class Library.

比如，C++有"标准模板库"，Java有"Java 类库"

These mean programmers don't have to waste time implementing things from scratch,

程序员不用浪费时间从零写

and can instead wield the power of data structures to do more interesting things,

时间可以花在更有趣的事情

once again allowing us to operate at a new level of abstraction!

又提升了一层抽象！

I'll see you next week.

下周见！

Hi, I'm Carrie Anne and welcome to Crash Course computer science.

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Over the past a few episodes,

前几集我们聊了基础，比如函数，算法和数据结构

we've been building up our understanding of computer science fundamentals,

前几集我们聊了基础，比如函数，算法和数据结构

such as functions, algorithms and data structures.

前几集我们聊了基础，比如函数，算法和数据结构

Today, we're going to take a step back and look at the person

今天，我们来看一位对计算机理论 贡献巨大的人

who formulated many of the theoretical concepts that underline modern computation.

今天，我们来看一位对计算机理论 贡献巨大的人

The father of computer science

计算机科学之父

and not quite Benedict Cumberbatch lookalike, Alan Turing.

长得不怎么像 本尼 的 阿兰·图灵

Alan Mathison Turing was born in London in 1912

阿兰·马蒂森·图灵 于 1921 年出生在伦敦，\N 从小就表现出惊人数学和科学能力

and showed an incredible aptitude for maths and science throughout his early education.

阿兰·马蒂森·图灵 于 1921 年出生在伦敦，\N 从小就表现出惊人数学和科学能力

His first brush of what we now call computer science came in 1935

他对计算机科学的建树始于 1935 年

while he was a master student at King's College in Cambridge.

当时他是剑桥国王学院的硕士生

He set out to solve a problem posed by German Mathematician David Hilbert

他开始解决德国数学家 大卫·希尔伯特 提出的问题

known as the Entscheidungsproblem

叫 Entscheidungsproblem （德语）

or decision problem,

即"可判定性问题":

which asked the following:

即"可判定性问题":

is there an algorithm that takes, as input, a statement written in formal logic,

是否存在一种算法，输入正式逻辑语句 \N 输出准确的"是"或"否"答案？

and produces a "yes" or "no" answer that's always accurate?

是否存在一种算法，输入正式逻辑语句 \N 输出准确的"是"或"否"答案？

If such an algorithm existed,

如果这样的算法存在， \N 可以回答比如 "是否有一个数大于所有数"

we could use it to answer questions like, "Is there a number bigger than all numbers?"

如果这样的算法存在， \N 可以回答比如 "是否有一个数大于所有数"

No, there's not. We know the answer to that one,

不, 没有. 我们知道答案

but there are many other questions in mathematics that we'd like to know the answer too.

但有很多其他数学问题，我们想知道答案

So if this algorithm existed, we'd want to know it.

所以如果这种算法存在, 我们想知道

The American mathematician Alonzo Church first presented a solution to this problem in 1935.

美国数学家 阿隆佐·丘奇 \N 于 1935年 首先提出解决方法

He developed a system of mathematical expressions called Lambda Calculus

开发了一个叫"Lambda 算子"的数学表达系统

and demonstrated that no such universal algorithm could exist.

证明了这样的算法不存在

Although Lambda Calculus was capable of representing any computation,

虽然"Lambda 算子"能表示任何计算

the mathematical technique was difficult to apply and understand.

但它使用的数学技巧 难以理解和使用

At pretty much the same time on the other side of the Atlantic,

同时在大西洋另一边

Alan Turing came up with his own approach to solve the decision problem.

阿兰·图灵 想出了自己的办法来解决"可判定性问题"

He proposed a hypothetical computing machine, which we now call a Turing Machine.

提出了一种假想的计算机，现在叫"图灵机"

Turing Machines provided a simple, yet powerful

图灵机提供了简单又强大的数学计算模型

mathematical model of computation.

图灵机提供了简单又强大的数学计算模型

Although using totally different mathematics,

虽然用的数学不一样

they were functionally equivalent to lambda calculus in terms of their computational power.

但图灵机的计算能力和 Lambda 算子一样

However their relative simplicity made them much more popular

同时因为图灵机更简单，所以在新兴的计算机领域更受欢迎

in the burgeoning field of computer science.

同时因为图灵机更简单，所以在新兴的计算机领域更受欢迎

In fact, they're simple enough that I'm going to explain it right now.

因为它如此简单，我现在就给你解释

A Turing Machine is a theoretical computing device

图灵机是一台理论计算设备

There's also a state variable in which we can hold a piece of information

还有一个状态变量，保存当前状态

about the current state of the machine.

还有一个状态变量，保存当前状态

And a set of rules that describes what the machine does.

还有一组规则，描述机器做什么

Given a state and the current symbol the head is reading,

规则是根据 当前状态+读写头看到的符号，决定机器做什么

the rule can be to write a symbol on the tape,

结果可能是在纸带写入一个符号

change the state of the machine, move the read/write head to the left or right by one spot

或改变状态，或把读写头移动一格

or any combination of these actions.

或执行这些动作的组合

To make this concrete, let's work through a simple example:

为了更好理解，讲个简单例子：

a Turing Machine that reads a string of ones ending in a zero

让图灵机读一个以零结尾的字符串

and computes whether there is an even number of ones.

并计算 1 的出现次数 是不是偶数

If that's true,

如果是, 在纸带上写一个 1

the machine will write a one to the tape

如果是, 在纸带上写一个 1

and if it's false, it'll write a zero.

如果不是，在纸带上写一个 0

First we need to define our Turing machine rules.

首先要定义"图灵机"的规则

If the state is even and the current symbol of the tape is one,

如果当前状态是"偶数", 当前符号是1

then we update the machine state to odd and move the head to the right.

那么把状态更新为"奇数"，把读写头向右移动

On the other hand if the state is even and the current symbol is zero,

如果当前状态为偶数，当前符号是 0

which means we've reached the end of the string of ones,

意味着到了字符串结尾

then we write one to the tape and change the state to halt,

那么在纸带上写一个 1，并且把状态改成 停机(halt)

as in we're finished and the Turing machine has completed the computation.

状态改为"停机" 是因为图灵机已完成计算

We also need rules for when the Turing machine is in an odd state,

但我们还需要 2 条规则，来处理状态为奇数的情况

one rule for the symbol on the tape is a zero and another for when it is one.

一条处理 奇数 + 纸带是 0 的情况 \N 一条处理 奇数 + 纸带是 1 的情况

Lastly we need to define a Starting state, which we'll set to be even.

最后，要决定机器的初始状态，这里定成"偶数"

Now we've defined the rules in the starting state of our Turing machine,

定义好了 起始状态+规则

which is comparable to a computer program, we can run it on some example input.

就像写好了程序，现在可以输入了

Let's say we store 1 1 0 onto tape.

假设把"1 1 0"放在纸带上，有两个 1，是偶数

That's two ones, which means there is an even number of ones,

假设把"1 1 0"放在纸带上，有两个 1，是偶数

and if that's news to you,

如果"偶数"对你是新知识 \N 也许我们该开一门【十分钟速成课：数学】

We should probably get working on crash course Math.

如果"偶数"对你是新知识 \N 也许我们该开一门【十分钟速成课：数学】

Notice that our rules only ever move their head to the right

注意，规则只让 读写头 向右移动

so the rest of the tape is irrelevant.

其他部分无关紧要，为了简单所以留空

We'll leave it blank for simplicity.

其他部分无关紧要，为了简单所以留空

Our Turing machine is all ready to go so let's start it.

"图灵机"准备好了，开始吧

Our state is even and the first number we see is one.

机器起始状态为"偶数"，看到的第一个数是 1

That matches our topmost rule and so we execute the effect,

符合最上面那条规则，所以执行对应的步骤

which is to update the state to odd and move the read/write head to the right by one spot.

把状态更新到"奇数"， 读写头向右移动一格

Okay, now we see another one on the tape, But this time our state is odd

然后又看到 1, 但机器状态是"奇数"，所以执行第三条规则

and so we execute our third rule

然后又看到 1, 但机器状态是"奇数"，所以执行第三条规则

which sets the state back to even and moves the head to the right.

使机器状态变回"偶数"，读写头向右移动一格

Now we see a 0 and our current state is even

现在看到 0，并且机器状态是 偶数，所以执行第二条规则

so we execute our second rule

现在看到 0，并且机器状态是 偶数，所以执行第二条规则

which is to write a 1 to the tape signifying that yes, it's true,

在纸带上写 1，表示"真" 的确有偶数个 1

there is an even number of ones,

在纸带上写 1，表示"真" 的确有偶数个 1

and finally the machine halts.

然后机器停机

That's how Turing machines work.

这就是图灵机的原理，很简单对吧？

Pretty simple, right?

这就是图灵机的原理，很简单对吧？

so you might be wondering why there's such a big deal.

你可能想知道 有什么大不了的

Well, Turing shows that this simple hypothetical machine

图灵证明了这个简单假想机器

can perform any computation if given enough time and memory.

如果有足够时间和内存，可以执行任何计算

It's a general-purpose computer.

它是一台通用计算机

Our program was a simple example.

刚才的程序就是个简单例子

But with enough rules, states and tape,

只要有足够的规则，状态和纸带 可以创造任何东西

you could build anything

只要有足够的规则，状态和纸带 可以创造任何东西

- a web browser, world of warcraft, whatever!

浏览器, 魔兽世界 任何东西！

Of course it would be ridiculously inefficient, but it is theoretically possible.

当然 这样做效率很低，但理论上可行.

And that's why, as a model of computing,

所以图灵机是很强大的计算模型

it's such a powerful idea.

所以图灵机是很强大的计算模型

In fact, in terms of what it can and cannot compute

事实上，就可计算和不可计算而言

there's no computer more powerful than a turing machine.

没有计算机比图灵机更强大

A computer that is as powerful is called Turing complete.

和图灵机一样强大的，叫 "图灵完备"

Every modern computing system, your laptop, your smartphone

每个现代计算系统 比如笔记本电脑，智能手机

and even the little computer inside your microwave and thermostat

甚至微波炉和恒温器内部的小电脑

are all Turing Complete.

都是"图灵完备"的

To answer Hilbert's decision problem,

为了回答可判定性问题

Turing applied these new Turing machines to an intriguing computational puzzle:

他把图灵机用于一个有趣计算问题：

the halting problem.

"停机问题"

Put simply this asks

简单说就是

"Is there an algorithm that can determine,

"给定图灵机描述和输入纸带，是否有算法可以确定

given a description of a turing machine and the input from its tape,

"给定图灵机描述和输入纸带，是否有算法可以确定

whether the Machine will run forever or halt?"

机器会永远算下去还是到某一点会停机？

For example we know our Turing machine will halt when given the input 1 1 0

我们知道输入 1 1 0，图灵机会停机

Because we literally walk through the example until it halted,

因为刚做过这个例子，它最后停机了

but what about a more complex problem?

但如果是更复杂的问题呢？

Is there a way to figure out if the program will halt without executing it?

有没有办法在不执行的情况，弄清会不会停机？

Some programs might take years to run

一些程序可能要运行好几年

so it would be useful to know before we run it

所以在运行前知道 会不会出结果很有用

and wait and wait and wait and then start getting worried and wonder

否则就要一直等啊等，忧虑到底会不会出结果

and then decades later when you're old and gray control-alt-delete.

当几十年后变老了，再按强制结束

So much sadness!

好悲伤！

Unfortunately, Turing came up with a proof that shows the halting problem was in fact unsolvable,

图灵通过一个巧妙逻辑矛盾 \N 证明了停机问题是无法解决的

through a clever logical contradiction.

图灵通过一个巧妙逻辑矛盾 \N 证明了停机问题是无法解决的

Let's follow his reasoning.

我们来看看他的推理

Imagine we have a hypothetical Turing machine that takes a description of a program

想象有一个假想图灵机，\N 输入：问题的描述 + 纸带的数据

and some input for his tape

想象有一个假想图灵机，\N 输入：问题的描述 + 纸带的数据

and always outputs either Yes, it halts, or no, it doesn't.

输出 Yes 代表会"停机"，输出 No 代表不会

And I'm going to give this machine a fun name

我要给这台机器一个有趣的名字叫 H， \N 来自"停机"的第一个字母

H for Halts.

我要给这台机器一个有趣的名字叫 H， \N 来自"停机"的第一个字母

Don't worry about how it works.

不用担心它具体怎么工作

Let's just assume such a machine exists.

假设这样的机器存在就好 毕竟重点是推论

We're talking theory here.

假设这样的机器存在就好 毕竟重点是推论

Turing reasons if there existed a program whose halting behavior was not decidable by H,

图灵推理说： 如果有个程序， H 无法判断是否会"停机"

it would mean the halting problem is unsolvable.

意味着"停机问题"无法解决

To find one, Turing designed another Turing machine that built on top of H.

为了找到这样的程序，图灵用 H 设计了另一个图灵机

If H says the program halts,

如果 H 说程序会"停机"，那么新机器会永远运行（即不会停机）

then we'll make our new machine loop forever.

如果 H 说程序会"停机"，那么新机器会永远运行（即不会停机）

If the answer is no, it doesn't the halt.

如果 H 的结果为 No，代表不会停机

That will have the new machine output no and halt.

那么让新机器输出 No，然后"停机"

In essence, we're building a machine that does the opposite of what H says.

实质上是一台和 H 输出相反的机器

Halt if the program doesn't halt

如果程序不停机，就停机

and run forever if the program halts.

如果程序停机，就永远运行下去

So this argument will also need to add a splitter to the front of our new machine.

我们还需要在机器前面加一个分离器

So it accepts only one input and passes that as both the program and input into H.

让机器只接收一个输入，\N 这个输入既是程序，也是输入

Let's call this new Machine Bizzaro.

我们把这台新机器叫 异魔

So far this seems like a plausible machine right.

目前为止，这个机器不难理解

Now it's going to get pretty complicated.

但接下来马上会变复杂，会有点难懂

But bear with me for a second.

但接下来马上会变复杂，会有点难懂

Look what happens when you pass bizzaro a description of itself as the input.

如果把 异魔 的描述，作为本身的输入会怎样

This means We're asking h what bizzaro will do when asked to evaluate itself.

意味着在问 H ，当异魔的输入是自己时会怎样

But if H says Bizzaro halts,

但如果 H 说异魔会停机

then Bizzaro enters its infinite loop and thus doesn't halt.

那么异魔会进入无限循环，因此不会停机

And if H says the Bizzaro doesn't halt, then Bizzaro outputs no and halt.

如果 H 说异魔不会停机，那么异魔会输出 No 然后停机

So H can't possibly decide the halting problem correctly

所以 H 不能正确判定 停机问题

because there is no answer.

因为没有答案

It's a paradox.

这是一个悖论

And this paradox means that the halting problem cannot be solved with Turing machines.

意味着"停机问题"不能用图灵机解决

Remember Turing proves that Turing machines could implement any computation.

还记得刚刚说： 图灵证明了图灵机可以实现任何计算

So this solution to the halting problem proves

"停机问题"证明了，不是所有问题都能用计算解决

that not all problems can be solved by computation.

"停机问题"证明了，不是所有问题都能用计算解决

Wow, that's some heavy stuff.

哇，好难理解

I might have to watch that again myself.

我都可能要再看一遍

Long story short, Church and Turing showed there were limits to the ability of computers.

长话短说，丘奇和图灵证明了计算机的能力有极限

No matter how much time or memory you have,

无论有多少时间或内存，有些问题是计算机无法解决的

there are just some problems that cannot be solved ever.

无论有多少时间或内存，有些问题是计算机无法解决的

The concurrent efforts by Church and Turing to determine the limits of computation,

丘奇和图灵证明了计算是有极限的，

and in general, formalize computability, are now called the Church-Turing Thesis.

起步了可计算性理论，现在叫"丘奇-图灵论题"

At this point in 1936, Turing was only 24 years old

当时是1936年，图灵只有24岁

and really only just beginning his career.

他的职业生涯才刚刚开始

From 1936 through 1938,

从1936年到1938年 \N 在丘奇指导下，他在普林斯顿拿到博士学位

he completed a PhD at Princeton University under the guidance of Church

从1936年到1938年 \N 在丘奇指导下，他在普林斯顿拿到博士学位

then after graduating he returned to Cambridge.

毕业后回到剑桥

Shortly after in 1939, Britain became embroiled in World War II.

1939年后不久，英国卷入第二次世界大战

Turing's genius was quickly applied for the war effort.

图灵的才能很快被投入战争

In fact, a year before the war started,

事实上，在战争开始前一年

he was already working part-time at the UK's government Code and Cypher school,

他已经在英国政府的 密码破译学校兼职

which was the British code breaking group based out of Bletchley Park.

位于"布莱切利园"的一个密码破译组织

One of his main efforts was figuring out how to decrypt German communications.

他的工作内容之一是破解德国的通信加密

Especially those that use the Enigma Machine.

特别是"英格玛机"加密的信息

In short, these machines scrambled text

简单说，英格玛机会加密明文

like you type the letters H-E-L-L-O

如果输入字母 H-E-L-L-O

and the letters X-W-D-B-J would come out.

机器输出 X-W-D-B-J

This process is called Encryption.

这个过程叫"加密"

The scrambling wasn't random.

文字不是随便打乱的

The behavior was defined by a series of real world rotors on the top of the enigma machine.

加密由"英格玛机"顶部的齿轮组合决定

Each were 26 possible rotational positions.

每个齿轮有26个可能位置

There was also a plug board at the front of the machine that allow pairs of letters to be swapped.

机器前面还有插板，可以将两个字母互换

In total, there were billions of possible settings.

总共有上十亿种可能

If you had your only enigma machine and you knew the correct rotor and plug board settings,

如果你有"英格玛机"，并且知道正确的齿轮和插头设置

you could type in X-W-D-B-J and "hello" would come out.

输入X-W-D-B-J，机器会输出 hello

In other words, you decrypted the message.

解密了这条消息

Of course, the German military wasn't sharing their enigma settings on Social Media.

当然，德军不会把机器设置发到微博上

So the allies had to break the code.

盟军必须自己破译密码

With billions of Rotor and plug board combinations,

有数十亿种组合，根本没法手工尝试所有组合

there was no way to check them all by hand.

有数十亿种组合，根本没法手工尝试所有组合

Fortunately for Turing, Enigma Machines and the people who operated them were not perfect.

幸运的是，英格玛机和操作员不是完美的

Like one key flaw was that a letter would never be encoded as itself,

一个大缺陷是：字母加密后绝不会是自己

as in an H was never encrypted as an H.

H 加密后绝对不是 H

Turing building on earlier work by Polish code breakers

图灵接着之前波兰破译专家的成果继续工作

designed a special-purpose electro-mechanical computer called the bombe

设计了一个机电计算机，叫 Bombe

that took advantages of this flaw.

利用了这个缺陷，它对加密消息尝试多种组合

It tried lots and lots of combinations of enigma settings for a given encrypted message.

利用了这个缺陷，它对加密消息尝试多种组合

If the bombe found a setting that led to a letter being encoded as itself

如果发现字母解密后和原先一样

which we know no enigma machines could do.

我们知道英格玛机决不会这么做

That combination was discarded then the machine moved on to try another combination.

这个组合会被跳过，接着试另一个组合

So bombe was used to greatly narrow the number of Possible enigma settings.

Bombe 大幅减少了搜索量

This allowed human code breakers to hone their efforts on the most probable solutions,

让破译人员把精力花在更有可能的组合

looking for things like common german words in fragments of decoded text.

比如在解码文本中找常见的德语单词

Periodically, the Germans would suspect someone was decoding their communications

德国人时不时会怀疑有人在破解，然后升级英格玛机

and upgrade the enigma machine,

德国人时不时会怀疑有人在破解，然后升级英格玛机

like they'd add another rotor creating many more combinations.

比如加一个齿轮，创造更多可能组合

They even built entirely new encryption machines.

他们甚至还做了全新的加密机

Throughout the war, Turing and his colleagues at Bletchley Park

整个战争期间，图灵和同事在布莱切利园努力破解加密

worked tirelessly to defeat these mechanisms.

整个战争期间，图灵和同事在布莱切利园努力破解加密

And overall, the intelligence gained from decrypted German communications

解密得到的德国情报，为盟军赢得了很多优势

gave the allies an edge in many theaters

解密得到的德国情报，为盟军赢得了很多优势

with some historians arguing is shortened the war by years.

有些史学家认为他们把战争减短了好几年

After the war, Turing return to academia

战后，图灵回到学术界 \N 为许多早期计算机工作做出贡献

and contributed to many early electronic computing efforts

战后，图灵回到学术界 \N 为许多早期计算机工作做出贡献

like the Manchester Mark 1, which was an early and influential stored-program computer.

比如曼彻斯特 1 号，一个早期有影响力的存储程序计算机

But his most famous post-war contribution was the Artificial Intelligence.

但他最有名的战后贡献是"人工智能"

A field's so new that it didn't get that name until 1956.

这个领域很新，直到1956年才有名字

It's a huge topic. So we'll get to it again in future episodes.

这个话题很大，以后再谈（第34集）

In 1950, Turing could envision a future where computers

1950 年，图灵设想了未来的计算机

were powerful enough to exhibit intelligence equivalent to

拥有和人类一样的智力，或至少难以区分

or at least indistinguishable from that of a human.

拥有和人类一样的智力，或至少难以区分

Turing postulated that a computer would deserve to be called intelligent

图灵提出 \N 如果计算机能欺骗人类相信它是人类，才算是智能

if it could deceive a human into believing that it was human.

图灵提出 \N 如果计算机能欺骗人类相信它是人类，才算是智能

This became the basis of a simple test, now called the Turing test.

这成了智能测试的基础，如今叫"图灵测试"

Imagine that you are having a conversation with two different people

想像你在和两个人沟通 \N 不用嘴或面对面，而是来回发消息

not by voice or in person, but by sending type notes back and forth,

想像你在和两个人沟通 \N 不用嘴或面对面，而是来回发消息

you can ask any questions you want and you get replies.

可以问任何问题，然后会收到回答

But one of those two people is actually a computer.

但其中一个是计算机

If you can't tell which one is human and which one is a computer,

如果你分不出哪个是人类，哪个是计算机

then the computer passes the test.

那么计算机就通过了图灵测试

There's a modern version of this test called

这个测试的现代版叫

a completely automated public turing test to tell computers and humans apart

"公开全自动图灵测试，用于区分计算机和人类"

or Captcha for short.

简称"验证码"

These are frequently used on the internet to prevent automated systems

防止机器人发垃圾信息等

from doing things like posting spam on websites.

防止机器人发垃圾信息等

I'll admit sometimes I can't read what those squiggly things say.

我承认 有时我都认不出那些扭曲的东西是什么字

Does that mean I'm a computer?

这难道意味着我是计算机？

Normally in this series, we don't delve into the personal lives of these historical figures.

通常这个系列我们不会深入历史人物的个人生活

But in Turing's case his name has been inextricably tied to tragedy

但图灵与悲剧密不可分

so his story is worth mentioning.

所以他的故事值得一提

Turing was gained a time when homosexuality was illegal in the United Kingdom and much of the world.

图灵那个时代，同性恋是违法的，英国和大部分国家都是

And an investigation into a 1952 Burglary at his home

1952 年调查他家的入室盗窃案时，向当局暴露了他的性取向

revealed his sexual orientation to the authorities,

1952 年调查他家的入室盗窃案时，向当局暴露了他的性取向

who charged him with gross indecency.

被起诉 "行为严重不检点"

Turing was convicted and given a choice between imprisonment,

图灵被定罪，有2个选择： \N 1. 入狱 2. 接受激素来压制性欲

or probation with hormonal treatments to suppress his sexuality.

图灵被定罪，有2个选择： \N 1. 入狱 2. 接受激素来压制性欲

He chose the latter in part to continue his academic work,

他选了后者，部分原因是为了继续学术工作

but it altered his mood and personality.

但药物改变了他的情绪和性格

Although the exact circumstances will never be known,

虽然确切情况永远无法得知

it's most widely accepted that Alan Turing took his own life by poison in 1954.

图灵于1954年服毒自尽，年仅41岁

He was only 41.

图灵于1954年服毒自尽，年仅41岁

Many things have been named in recognition of Turing's contributions to theoretical computer science

由于图灵对计算机科学贡献巨大，许多东西以他命名

But perhaps the most prestigious among them is the Turing award

其中最出名的是"图灵奖"

the highest distinction in the field of computer science.

计算机领域的最高奖项

Equivalent to a Nobel prize in Physics, chemistry or other sciences.

相当于物理, 化学等其它领域的诺贝尔奖

Despite a life cut short, Alan inspire the first generation of computer scientists

虽然英年早逝，但图灵激励了第一代计算机科学家

and lead key groundwork that enabled a digital era that we get to enjoy today.

而且为如今便利的数字时代 做出了重要基石性工作

I'll see you next week.

我们下周见

Hi, I'm Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

So we've talked a lot about sorting in this series

之前花了很多时间讲排序

and often code to sort a list of numbers might only be ten lines long,

也写了些 10 行左右的排序代码

which is easy enough for a single programmer to write.

对1个程序员来说很容易写

Plus, it's short enough that you don't need any special tools

而且代码很短，不必用专门工具

- you could do it in Notepad.

- 记事本就够了

Really!

真的！

But, a sorting algorithm isn't a program;

但排序算法很少会是独立程序 \N 更可能是大项目的一小部分

it's likely only a small part of a much larger program.

但排序算法很少会是独立程序 \N 更可能是大项目的一小部分

For example, Microsoft Office has roughly 40 millions lines of code.

举个例子，微软的 Office 大约有 4000 万代码

40 MILLION!

4000 万！

That's way too big for any one person to figure out and write!

太多了，一个人不可能做到

To build huge programs like this, programmers use a set of tools and practices.

为了写大型程序，程序员用各种工具和方法

Taken together, these form the discipline of Software Engineering

所有这些形成了"软件工程"学科

- a term coined by engineer Margaret Hamilton,

- 这个词由工程师 Margaret Hamilton 创造

who helped NASA prevent serious problems during the Apollo missions to the moon.

她帮助 NASA 在阿波罗计划中避免了严重问题

She once explained it this way:

她曾说过：

"It's kind of like a root canal: you waited till the end,

"有点像牙根管治疗：你总是拖到最后才做 \N 但有些事可以预先做好

[but] there are things you could have done beforehand.

"有点像牙根管治疗：你总是拖到最后才做 \N 但有些事可以预先做好

It's like preventative healthcare,

有点像预防性体检, 只不过是预防软件出错"

but it's preventative software."

有点像预防性体检, 只不过是预防软件出错"

As I mentioned in episode 12,

第 12 集提过

breaking big programs into smaller functions allows many people to work simultaneously.

把大项目分解成小函数 可以让多人同时工作

They don't have to worry about the whole thing,

不用关心整个项目，关心自己的函数就好了

just the function they're working on.

不用关心整个项目，关心自己的函数就好了

So, if you're tasked with writing a sort algorithm,

如果你的任务是写排序算法

you only need to make sure it sorts properly and efficiently.

你只需要确保高效和正确就可以了

However, even packing code up into functions isn't enough.

然而把代码打包成函数 依然不够

Microsoft Office probably contains hundreds of thousands of them.

如果只是这样，微软 Office 会有几十万个函数

That's better than dealing with 40 million lines of code,

虽然比 4000 万行代码要好一些，但还是太多了

but it's still way too many "things" for one person or team to manage.

虽然比 4000 万行代码要好一些，但还是太多了

The solution is to package functions into hierarchies,

解决办法是：把函数打包成层级. \N 把相关代码都放在一起，打包成对象（objects）

pulling related code together into "objects".

解决办法是：把函数打包成层级. \N 把相关代码都放在一起，打包成对象（objects）

For example, car's software might have several functions related to cruise control,

例如，汽车软件中 可能有几个和定速巡航有关的函数

like setting speed, nudging speed up or down, and stopping cruise control altogether.

比如 设定速度，逐渐加速减速，停止定速巡航

Since they're all related,

因为这些函数都相关，可以包装成一个"定速巡航对象"

we can wrap them up into a unified cruise control object.

因为这些函数都相关，可以包装成一个"定速巡航对象"

But, we don't have to stop there,

但不止如此，我们还可以做更多

cruise control is just one part of the engine's software.

"定速巡航"只是引擎软件的一部分

There might also be sets of functions that control spark plug ignition,

可能还有 "火花塞点火" "燃油泵" 和 "散热器"

fuel pumps, and the radiator.

可能还有 "火花塞点火" "燃油泵" 和 "散热器"

So we can create a "parent" Engine Object

我们可以做一个"引擎对象" 来包括所有"子"对象

that contains all of these "children" objects.

我们可以做一个"引擎对象" 来包括所有"子"对象

In addition to children \*objects\*,

除了子对象，"引擎对象"可能有自己的函数

the engine itself might have its \*own\* functions.

除了子对象，"引擎对象"可能有自己的函数

You want to be able to stop and start it, for example.

比如 开关引擎

It'll also have its own variables,

它也会有自己的变量，比如汽车行驶了多少英里

like how many miles the car has traveled.

它也会有自己的变量，比如汽车行驶了多少英里

In general, objects can contain other objects, functions and variables.

总的来说，对象可以包其它对象，函数和变量

And of course, the engine is just one part of a Car Object.

当然，"引擎对象"只是"汽车对象"的一部分

There's also the transmission, wheels, doors, windows, and so on.

还有传动装置，车轮，门，窗等

Now, as a programmer, if I want to set the cruise control,

作为程序员，如果想设"定速巡航"

I navigate down the object hierarchy,

要一层层向下

from the outermost objects to more and more deeply nested ones.

从最外面的对象往里找

Eventually, I reach the function I want to trigger:

最后找到想执行的函数：

"Car, then engine, then cruise control, then set cruise speed to 55".

Car. Engine. CruiseControl. setCruiseSpeed(55)

Programming languages often use something equivalent to the syntax shown here.

编程语言经常用类似这样的语法

The idea of packing up functional units into nested objects is called

把函数打包成对象的思想叫 "面向对象编程"

Object Oriented Programming.

把函数打包成对象的思想叫 "面向对象编程"

This is very similar to what we've done all series long:

这种思想和之前类似

hide complexity by encapsulating low-level details in higher-order components.

通过封装组件，隐藏复杂度

Before we packed up things like transistor circuits into higher-level boolean gates.

之前把晶体管 打包成了逻辑门

Now we're doing the same thing with software.

现在软件也这样做

Yet again, it's a way to move up a new level of abstraction!

又提升了一层抽象！

Breaking up a big program, like a car's software,

把大型软件（如汽车软件）\N 拆成一个个更小单元，适合团队合作

into functional units is perfect for teams.

把大型软件（如汽车软件）\N 拆成一个个更小单元，适合团队合作

One team might be responsible for the cruise control system,

一个团队负责定速巡航系统

and a single programmer on that team tackles a handful of functions.

团队里的一位程序员负责其中一些函数

This is similar to how big, physical things are built, like skyscrapers.

类似建摩天大楼

You'll have electricians running wires,

有电工装电线

plumbers fitting pipes,

管道工配管

welders welding,

焊接工焊接

painters painting,

油漆工涂油漆

and hundreds of other people teeming all over the hull.

还有成百上千人做其他事情

They work together on different parts simultaneously,

在不同岗位同时工作，各尽其能

leveraging their different skills.

在不同岗位同时工作，各尽其能

Until one day, you've got a whole working building!

直到整栋楼完成

But, returning to our cruise control example

回到定速巡航的例子

its code is going to have to make use of functions in other parts of the engine's software,

定速巡航 要用到引擎的其它函数，来保持车速

to, you know, keep the car at a constant speed.

定速巡航 要用到引擎的其它函数，来保持车速

That code isn't part of the cruise control team's responsibility.

定速巡航 团队不负责这些代码，另一个团队负责

It's another team's code.

定速巡航 团队不负责这些代码，另一个团队负责

Because the cruise control team didn't write that,

因为是其他团队的代码，

they're going to need good documentation about what each function in the code does,

定速巡航 团队需要文档 帮助理解代码都做什么

and a well-defined Application Programming Interface

以及定义好的 "程序编程接口" -简称 API

-- or API for short.

以及定义好的 "程序编程接口" -简称 API

You can think of an API as the way that

API 帮助不同程序员合作 \N 不用知道具体细节，只要知道怎么使用就行了

collaborating programmers interact across various parts of the code.

API 帮助不同程序员合作 \N 不用知道具体细节，只要知道怎么使用就行了

For example, in the IgnitionControl object,

例如"点火控制"对象中，可能有"设置发动机转数"函数

there might be functions to set the RPM of the engine,

例如"点火控制"对象中，可能有"设置发动机转数"函数

check the spark plug voltage,

"检查火花塞电压"函数

as well as fire the individual spark plugs.

"点燃单个火花塞"函数

Being able to set the motor's RPM is really useful,

"设置发动机转速"非常有用

the cruise control team is going to need to call that function.

"定速巡航"团队要用到这个函数

But, they don't know much about how the ignition system works.

但他们对点火系统不怎么了解

It's not a good idea to let them call functions that fire the individual spark plugs.

让他们调用"点燃单个火花塞"函数，不是好主意

Or the engine might explode!

引擎可能会炸！

Maybe.

可能啦

The API allows the right people access to the right functions and data.

API 控制哪些函数和数据让外部访问 \N 哪些仅供内部

Object Oriented Programming languages do this

"面向对象"的编程语言 \N 可以指定函数是 public 或 private，来设置权限

by letting you specify whether functions are public or private.

"面向对象"的编程语言 \N 可以指定函数是 public 或 private，来设置权限

If a function is marked as "private",

如果函数标记成 private

it means only functions inside that object can call it.

意味着 只有同一个对象内的其他函数能调用它

So, in this example, only other functions inside of IgnitionControl,

在这个例子里，只有内部函数比如 setRPM

like the setRPM function,

在这个例子里，只有内部函数比如 setRPM

can fire the sparkplugs.

才能调用 fireSparkplug 函数

On the other hand, because the setRPM function is marked as public,

而 setRPM 函数是 public \N 所以其它对象可以调用它，比如 定速巡航

other objects can call it, like cruise control.

而 setRPM 函数是 public \N 所以其它对象可以调用它，比如 定速巡航

This ability to hide complexity, and selectively reveal it,

"面向对象编程"的核心是 \N 隐藏复杂度，选择性的公布功能

is the essence of Object Oriented Programming,

"面向对象"的核心是 \N 隐藏复杂度，选择性的公布功能

and it's a powerful and popular way to tackle building large and complex programs.

因为做大型项目很有效，所以广受欢迎

Pretty much every piece of software on your computer, or game running on your console,

计算机上几乎所有软件，游戏机里几乎所有游戏

was built using an Object Oriented Programming Language,

都是 "面向对象" 编程语言写的

like C++, C# or Objective-C.

比如 C++, C#, Objective-C 等

Other popular "OO" languages you may have heard of are Python and Java.

其他流行 OO 语言，你可能听过 Python 和 Java

It's important to remember that code, before being compiled, is just text.

有一点很重要：代码在编译前就只是文字而已

As I mentioned earlier,

前面提过，你可以用记事本或任何文字处理器

you could write code in Notepad or any old word processor.

前面提过，你可以用记事本或任何文字处理器

Some people do.

有人确实这样做

But generally, today's software developers use special-purpose applications for writing programs,

但一般来说，现代软件开发者 会用专门的工具来写代码

ones that integrate many useful tools for writing, organizing, compiling and testing code.

工具里集成了很多有用功能\N 帮助写代码，整理，编译和测代码

Because they put everything you need in one place,

因为集成了所有东西

they're called Integrated Development Environments ,

因此叫 集成开发环境，简称 IDE

or IDEs for short.

因此叫 集成开发环境，简称 IDE

All IDEs provide a text editor for writing code,

所有 IDE 都有写代码的界面

often with useful features like automatic color-coding to improve readability.

还带一些有用功能，比如代码高亮，来提高可读性

Many even check for syntax errors as you type, like spell check for code.

许多 IDE 提供实时检查，比如拼写

Big programs contain lots of individual source files,

大型项目有很多源代码文件

so IDEs allow programmers to organize and efficiently navigate everything.

IDE 帮助开发者整理和看代码

Also built right into the IDE is the ability to compile and run code.

很多 IDE 还可以直接编译和运行代码

And if your program crashes,

如果程序崩了，因为你还没写完呢

because it's still a work in progress,

如果程序崩了，因为你还没写完呢

the IDE can take you back to the line of code where it happened,

IDE 可以定位到出错代码

and often provide additional information to help you track down and fix the bug,

还会提供信息 帮你解决问题

which is a process called debugging.

这叫 调试（debug）

This is important

调试很重要

because most programmers spend 70 to 80% of their time testing and debugging,

大多数程序员会花 70％~80％ 时间调试，而不是在写代码

not writing new code.

大多数程序员会花 70％~80％ 时间调试，而不是在写代码

Good tools, contained in IDEs,

好工具能极大帮助程序员防止和解决错误

can go a long way when it comes to helping programmers prevent and find errors.

好工具能极大帮助程序员防止和解决错误

Many computer programmers can be pretty loyal to their IDEs though

很多开发者只用一款 IDE

But let's be honest.

但承认吧，VIM 才是最棒的编辑器

VIM is where it's at.

但承认吧，VIM 才是最棒的编辑器

Providing you know how to quit.

如果你知道怎么退出的话

In addition to coding and debugging,

除了写代码和调试

another important part of a programmer's job is documenting their code.

程序员工作的另一个重要部分是 给代码写文档

This can be done in standalone files called "readme",

文档一般放在一个叫 README 的文件里

which tell other programmers to read that help file before diving in.

告诉其他程序员，看代码前先看这个文件

It can also happen right in the code itself with comment

文档也可以直接写成"注释"，放在源代码里

These are specially-marked statements that the program knows

注释是标记过的一段文字

to ignore when the code is compiled.

编译代码时 注释会被忽略

They exist only to help programmers figure out what's what in the source code.

注释存在的唯一作用 \N 就是帮助开发者理解代码

Good documentation helps programmers when they revisit code they haven't seen for awhile,

好文档能帮助开发者 \N 几个月后理解自己的代码，对其他人也很重要

but it's also crucial for programmers who are totally new to it.

好文档能帮助开发者 \N 几个月后理解自己的代码，对其他人也很重要

I just want to take a second here and reiterate that it's THE WORST

我想花一秒 再强调一下注释很重要

when someone parachutes a load of uncommented and undocumented code into your lap,

最糟糕的就是拿到一堆代码，没有任何注释和文档

and you literally have to go line by line to understand what the code is doing.

结果得逐行读代码，理解到底干嘛的

Seriously.

我是认真的

Don't be that person.

别做那种人

Documentation also promotes code reuse.

文档还可以提高复用性

So, instead of having programmers constantly write the same things over and over,

与其让程序员一遍遍写同样的东西

they can track down someone else's code that does what they need.

可以直接用别人写好的来解决问题

Then, thanks to documentation,

读文档看怎么用就行，不用读代码

they can put it to work in their program, without ever having to read through the code.

读文档看怎么用就行，不用读代码

"Read the docs" as they say.

"读文档啊"

In addition to IDEs,

除了 IDE，还有一个重要软件帮助团队协作

another important piece of software that

除了 IDE，还有一个重要软件帮助团队协作

helps big teams work collaboratively on big coding projects is called

除了 IDE，还有一个重要软件帮助团队协作

Source Control,

源代码管理

also known as version control or revision control.

也叫"版本控制"

Most often, at a big software company like Apple or Microsoft,

苹果或微软这样的大型软件公司

code for projects is stored on centralized servers,

会把代码放到一个中心服务器上

called a code repository .

叫"代码仓库"

When a programmer wants to work on a piece of code,

程序员想改一段代码时

they can check it out,

可以 check out

sort of like checking out a book out from a library.

有点像从图书馆借书

Often, this can be done right in an IDE.

一般这种操作，可以直接在 IDE 内完成

Then, they can edit this code all they want on their personal computer,

然后开发者在自己的电脑上编辑代码

adding new features and testing if they work.

加新功能，测试

When the programmer is confident their changes are working and there are no loose ends,

如果代码没问题了，所有测试通过了

they can check the code back into the repository,

可以把代码放回去

known as committing code, for everyone else to use.

这叫 提交 (commit)

While a piece of code is checked out,

当代码被 check out，而且可能被改过了

and presumably getting updated or modified,

当代码被 check out，而且可能被改过了

other programmers leave it alone.

其他开发者不会动这段代码

This prevents weird conflicts and duplicated work.

防止代码冲突和重复劳动

In this way, hundreds of programmers can be simultaneously checking in and out pieces of code,

这样多名程序员可以同时写代码，建立庞大的系统

iteratively building up huge systems.

这样多名程序员可以同时写代码，建立庞大的系统

Critically, you don't want someone committing buggy code,

重要的是，你不希望提交的代码里有问题 \N 因为其他人可能用到了这些代码

because other people and teams may rely on it.

重要的是，你不希望提交的代码里有问题 \N 因为其他人可能用到了这些代码

Their code could crash, creating confusion and lost time.

导致他们的代码崩溃，造成困惑而且浪费时间

The master version of the code, stored on the server,

代码的主版本 (master)

should always compile without errors and run with minimal bugs.

应该总是编译正常，尽可能少 bug

But sometimes bugs creep in.

但有时 bug 还是会出现

Fortunately, source control software keeps track of all changes,

幸运的是，源代码管理可以跟踪所有变化

and if a bug is found,

如果发现 bug

the whole code, or just a piece,

全部或部分代码，可以"回滚"到之前的稳定版

can be rolled back to an earlier, stable version.

全部或部分代码，可以"回滚"到之前的稳定版

It also keeps track of who made each change,

"源代码管理" 也记录了谁改了什么代码

so coworkers can send nasty,

所以同事可以给你发 讨厌的

I mean, helpful

我是说"有帮助的" 邮件给写代码的人

and encouraging emails to the offending person.

我是说"有帮助的" 邮件给写代码的人

Debugging goes hand in hand with writing code,

写代码和测代码密不可分

and it's most often done by an individual or small team.

测试一般由个人或小团队完成

The big picture version of debugging is Quality Assurance testing, or QA.

测试可以统称 "质量保证测试"，简称 QA

This is where a team rigorously tests out a piece of software,

严格测试软件的方方面面

attempting to create unforeseen conditions that might trip it up.

模拟各种可能情况，看软件会不会出错

Basically, they elicit bugs.

基本上就是找 bug

Getting all the wrinkles out is a huge effort,

解决大大小小的错误需要很多工作

but vital in making sure the software works

但对确保软件质量至关重要

as intended for as many users in as many situations as imaginable before it ships.

让软件在各种情况下按预期运行

You've probably heard of beta software.

你可能听过 "beta 版" 软件

This is a version of software that's mostly complete,

意思是软件接近完成

but not 100% fully tested.

但不是 100％ 完全测试过

Companies will sometimes release beta versions to the public to help them identify issues,

公司有时会向公众发布 beta 版，以帮助发现问题

it's essentially like getting a free QA team.

用户就像免费的 QA 团队

What you don't hear about as much

你听过比较少的是 \N beta 版之前的版本：alpha 版本

is the version that comes before the beta: the alpha version.

你听过比较少的是 \N beta 版之前的版本：alpha 版本

This is usually so rough and buggy, it's only tested internally.

alpha 版一般很粗糙，错误很多，\N 经常只在公司内部测试

So, that's the tip of the iceberg in terms of the tools, tricks and techniques

以上只是软件工程师用的工具和技巧的冰山一角

that allow software engineers to construct the huge pieces of software that we know and love today,

它们帮助软件工程师制作令人喜爱的软件

like YouTube, Grand Theft Auto 5, and Powerpoint.

如 YouTube，GTA5 和 PPT 等等

As you might expect,

如你所料

all those millions of lines of code needs some serious processing power to run at useful speeds,

这些代码要强大的处理能力 才能高速速度运行

so next episode we'll be talking about how computers got so incredibly fast.

所以下集讨论，计算机怎么发展到如今这么快

See you then.

到时见

This episode is brought to you by Curiosity Stream.

本集由 Curiosity Stream 赞助播出

Hi, I’m Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne \N欢迎收看计算机科学速成课！

Over the past six episodes, we delved into software,

过去 6 集我们聊了软件 \N 从早期编程方式到现代软件工程

from early programming efforts to modern software engineering practices.

过去 6 集我们聊了软件 \N 从早期编程方式到现代软件工程

Within about 50 years, software grew in complexity from machine code punched by hand onto paper tape

在大概50年里 软件从纸带打孔 \N 变成面向对象编程语言 在集成开发环境中写程序

to object oriented programming languages, compiled in integrated development environments.

在大概50年里 软件从纸带打孔 \N 变成面向对象编程语言 在集成开发环境中写程序

But this growth in sophistication would not have been possible without improvements in hardware.

但如果没有硬件的大幅度进步 \N 软件是不可能做到这些的

To appreciate computing hardware’s explosive growth in power and sophistication,

为了体会硬件性能的爆炸性增长 \N 我们要回到电子计算机的诞生年代

we need to go back to the birth of electronic computing.

为了体会硬件性能的爆炸性增长 \N 我们要回到电子计算机的诞生年代

From roughly the 1940’s through the mid-1960s, every computer was built from individual parts,

大约 1940年代~1960年代中期这段时间里 \N 计算机都由独立部件组成

called discrete components, which were all wired together.

叫"分立元件" \N 然后不同组件再用线连在一起

For example, the ENIAC, consisted of more than 17,000 vacuum tubes, 70,000 resistors,

举例, ENIAC 有1万7千多个真空管, 7万个电阻

10,000 capacitors, and 7,000 diodes, all of which required 5 million hand-soldered connections.

1万个电容器, 7千个二极管, 5百万个手工焊点

Adding more components to increase performance meant more connections, more wires

如果想提升性能，就要加更多部件 \N 这导致更多电线，更复杂

and just more complexity, what was dubbed the Tyranny of Numbers.

这个问题叫 "数字暴政''

By the mid 1950s, transistors were becoming commercially available

1950 年代中期，晶体管开始商业化（市场上买得到） \N 开始用在计算机里

and being incorporated into computers.

1950 年代中期，晶体管开始商业化（市场上买得到） \N 开始用在计算机里

These were much smaller, faster and more reliable than vacuum tubes

晶体管比电子管 更小更快更可靠

but each transistor was still one discrete component.

但晶体管依然是分立元件

In 1959, IBM upgraded their vacuum-tube-based "709" computers to transistors by replacing

1959年，IBM 把 709 计算机从原本的电子管 \N 全部换成晶体管

by replacing all the discrete vacuum tubes with discrete transistors.

1959年，IBM 把 709 计算机从原本的电子管 \N 全部换成晶体管

The new machine, the IBM 7090, was six times faster and half the cost.

诞生的新机器 IBM 7090 \N 速度快 6 倍，价格只有一半

These transistorized computers marked the second generation of electronic computing.

晶体管标志着"计算 2.0 时代"的到来

However, although faster and smaller,

虽然更快更小 \N 但晶体管的出现 还是没有解决"数字暴政"的问题

discrete transistors didn’t solve the Tyranny of Numbers.

虽然更快更小 \N 但晶体管的出现 还是没有解决"数字暴政"的问题

It was getting unwieldy to design,

有几十万个独立元件的计算机不但难设计 \N 而且难生产

let alone physically manufacture computers with hundreds of thousands of individual components.

有几十万个独立元件的计算机不但难设计 \N 而且难生产

By the the 1960s, this was reaching a breaking point.

1960 年代，这个问题的严重性达到顶点 \N 电脑内部常常一大堆电线缠绕在一起

The insides of computers were often just huge tangles of wires.

1960 年代，这个问题的严重性达到顶点 \N 电脑内部常常一大堆电线缠绕在一起

Just look at what the inside of a PDP-8 from 1965 looked like!

看看这个 1965 年 PDP-8 计算机的内部

The answer was to bump up a new level of abstraction, and package up underlying complexity!

解决办法是引入一层新抽象，封装复杂性

The breakthrough came in 1958, when Jack Kilby, working at Texas Instruments,

突破性进展在 1958 年 \N 当时 Jack Killby 在德州仪器工作

demonstrated such an electronic part, "wherein all the components of the electronic circuit are completely integrated.

演示了一个电子部件：\N "电路的所有组件都集成在一起"

Put simply: instead of building computer parts out of many discrete components

简单说就是：\N 与其把多个独立部件用电线连起来，拼装出计算机

and wiring them all together,

简单说就是：\N 与其把多个独立部件用电线连起来，拼装出计算机

you put many components together, inside of a new, single component.

我们把多个组件包在一起，变成一个新的独立组件

These are called Integrated Circuits, or ICs.

这就是 集成电路（IC）

A few months later in 1959, Fairchild Semiconductor, lead by Robert Noyce, made ICs practical.

几个月后，在1959年 Robert Noyce 的仙童半导体 \N 让集成电路变为现实

Kilby built his ICs out of germanium, a rare and unstable material.

Kilby 用锗来做集成电路，锗很稀少而且不稳定

But, Fairchild used the abundant silicon, which makes up about a quarter of the earth's crust!

仙童半导体公司用硅 \N 硅的蕴藏量丰富，占地壳四分之一

It’s also more stable, therefore more reliable.

也更稳定可靠

For this reason, Noyce is widely regarded as the father of modern ICs,

所以 Noyce 被公认为现代集成电路之父

ushering in the electronics era... and also Silicon Valley, where Fairchild was based

开创了电子时代，创造了硅谷（仙童公司所在地）

and where many other semiconductor companies would soon pop up.

之后有很多半导体企业都出现在硅谷

In the early days, an IC might only contain a simple circuit with just a few transistors,

起初，一个 IC 只有几个晶体管 \N 例如这块早期样品，由西屋公司制造

like this early Westinghouse example.

起初，一个 IC 只有几个晶体管 \N 例如这块早期样品，由西屋公司制造

But even this allowed simple circuits, like the logic gates from Episode 3,

即使只有几个晶体管 \N 也可以把简单电路，第 3 集的逻辑门，能封装成单独组件

to be packaged up into a single component.

即使只有几个晶体管 \N 也可以把简单电路，第 3 集的逻辑门，能封装成单独组件

ICs are sort of like lego for computer engineers

IC 就像电脑工程师的乐高积木 \N 可以组合出无数种设计

"building blocks" that can be arranged into an infinite array of possible designs.

IC 就像电脑工程师的乐高积木 \N 可以组合出无数种设计

However, they still have to be wired together at some point

但最终还是需要连起来， \N 创造更大更复杂的电路，比如整个计算机

to create even bigger and more complex circuits, like a whole computer.

但最终还是需要连起来， \N 创造更大更复杂的电路，比如整个计算机

For this, engineers had another innovation: printed circuit boards, or PCB

所以工程师们再度创新：印刷电路板，简称 PCB

Instead of soldering and bundling up bazillions of wires, PCBs, which could be mass manufactured,

PCB 可以大规模生产，无需焊接或用一大堆线. \N 它通过蚀刻金属线的方式，把零件连接到一起

have all the metal wires etched right into them to connect components together.

PCB 可以大规模生产，无需焊接或用一大堆线. \N 它通过蚀刻金属线的方式，把零件连接到一起

By using PCBs and ICs together, one could achieve exactly the same functional circuit

把 PCB 和 IC 结合使用 \N 可以大幅减少独立组件和电线，但做到相同的功能

as that made from discrete components,

把 PCB 和 IC 结合使用 \N 可以大幅减少独立组件和电线，但做到相同的功能

but with far fewer individual components and tangled wires.

把 PCB 和 IC 结合使用 \N 可以大幅减少独立组件和电线，但做到相同的功能

Plus, it’s smaller, cheaper and more reliable.

而且更小，更便宜，更可靠. 三赢！

Triple win!

而且更小，更便宜，更可靠. 三赢！

Many early ICs were manufactured using teeny tiny discrete components

许多早期 IC 都是把很小的分立元件 \N 封装成一个独立单元，例如这块 1964 年的IBM样品

packaged up as a single unit, like this IBM example from 1964.

许多早期 IC 都是把很小的分立元件 \N 封装成一个独立单元，例如这块 1964 年的IBM样品

However, even when using really really itty-bitty components,

不过，即使组件很小 \N 塞5个以上的晶体管还是很困难

it was hard to get much more than around five transistors onto a single IC.

不过，即使组件很小 \N 塞5个以上的晶体管还是很困难

To achieve more complex designs, a radically different fabrication process was needed that

为了实现更复杂的设计，需要全新的制作工艺 \N "光刻"登场！

changed everything: Photolithography!

为了实现更复杂的设计，需要全新的制作工艺 \N "光刻"登场！

In short, it’s a way to use light to transfer complex patterns to a material, like a semiconductor

简单说就是 \N用光把复杂图案印到材料上，比如半导体

It only has a few basic operations, but these can be used to create incredibly complex circuits.

它只有几个基础操作，但可以制作出复杂电路

Let’s walk through a simple, although extensive example, to make one of these!

下面用一个简单例子，来做一片这个！

We start with a slice of silicon, which, like a thin cookie, is called a wafer.

我们从一片硅开始，叫"晶圆" \N 长得像薄饼干一样

Delicious!

美味！

Silicon, as we discussed briefly in episode 2, is special because it’s a semiconductor,

我们在第 2 集讨论过 \N 硅很特别，它是半导体

that is, a material that can sometimes conduct electricity and other times does not.

它有时导电，有时不导电 \N 我们可以控制导电时机

We can control where and when this happens,

它有时导电，有时不导电 \N 我们可以控制导电时机

making Silicon the perfect raw material for making transistors.

所以硅是做晶体管的绝佳材料

We can also use a wafer as a base to lay down complex metal circuits, so everything is integrated,

我们可以用晶圆做基础 \N 把复杂金属电路放上面，集成所有东西

perfect for... integrated circuits!

非常适合做.. 集成电路!

The next step is to add a thin oxide layer on top of the silicon,

下一步是在硅片顶部 \N 加一层薄薄的氧化层, 作为保护层

which acts as a protective coating.

下一步是在硅片顶部 \N 加一层薄薄的氧化层, 作为保护层

Then, we apply a special chemical called a photoresist.

然后加一层特殊化学品, 叫 "光刻胶" \N 光刻胶被光照射后 会变得可溶

When exposed to light, the chemical changes, and becomes soluble,

然后加一层特殊化学品, 叫 "光刻胶" \N 光刻胶被光照射后 会变得可溶

so it can be washed away with a different special chemical.

可以用一种特殊化学药剂洗掉

Photoresists aren’t very useful by themselves,

单单光刻胶本身，并不是很有用 \N 但和"光掩膜"配合使用会很强大

but are super powerful when used in conjunction with a photomask.

单单光刻胶本身，并不是很有用 \N 但和"光掩膜"配合使用会很强大

This is just like a piece of photographic film, but instead of a photo of

光掩膜就像胶片一样，只不过不是 \N 吃墨西哥卷饼的可爱仓鼠，而是要转移到晶圆上的图案

a hamster eating a tiny burrito, it contains a pattern to be transferred onto the wafer.

光掩膜就像胶片一样，只不过不是 \N 吃墨西哥卷饼的可爱仓鼠，而是要转移到晶圆上的图案

We do this by putting a photomask over the wafer, and turning on a powerful light.

把光掩膜盖到晶圆上，用强光照射 \N 挡住光的地方，光刻胶不会变化

Where the mask blocks the light, the photoresist is unchanged.

把光掩膜盖到晶圆上，用强光照射 \N 挡住光的地方，光刻胶不会变化

But where the light does hit the photoresist it changes chemically ，

光照到的地方，光刻胶会发生化学变化 \N 洗掉它之后，暴露出氧化层

which lets us wash away only the photoresist that was exposed to light, selectively revealing areas of our oxide layer.

光照到的地方，光刻胶会发生化学变化 \N 洗掉它之后，暴露出氧化层

Now, by using another special chemical, often an acid, we can remove any exposed oxide,

用另一种化学物质 - 通常是一种酸 \N 可以洗掉"氧化层"露出的部分, 蚀刻到硅层

and etch a little hole the entire way down to the raw silicon.

用另一种化学物质 - 通常是一种酸 \N 可以洗掉"氧化层"露出的部分, 蚀刻到硅层

Note that the oxide layer under the photoresist is protected.

注意，氧化层被光刻胶保护住了.

To clean up, we use yet another special chemical that washes away any remaining photoresist.

为了清理光刻胶，我们用另一种化学药品洗掉它

Yep, there are a lot of special chemicals in photolithography,

是的，光刻法用很多化学品，每种都有特定用途

each with a very specific function!

是的，光刻法用很多化学品，每种都有特定用途

So now we can see the silicon again,

现在硅又露出来了 \N 我们想修改硅露出来的区域 让它导电性更好

we want to modify only the exposed areas to better conduct electricity.

现在硅又露出来了 \N 我们想修改硅露出来的区域 让它导电性更好

To do that, we need to change it chemically through a process called: doping.

所以用一种化学过程来改变它，叫"掺杂"

I’m not even going to make a joke. Let’s move on.

不是开玩笑！我们继续

Most often this is done with a high temperature gas, something like Phosphorus,

"掺杂" 通常用高温气体来做，比如磷 \N 渗透进暴露出的硅，改变电学性质

which penetrates into the exposed area of silicon.

"掺杂" 通常用高温气体来做，比如磷 \N 渗透进暴露出的硅，改变电学性质

This alters its electrical properties.

"掺杂" 通常用高温气体来做，比如磷 \N 渗透进暴露出的硅，改变电学性质

We’re not going to wade into the physics and chemistry of semiconductors,

半导体的具体物理和化学性质我们不会深究,

but if you’re interested, there’s a link in the description to an excellent video

如果你感兴趣，描述里有个视频链接 \N 视频制作者是 Derek Muller 他的频道叫 Veritasium

by our friend Derek Muller from Veritasium.

如果你感兴趣，描述里有个视频链接 \N 视频制作者是 Derek Muller 他的频道叫 Veritasium

But, we still need a few more rounds of photolithography to build a transistor.

但我们还需要几轮光刻法 来做晶体管

The process essentially starts again, first by building up a fresh oxide layer ...

过程基本一样，先盖氧化层，再盖光刻胶

which we coat in photoresist.

过程基本一样，先盖氧化层，再盖光刻胶

Now, we use a photomask with a new and different pattern,

然后用新的光掩膜，这次图案不同 \N 在掺杂区域上方开一个缺口

allowing us to open a small window above the doped area.

然后用新的光掩膜，这次图案不同 \N 在掺杂区域上方开一个缺口

Once again, we wash away remaining photoresist.

洗掉光刻胶

Now we dope, and avoid telling a hilarious joke, again, but with a different gas that

然后用另一种气体掺杂 \N 把一部分硅转成另一种形式

converts part of the silicon into yet a different form.

然后用另一种气体掺杂 \N 把一部分硅转成另一种形式

Timing is super important in photolithography in order to control things like doping diffusionand etch depth.

为了控制深度，时机很重要 \N 我们不想超过之前的区域

In this case, we only want to dope a little region nested inside the other.

为了控制深度，时机很重要 \N 我们不想超过之前的区域

Now we have all the pieces we need to create our transistor!

现在 所有需要的组件都有了

The final step is to make channels in the oxide layer

最后一步，在氧化层上做通道 \N 这样可以用细小金属导线，连接不同晶体管

so that we can run little metal wires to different parts of our transistor.

最后一步，在氧化层上做通道 \N 这样可以用细小金属导线，连接不同晶体管

Once more, we apply a photoresist, and use a new photomask to etch little channels.

再次用光刻胶和光掩膜 蚀刻出小通道

Now, we use a new process, called metalization,

现在用新的处理方法 叫"金属化" \N 放一层薄薄的金属，比如铝或铜

that allows us to deposit a thin layer of metal, like aluminium or copper.

现在用新的处理方法 叫"金属化" \N 放一层薄薄的金属，比如铝或铜

But we don’t want to cover everything in metal.

但我们不想用金属盖住所有东西 \N 我们想蚀刻出具体的电路

We want to etch a very specific circuit design.

但我们不想用金属盖住所有东西 \N 我们想蚀刻出具体的电路

So, very similar to before, we apply a photoresist, use a photomask, dissolve the exposed resist,

所以又是类似的步骤 \N 用光刻胶+光掩膜，然后溶掉暴露的光刻胶，暴露的金属

and use a chemical to remove any exposed metal.

所以又是类似的步骤 \N 用光刻胶+光掩膜，然后溶掉暴露的光刻胶，暴露的金属

Whew!

咻~

Our transistor is finally complete!

晶体管终于做好了！ \N 它有三根线，连接着硅的三个不同区域

It has three little wires that connect to three different parts of the silicon

晶体管终于做好了！ \N 它有三根线，连接着硅的三个不同区域

each doped a particular way to create, in this example, what’s called a bipolar junction transistor.

每个区域的掺杂方式不同，这叫双极型晶体管

Here’s the actual patent from 1962, an invention that changed our world forever!

这个 1962 年的真实专利，永远改变了世界

Using similar steps, photolithography can create other useful electronic elements, like

用类似步骤，光刻可以制作其他电子元件 \N 比如电阻和电容，都在一片硅上

resistors and capacitors, all on a single piece of silicon

用类似步骤，光刻可以制作其他电子元件 \N 比如电阻和电容，都在一片硅上

plus all the wires needed to hook them up into circuits

而且互相连接的电路也做好了

Goodbye discrete components!

再见了，分立元件！

In our example, we made one transistor, but in the real world,

之前的例子 只做了一个晶体管，但现实中 \N 光刻法一次会做上百万个细节

photomasks lay down millions of little details all at once.

之前的例子 只做了一个晶体管，但现实中 \N 光刻法一次会做上百万个细节

Here is what an IC might look like from above, with wires crisscrossing above and below each other,

芯片放大是这样的，导线上下交错，连接各个元件

interconnecting all the individual elements together into complex circuits.

芯片放大是这样的，导线上下交错，连接各个元件

Although we could create a photomask for an entire wafer,

尽管可以把光掩膜投影到一整片晶圆上 \N 但光可以投射成任意大小

we can take advantage of the fact that light can be focused and projected to any size we want.

尽管可以把光掩膜投影到一整片晶圆上 \N 但光可以投射成任意大小

In the same way that a film can be projected to fill an entire movie screen,

就像投影仪可以投满荧幕一样

we can focus a photomask onto a very small patch of silicon, creating incredibly fine details.

我们可以把光掩膜 \N 聚焦到极小的区域，制作出非常精细的细节

A single silicon wafer is generally used to create dozens of ICs.

一片晶圆可以做很多 IC \N 整块都做完后，可以切割然后包进微型芯片

Then, once you’ve got a whole wafer full, you cut them up and package them into microchips,

一片晶圆可以做很多 IC \N 整块都做完后，可以切割然后包进微型芯片

those little black rectangles you see in electronics all the time.

微型芯片就是在电子设备中那些小长方体

Just remember: at the heart of each of those chips is one of these small pieces of silicon.

记住，芯片的核心都是一小片 IC

As photolithography techniques improved, the size of transistors shrunk, allowing for greater densities.

随着光刻技术发展，晶体管变小 密度变高

At the start of the 1960s, an IC rarely contained more than 5 transistors,

1960 年代初，IC 很少超过 5 个晶体管，因为塞不下

they just couldn’t possibly fit.

1960 年代初，IC 很少超过 5 个晶体管，因为塞不下

But, by the mid 1960s, we were starting to see ICs with over 100 transistors on the market.

但 1960 年代中期 \N 市场上开始出现超过 100 个晶体管的 IC

In 1965, Gordon Moore could see the trend: that approximately every two years,

1965年，戈登·摩尔看到了趋势：每两年左右，

thanks to advances in materials and manufacturing, you could fit twice the number of transistors

得益于材料和制造技术的发展 \N 同样大小的空间，能塞进两倍数量的晶体管！

into the same amount of space.

得益于材料和制造技术的发展 \N 同样大小的空间，能塞进两倍数量的晶体管！

This is called Moore’s Law.

这叫 摩尔定律

The term is a bit of a misnomer though.

然而这个名字不太对 \N 因为它不是定律，只是一种趋势

It’s not really a law at all, more of a trend.

然而这个名字不太对 \N 因为它不是定律，只是一种趋势

But it’s a good one.

但它是对的

IC prices also fell dramatically, from an average of $50 in 1962 to around $2 in 1968.

芯片的价格也急剧下降 \N 1962年平均50美元，下降到1968年2美元左右

Today, you can buy ICs for cents.

如今 几美分就能买到 IC

Smaller transistors and higher densities had other benefits too.

晶体管更小密度更高 还有其他好处

The smaller the transistor, the less charge you have to move around, allowing it to switch

晶体管越小，要移动的电荷量就越少 \N 能更快切换状态 耗电更少

states faster and consume less power.

晶体管越小，要移动的电荷量就越少 \N 能更快切换状态 耗电更少

Plus, more compact circuits meant less delay in signals resulting in faster clock speeds.

电路更紧凑 还意味着信号延迟更低 \N 导致时钟速度更快

In 1968, Robert Noyce and Gordon Moore teamed up and founded a new company,

1968年，罗伯特·诺伊斯 和 戈登·摩尔 \N 联手成立了一家新公司

combining the words Integrated and Electronics...

结合 Intergrated(集成) 和 Electronics(电子) 两个词

Intel. the largest chip maker today.

取名 Intel， 如今最大的芯片制造商

The Intel 4004 CPU, from Episodes 7 and 8, was a major milestone.

Intel 4004 CPU, 在第 7, 8 集介绍过 \N 是个重要里程碑

Released in 1971, it was the first processor that shipped as an IC, what’s called a microprocessor,

发布于1971年 \N 是第一个用 IC 做的处理器，也叫微型处理器

because it was so beautifully small!

因为真的非常小！

It contained 2,300 transistors.

它有2300个晶体管

People marveled at the level of integration, an entire CPU in one chip,

人们惊叹于它的整合水平 \N 整个 CPU 在一个芯片里

which just two decades earlier would have filled an entire room using discrete components.

而仅仅 20 年前，用分立元件会占满整个屋子

This era of integrated circuits, especially microprocessors, ushered in the third generation of computing.

集成电路的出现 \N 尤其是用来做微处理器，开启了计算 3.0

And the Intel 4004 was just the start.

而 Intel 4004 只是个开始，CPU 晶体管数量爆发增长

CPU transistor count exploded!

而 Intel 4004 只是个开始，CPU 晶体管数量爆发增长

By 1980, CPUs contained 30 thousand transistors.

1980年，3 万晶体管 \N 1990年，100 万晶体管

By 1990, CPUs breached the 1 million transistor count.

1980年，3 万晶体管 \N 1990年，100 万晶体管

By 2000, 30 million transistors,

2000年，3000 万个晶体管

and by 2010, ONE. BILLION. TRANSISTORS. IN ONE. IC. OMG!

2010年，10亿个晶体管！\N 在一个芯片里！我的天啊！

To achieve this density, the finest resolution possible with photolithography has improved

为了达到这种密度，光刻的分辨率 \N 从大约一万纳米，大概是人类头发直径的 1/10

from roughly 10 thousand nanometers, that’s about 1/10th the thickness of a human hair,

为了达到这种密度，光刻的分辨率 \N 从大约一万纳米，大概是人类头发直径的 1/10

to around 14 nanometers today.

发展到如今的 14 纳米 \N 比血红细胞小 400 倍！

That’s over 400 times smaller than a red blood cell!

发展到如今的 14 纳米 \N 比血红细胞小 400 倍！

And of course, CPU’s weren’t the only components to benefit.

当然，CPU 不是唯一受益的元件

Most electronics advanced essentially exponentially:

大多数电子器件都在指数式发展：\N 内存，显卡，固态硬盘，摄像头感光元件，等等

RAM, graphics cards, solid state hard drives, camera sensors, you name it.

大多数电子器件都在指数式发展：\N 内存，显卡，固态硬盘，摄像头感光元件，等等

Today’s processors, like the A10 CPU inside Of an iPhone 7, contains a mind melting 3.3 BILLION

如今的处理器，比如 iPhone 7 的 A10 CPU \N 有33亿个晶体管

transistors in an IC roughly 1cm by 1cm.

面积仅有 1cm x 1cm，比一张邮票还小

That’s smaller than a postage stamp!

面积仅有 1cm x 1cm，比一张邮票还小

And modern engineers aren’t laying out these designs by hand, one transistor at a time

现代工程师设计电路时，当然不是手工一个个设计晶体管 \N 这不是人力能做到的

- it’s not humanly possible.

现代工程师设计电路时，当然不是手工一个个设计晶体管 \N 这不是人力能做到的

Starting in the 1970’s, very-large-scale integration, or VLSI software, has been used

1970年代开始，超大规模集成(VLSI)软件 \N 用来自动生成芯片设计

to automatically generate chip designs instead.

1970年代开始，超大规模集成(VLSI)软件 \N 用来自动生成芯片设计

Using techniques like logic synthesis, where whole, high-level components can be laid down,like a memory cache

用比如 "逻辑综合" 这种技术 \N 可以放一整个高级组件，比如内存缓存

the software generates the circuit in the most efficient way possible.

软件会自动生成电路，做到尽可能高效

Many consider this to be the start of fourth generation computers.

许多人认为这是计算 4.0 的开始

Unfortunately, experts have been predicting the end of Moore’s Law for decades

坏消息是，专家们几十年来 \N 一直在预言摩尔定律的终结

and we might finally be getting close to it.

现在可能终于接近了

There are two significant issues holding us back from further miniaturization.

进一步做小，会面临 2 个大问题

First, we’re bumping into limits on how fine we can make features on a photomask and

1. 用光掩膜把图案弄到晶圆上 \N 因为光的波长，精度已达极限

it’s resultant wafer due to the wavelengths of light used in photolithography.

1. 用光掩膜把图案弄到晶圆上 \N 因为光的波长，精度已达极限

In response, scientists have been developing light sources with smaller and smaller wavelengths

所以科学家在研制波长更短的光源，投射更小的形状

that can project smaller and smaller features.

所以科学家在研制波长更短的光源，投射更小的形状

The second issue is that when transistors get really really small, where electrodes

2. 当晶体管非常小，电极之间可能只距离几个原子 \N 电子会跳过间隙，这叫：量子隧道贯穿

might be separated by only a few dozen atoms, electrons can jump the gap, a phenomenon called

2. 当晶体管非常小，电极之间可能只距离几个原子 \N 电子会跳过间隙，这叫：量子隧穿效应

quantum tunneling.

2. 当晶体管非常小，电极之间可能只距离几个原子 \N 电子会跳过间隙，这叫：量子隧穿效应

If transistors leak current, they don’t make very good switches.

如果晶体管漏电，就不是好开关

Nonetheless, scientists and engineers are hard at work figuring out ways around these problems.

科学家和工程师在努力找解决方法

Transistors as small as 1 nanometer have been demonstrated in research labs.

实验室中已造出小至1纳米的晶体管

Whether this will ever be commercially feasible remains MASKED in mystery.

能不能商业量产依然未知，未来也许能解决

But maybe we’ll be able to RESOLVE it in the future.

能不能商业量产依然未知，未来也许能解决

I’m DIEING to know. See you next week.

我非常期待！下周见！

This episode is supported by Hover.

本集由 Hover 赞助播出

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Computers in the 1940s and early 50s ran one program at a time.

1940,1950 年代的电脑，每次只能运行一个程序

A programmer would write one at their desk, for example, on punch cards.

程序员在打孔纸卡上写程序

Then, they'd carry it to a room containing a room-sized computer,

然后拿到一个计算机房间, 交给操作员

and hand it to a dedicated computer operator.

然后拿到一个计算机房间, 交给操作员

That person would then feed the program into the computer when it was next available.

等计算机空下来了，操作员会把程序放入

The computer would run it, spit out some output, and halt.

然后运行，输出结果，停机

This very manual process worked OK back when computers were slow,

以前计算机慢，这种手动做法可以接受

and running a program often took hours, days or even weeks.

运行一个程序通常要几小时，几天甚至几周

But, as we discussed last episode,

但上集说过，计算机越来越快，越来越快

computers became faster... and faster... and faster

但上集说过，计算机越来越快，越来越快

exponentially so!

指数级增长！

Pretty soon, having humans run around and inserting programs into readers

很快，放程序的时间 比程序运行时间还长

was taking longer than running the actual programs themselves.

很快，放程序的时间 比程序运行时间还长

We needed a way for computers to operate themselves,

我们需要一种方式 让计算机自动运作

and so, operating systems were born.

于是"操作系统"诞生了

Operating systems , or OS for short, are just programs.

操作系统，简称 OS，其实也是程序

But, special privileges on the hardware let them run and manage other programs.

但它有操作硬件的特殊权限 \N 可以运行和管理其它程序

They're typically the first one to start when a computer is turned on,

操作系统一般是开机第一个启动的程序

and all subsequent programs are launched by the OS.

其他所有程序 都由操作系统启动

They got their start in the 1950s,

操作系统开始于 1950 年代 \N 那时计算机开始变得更强大更流行

as computers became more widespread and more powerful.

操作系统开始于 1950 年代 \N 那时计算机开始变得更强大更流行

The very first OS augmented the mundane, manual task of loading programs by hand.

第一个操作系统 加强了程序加载方式

Instead of being given one program at a time,

之前只能一次给一个程序，现在可以一次多个

computers could be given batches.

之前只能一次给一个程序，现在可以一次多个

When the computer was done with one,

当计算机运行完一个程序，会自动运行下一个程序

it would automatically and near-instantly start the next.

当计算机运行完一个程序，会自动运行下一个程序

There was no downtime while someone scurried around an office to find the next program to run.

这样就不会浪费时间，找下一个程序的纸卡

This was called batch processing .

这叫 批处理

While computers got faster, they also got cheaper.

电脑变得更快更便宜，开始在出现在世界各地

So, they were popping up all over the world,

电脑变得更快更便宜，开始在出现在世界各地

especially in universities and government offices.

特别是大学和政府办公室

Soon, people started sharing software.

很快，人们开始分享软件，但有一个问题

But there was a problem

很快，人们开始分享软件，但有一个问题

In the era of one-off computers, like the Harvard Mark 1 or ENIAC,

在哈佛1号和 ENIAC 那个时代，计算都是一次性的

programmers only had to write code for that one single machine.

程序员只需要给那"一台"机器写代码

The processor, punch card readers, and printers were known and unchanging.

处理器，读卡器，打印机都是已知的

But as computers became more widespread,

但随着电脑越来越普遍，计算机配置并不总是相同的

their configurations were not always identical,

但随着电脑越来越普遍，计算机配置并不总是相同的

like computers might have the same CPU, but not the same printer.

比如计算机可能有相同 CPU，但不同的打印机

This was a huge pain for programmers.

这对程序员很痛苦

Not only did they have to worry about writing their program,

不仅要担心写程序，还要担心程序怎么和不同型号打印机交互

but also how to interface with each and every model of printer,

不仅要担心写程序，还要担心程序怎么和不同型号打印机交互

and all devices connected to a computer, what are called peripherals.

以及计算机连着的其他设备，这些统称"外部设备"

Interfacing with early peripherals was very low level,

和早期的外部设备交互，是非常底层的

requiring programmers to know intimate hardware details about each device.

程序员要了解设备的硬件细节

On top of that, programmers rarely had access to every model of a peripheral to test their code on.

加重问题的是，程序员很少能拿到所有型号的设备来测代码

So, they had to write code as best they could, often just by reading manuals,

所以一般是阅读手册来写代码，祈祷能正常运行

and hope it worked when shared.

所以一般是阅读手册来写代码，祈祷能正常运行

Things weren't exactly plug-and-play

现在是"即插即用"，以前是"祈祷能用"

back then more plug-and-pray.

现在是"即插即用"，以前是"祈祷能用"

This was clearly terrible,

这很糟糕，所以为了程序员写软件更容易

so to make it easier for programmers,

这很糟糕，所以为了程序员写软件更容易

Operating Systems stepped in as intermediaries between software programs and hardware peripherals.

操作系统充当软件和硬件之间的媒介

More specifically, they provided a software abstraction, through APIs,

更具体地说，操作系统提供 API 来抽象硬件，叫"设备驱动程序"

called device drivers .

更具体地说，操作系统提供 API 来抽象硬件，叫"设备驱动程序"

These allow programmers to talk to common input and output hardware,

程序员可以用标准化机制 和输入输出硬件（I/O）交互

or I/O for short, using standardized mechanisms.

程序员可以用标准化机制 和输入输出硬件（I/O）交互

For example, programmers could call a function like "print highscore",

比如，程序员只需调用 print(highscore)

and the OS would do the heavy lifting to get it onto paper.

操作系统会处理 输到纸上的具体细节

By the end of the 1950s, computers had gotten so fast,

到 1950 年代尾声，电脑已经非常快了

they were often idle waiting for slow mechanical things, like printers and punch card readers.

处理器经常闲着，等待慢的机械设备（比如打印机和读卡器）

While programs were blocked on I/O,

程序阻塞在 I/O 上

the expensive processor was just chillin'... not like a villain

而昂贵的处理器则在度假，就是放松啥也不做

you know, just relaxing.

而昂贵的处理器则在度假，就是放松啥也不做

In the late 50's, the University of Manchester, in the UK,

50年代后期，英国曼彻斯特大学 \N 开始研发世界上第一台超级计算机，Atlas

started work on a supercomputer called Atlas, one of the first in the world.

50年代后期，英国曼彻斯特大学 \N 开始研发世界上第一台超级计算机，Atlas

They knew it was going to be wicked fast,

他们知道机器会超级快，所以需要一种方式来最大限度的利用它

so they needed a way to make maximal use of the expensive machine.

他们知道机器会超级快，所以需要一种方式来最大限度的利用它

Their solution was a program called the Atlas Supervisor, finished in 1962.

他们的解决方案是一个程序叫 Atlas Supervisor \N 于1962年完成

This operating system not only loaded programs automatically, like earlier batch systems,

这个操作系统 \N不仅像更早期的批处理系统那样，能自动加载程序

but could also run several at the same time on its single CPU.

还能在单个 CPU 上同时运行几个程序

It did this through clever scheduling.

它通过调度来做到这一点

Let's say we have a game program running on Atlas,

假设 Atlas 上有一个游戏在运行

and we call the function "print(highscore)"

并且我们调用一个函数 print(highscore)

which instructs Atlas to print the value of a variable named "highscore"

它让 Atlas 打印一个叫 highscore 的变量值

onto paper to show our friends that we're the ultimate champion of virtual tiddlywinks.

让朋友知道 我是最高分冠军

That function call is going to take a while, the equivalent of thousands of clock cycles,

print 函数运行需要一点时间，大概上千个时钟周期

because mechanical printers are slow in comparison to electronic CPUs.

但因为打印机比 CPU 慢，与其等着它完成操作

So instead of waiting for the I/O to finish,

但因为打印机比 CPU 慢，与其等着它完成操作

Atlas instead puts our program to sleep,

Atlas 会把程序休眠，运行另一个程序

then selects and runs another program that's waiting and ready to run.

Atlas 会把程序休眠，运行另一个程序

Eventually, the printer will report back to Atlas that it finished printing the value of "highscore".

最终, 打印机会告诉 Atlas, 打印已完成

Atlas then marks our program as ready to go,

Atlas 会把程序标记成可继续运行

and at some point, it will be scheduled to run again on the CPU,

之后在某时刻会安排给 CPU 运行

and continue onto the next line of code following the print statement.

并继续 print 语句之后的下一行代码

In this way, Atlas could have one program running calculations on the CPU,

这样, Atlas 可以在 CPU 上运行一个程序

while another was printing out data,

同时另一个程序在打印数据

and yet another reading in data from a punch tape.

同时另一个程序读数据

Atlas' engineers doubled down on this idea,

Atlas 的工程师做的还要多，配了4台纸带读取器，4台纸带打孔机

and outfitted their computer with 4 paper tape readers, 4 paper tape punches,

Atlas 的工程师做的还要多，配了4台纸带读取器，4台纸带打孔机

and up to 8 magnetic tape drives.

多达8个磁带驱动器

This allowed many programs to be in progress all at once,

使多个程序可以同时运行，在单个 CPU 上共享时间

sharing time on a single CPU.

使多个程序可以同时运行，在单个 CPU 上共享时间

This ability, enabled by the Operating System, is called

操作系统的这种能力叫"多任务处理"

multitasking.

操作系统的这种能力叫"多任务处理"

There's one big catch to having many programs running simultaneously on a single computer, though.

同时运行多个程序有个问题

Each one is going to need some memory,

每个程序都会占一些内存 \N 当切换到另一个程序时，我们不能丢失数据

and we can't lose that program's data when we switch to another program.

每个程序都会占一些内存 \N 当切换到另一个程序时，我们不能丢失数据

The solution is to allocate each program its own block of memory.

解决办法是 给每个程序分配专属内存块

So, for example, let's say a computer has 10,000 memory locations in total.

举个例子，假设计算机一共有 10000 个内存位置

Program A might get allocated memory addresses 0 through 999,

程序 A 分配到内存地址 0 到 999

and Program B might get 1000 through 1999, and so on.

而程序 B 分配到内存地址 1000 到 1999，以此类推

If a program asks for more memory,

如果一个程序请求更多内存，操作系统会决定是否同意

the operating system decides if it can grant that request,

如果一个程序请求更多内存，操作系统会决定是否同意

and if so, what memory block to allocate next.

如果同意，分配哪些内存块

This flexibility is great, but introduces a quirk.

这种灵活性很好，但带来一个奇怪的后果

It means that Program A could end up being allocated non-sequential blocks of memory,

程序 A 可能会分配到非连续的内存块

in say addresses 0 through 999, and 2000 through 2999.

比如内存地址 0 到 999，以及 2000 到 2999

And this is just a simple example

这只是个简单例子

- a real program might be allocated dozens of blocks scattered all over memory.

真正的程序可能会分配到内存中数十个地方

As you might imagine,

你可能想到了，这对程序员来说很难跟踪

this would get really confusing for programmers to keep track of.

你可能想到了，这对程序员来说很难跟踪

Maybe there's a long list of sales data in memory that

也许内存里有一长串销售额，每天下班后要算销售总额

a program has to total up at the end of the day,

也许内存里有一长串销售额，每天下班后要算销售总额

but this list is stored across a bunch of different blocks of memory.

但列表 存在一堆不连续的内存块里

To hide this complexity, Operating Systems virtualize memory locations.

为了隐藏这种复杂性，操作系统会把内存地址进行 "虚拟化"

With Virtual Memory, programs can assume their memory always starts at address 0,

这叫 "虚拟内存"，程序可以假定内存总是从地址0开始

keeping things simple and consistent.

简单又一致

However, the actual, physical location in computer memory

而实际物理位置 被操作系统隐藏和抽象了

is hidden and abstracted by the operating system.

而实际物理位置 被操作系统隐藏和抽象了

Just a new level of abstraction.

一层新的抽象

Let's take our example Program B,

用程序 B 来举例 \N 它被分配了内存地址 1000 到 1999

which has been allocated a block of memory from address 1000 to 1999.

用程序 B 来举例 \N 它被分配了内存地址 1000 到 1999

As far as Program B can tell, this appears to be a block from 0 to 999.

对程序 B 而言，它看到的地址是 0 到 999

The OS and CPU handle the virtual-to-physical memory remapping automatically.

操作系统会自动处理 \N 虚拟内存和物理内存之间的映射

So, if Program B requests memory location 42,

如果程序 B 要地址 42，实际上是物理地址 1042

it really ends up reading address 1042.

如果程序 B 要地址 42，实际上是物理地址 1042

This virtualization of memory addresses is even more useful for Program A,

这种内存地址的虚拟化 对程序 A 甚至更有用

which in our example, has been allocated two blocks of memory

在例子中，A 被分配了两块隔开的内存

that are separated from one another.

在例子中，A 被分配了两块隔开的内存

This too is invisible to Program A.

程序 A 不知道这点.

As far as it can tell, it's been allocated a continuous block of 2000 addresses.

以 A 的视角，它有 2000 个连续地址

When Program A reads memory address 999,

当程序 A 读内存地址 999 时 \N 会刚好映射到物理内存地址 999

that does coincidentally map to physical memory address 999.

当程序 A 读内存地址 999 时 \N 会刚好映射到物理内存地址 999

But if Program A reads the very next value in memory, at address 1000,

但如果程序 A 读下一个地址 1000

that gets mapped behind the scenes to physical memory address 2000.

会映射到物理地址 2000

This mechanism allows programs to have flexible memory sizes,

这种机制使程序的内存大小可以灵活增减 \N 叫"动态内存分配"

called dynamic memory allocation,

这种机制使程序的内存大小可以灵活增减 \N 叫"动态内存分配"

that appear to be continuous to them.

对程序来说，内存看起来是连续的.

It simplifies everything and offers tremendous flexibility to the Operating System

它简化了一切，为操作系统同时运行多个程序 \N 提供了极大的灵活性

in running multiple programs simultaneously.

它简化了一切，为操作系统同时运行多个程序 \N 提供了极大的灵活性

Another upside of allocating each program its own memory,

给程序分配专用的内存范围，\N 另一个好处是 这样隔离起来会更好

is that they're better isolated from one another.

给程序分配专用的内存范围，\N 另一个好处是 这样隔离起来会更好

So, if a buggy program goes awry, and starts writing gobbledygook,

如果一个程序出错，开始写乱七八糟的数据

it can only trash its own memory, not that of other programs.

它只能捣乱自己的内存，不会影响到其它程序.

This feature is called Memory Protection.

这叫 "内存保护"

This is also really useful in protecting against malicious software, like viruses.

防止恶意软件（如病毒）也很有用

For example, we generally don't want other programs to have the ability

例如，我们不希望其他程序有能力 \N 读或改邮件程序的内存

to read or modify the memory of, let say, our email,

例如，我们不希望其他程序有能力 \N 读或改邮件程序的内存

with that kind of access,

如果有这种权限 \N 恶意软件可能以你的名义发邮件，甚至窃取个人信息

malware could send emails on your behalf and maybe steal personal information.

如果有这种权限 \N 恶意软件可能以你的名义发邮件，甚至窃取个人信息

Not good!

一点都不好！

Atlas had both virtual and protected memory.

Atlas 既有"虚拟内存"也有"内存保护"

It was the first computer and OS to support these features!

是第一台支持这些功能的计算机和操作系统！

By the 1970s, computers were sufficiently fast and cheap.

到 1970 年代，计算机足够快且便宜

Institutions like a university could buy a computer and let students use it.

大学会买电脑让学生用

It was not only fast enough to run several programs at once,

计算机不仅能同时运行多个程序，还能让多用户能同时访问

but also give several users simultaneous, interactive access.

计算机不仅能同时运行多个程序，还能让多用户能同时访问

This was done through a terminal,

多个用户用"终端"来访问计算机

which is a keyboard and screen that connects to a big computer,

"终端"只是键盘+屏幕，连到主计算机 \N 终端本身没有处理能力

but doesn't contain any processing power itself.

"终端"只是键盘+屏幕，连到主计算机 \N 终端本身没有处理能力

A refrigerator-sized computer might have 50 terminals connected to it,

冰箱大小的计算机可能有50个终端，能让50个用户使用

allowing up to 50 users.

冰箱大小的计算机可能有50个终端，能让50个用户使用

Now operating systems had to handle not just multiple programs,

这时操作系统不但要处理多个程序，还要处理多个用户

but also multiple users.

这时操作系统不但要处理多个程序，还要处理多个用户

So that no one person could gobble up all of a computer's resources,

为了确保其中一个人 不会占满计算机资源

operating systems were developed that offered time-sharing.

开发了 分时操作系统

With time-sharing each individual user was only allowed to utilize

意思是 每个用户只能用一小部分处理器，内存等

a small fraction of the computer's processor, memory, and so on.

意思是 每个用户只能用一小部分处理器，内存等

Because computers are so fast,

因为电脑很快 \N 即使拿到 1/50 的资源也足以完成许多任务

even getting just 1/50th of its resources was enough for individuals to complete many tasks.

因为电脑很快 \N 即使拿到 1/50 的资源也足以完成许多任务

The most influential of early time-sharing Operating Systems was

早期分时操作系统中，最有影响力的是 \N Multics（多任务信息与计算系统）

Multics, or Multiplexed Information and Computing Service,

早期分时操作系统中，最有影响力的是 \N Multics（多任务信息与计算系统）

released in 1969.

于 1969 年发布

Multics was the first major operating system designed to be secure from the outset.

Multics 是第一个，从设计时就考虑到安全的操作系统

Developers didn't want mischievous users accessing data they shouldn't,

开发人员不希望恶意用户 访问不该访问的数据

like students attempting to access the final exam on their professor's account.

比如学生假装成教授，访问期末考试的文件

Features like this meant Multics was really complicated for its time,

这导致 Multics 的复杂度超过当时的平均水准

using around 1 Megabit of memory, which was a lot back then!

操作系统会占大约 1 Mb 内存，这在当时很多！

That might be half of a computer's memory, just to run the OS!

可能是内存的一半，只拿来运行操作系统！

Dennis Ritchie, one of the researchers working on Multics, once said:

Multics 的研究人员之一 Dennis Ritchie 曾说过

"One of the obvious things that went wrong with Multics as a commercial success

"阻碍 Multics 获得商业成功的一个明显问题是

was just that it was sort of over-engineered in a sense.

从某种方面来说，它被过度设计了，功能太多了"

There was just too much in it."

从某种方面来说，它被过度设计了，功能太多了"

This lead Dennis, and another Multics researcher,

所以 Dennis 和另一个 Multics 研究员 \N Ken Thompson 联手打造新的操作系统

Ken Thompson, to strike out on their own and build a new, lean operating system

所以 Dennis 和另一个 Multics 研究员 \N Ken Thompson 联手打造新的操作系统

called Unix.

叫 Unix

They wanted to separate the OS into two parts:

他们想把操作系统分成两部分：

First was the core functionality of the OS,

首先是操作系统的核心功能

things like memory management, multitasking,and dealing with I/O,

如内存管理，多任务和输入/输出处理 \N 这叫"内核"

which is called the kernel .

如内存管理，多任务和输入/输出处理 \N 这叫"内核"

The second part was a wide array of useful tools that came bundled with,

第二部分是一堆有用的工具

but not part of the kernel, things like programs and libraries.

但它们不是内核的一部分（比如程序和运行库）

Building a compact, lean kernel meant intentionally leaving some functionality out.

紧凑的内核 意味着功能没有那么全面

Tom Van Vleck, another Multics developer, recalled:

Multics 的另一个开发者 Tom Van Vleck 回忆说：

"I remarked to Dennis that easily half the code I was writing in Multics was error recovery code."

"我对 Dennis 说，我在 Multics 写的一半代码都是错误恢复代码"

He said, "We left all that stuff out of Unix.

他说:"Unix 不会有这些东西

If there's an error, we have this routine called panic,

如果有错误发生，我们就让内核"恐慌"（panic）

and when it is called, the machine crashes,

当调用它时，机器会崩溃

and you holler down the hall, 'Hey, reboot it.'""

你得在走廊里大喊，"嘿，重启电脑"

You might have heard of kernel panics.

你可能听过 "内核恐慌"（kernel panic）

This is where the term came from.

这就是这个词的来源

It's literally when the kernel crashes, has no recourse to recover,

内核如果崩溃，没有办法恢复

and so calls a function called "panic".

所以调用一个叫"恐慌"（panic）的函数

Originally, all it did was print the word "panic" and then enter an infinite loop.

起初只是打印"恐慌"一词，然后无限循环

This simplicity meant that Unix could be run on cheaper and more diverse hardware,

这种简单性意味着 \N Unix 可以在更便宜更多的硬件上运行

making it popular inside Bell Labs, where Dennis and Ken worked.

使 Unix 在 Dennis 和 Ken 工作的 \N 贝尔实验室大受欢迎

As more developers started using Unix to build and run their own programs,

越来越多开发人员用 Unix 写程序和运行程序

the number of contributed tools grew.

工具数量日益增长

Soon after its release in 1971,

1971 年发布后不久

it gained compilers for different programming languages and even a word processor,

就有人写了不同编程语言的编译器 \N 甚至文字处理器

quickly making it one of the most popular OSes of the 1970s and 80s.

使得 Unix 迅速成为 \N 1970~80年代最流行的操作系统之一

At the same time, by the early 1980s,

到 1980 年代早期

the cost of a basic computer had fallen to the point where individual people could afford one,

计算机的价格 降到普通人买得起 \N 这些叫"个人电脑"或"家庭电脑"

called a personal or home computer.

计算机的价格 降到普通人买得起 \N 这些叫"个人电脑"或"家庭电脑"

These were much simpler than the big mainframes

这些电脑比大型主机简单得多 \N 主机一般在大学，公司和政府

found at universities, corporations, and governments.

这些电脑比大型主机简单得多 \N 主机一般在大学，公司和政府

So, their operating systems had to be equally simple.

因此操作系统也得简单

For example, Microsoft's Disk Operating System, or MS-DOS, was just 160 kilobytes,

举例，微软的磁盘操作系统（MS-DOS）只有 160 kB \N 一张磁盘就可以容纳

allowing it to fit, as the name suggests, onto a single disk.

举例，微软的磁盘操作系统（MS-DOS）只有 160 kB \N 一张磁盘就可以容纳

First released in 1981, it became the most popular OS for early home computers,

于 1981 年发布，成为早期家用电脑最受欢迎的操作系统

even though it lacked multitasking and protected memory.

虽然缺少"多任务"和"保护内存"这样功能

This meant that programs could,

意味着程序经常使系统崩溃

and would, regularly crash the system.

意味着程序经常使系统崩溃

While annoying, it was an acceptable tradeoff,

虽然很讨厌但还可以接受，因为用户可以重启

as users could just turn their own computers off and on again!

虽然很讨厌但还可以接受，因为用户可以重启

Even early versions of Windows,

哪怕是微软 1985 年发布的早期 Windows \N 虽然在 90 年代很流行

first released by Microsoft in 1985 and which dominated the OS scene throughout the 1990s,

哪怕是微软 1985 年发布的早期 Windows \N 虽然在 90 年代很流行

lacked strong memory protection.

但却缺乏"内存保护"

When programs misbehaved,

当程序行为不当时，就会"蓝屏"

you could get the blue screen of death,

当程序行为不当时，就会"蓝屏"

a sign that a program had crashed so badly that it took down the whole operating system.

代表程序崩溃的非常严重，把系统也带崩溃了

Luckily, newer versions of Windows have better protections and usually don't crash that often.

幸运的是，新版Windows有更好的保护，不会经常崩溃

Today, computers run modern operating systems,

如今的计算机 有现代操作系统

like Mac OS X, Windows 10, Linux, iOS and Android.

比如 Mac OS X，Windows 10 \NLinux，iOS和Android

Even though the computers we own are most often used by just a single person,

虽然大部分设备只有一个人使用

you!

你！

their OS all have multitasking and virtual and protected memory.

操作系统依然有"多任务, "虚拟内存", "内存保护"

So, they can run many programs at once:

因此可以同时运行多个程序：

you can watch YouTube in your web browser,

一边在浏览器看 YouTube，一边在 Photoshop 修图

edit a photo in Photoshop,

一边在浏览器看 YouTube，一边在 Photoshop 修图

play music in Spotify and sync Dropbox all at the same time.

用 Spotify 放音乐，同步 Dropbox

This wouldn't be possible without those decades of research and development on Operating Systems,

如果没有操作系统这几十年的发展，这些都不可能,

and of course the proper memory to store those programs.

当然，我们也需要地方放程序

Which we'll get to next week.

下周会讨论

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

We've talked about computer memory several times in this series,

系列中 我们多次谈到内存（Memory）

and we even designed some in Episode 6.

甚至在第 6 集设计了一个简单内存

In general, computer memory is non-permanent.

一般来说，电脑内存是 "非永久性"

If your xbox accidently gets unplugged and turns off,

如果 Xbox 电源线不小心拔掉了，内存里所有数据都会丢失

any data saved in memory is lost.

如果 Xbox 电源线不小心拔掉了，内存里所有数据都会丢失

For this reason, it's called volatile memory.

所以内存叫"易失性"存储器

What we haven't talked so much about this series is storage,

我们还没谈过的话题 是存储器（Storage）

which is a tad different.

存储器（Storage）和内存（Memory）有点不同

Any data written to storage, like your hard drive,

任何写入"存储器"的数据，比如你的硬盘 \N 数据会一直存着，直到被覆盖或删除，断电也不会丢失

will stay there until it's over-written or deleted, even if the power goes out.

任何写入"存储器"的数据，比如你的硬盘 \N 数据会一直存着，直到被覆盖或删除，断电也不会丢失

It's non-volatile.

存储器是"非易失性"的

It used to be that volatile memory was fast and non-volatile storage was slow,

以前是"易失性"的速度快，"非易失性"的速度慢

but as computing technologies have improved, this distinction is becoming less true,

但随着技术发展，两者的差异越来越小

and the terms have started to blend together.

但随着技术发展，两者的差异越来越小

Nowadays, we take for granted technologies like this little USB stick,

如今我们认为稀松平常的技术，比如这个 U 盘

which offers gigabytes of memory, reliable over long periods of time, all at low cost,

能低成本+可靠+长时间 存储上 GB 的数据

but this wasn't always true.

但以前可不是这样的

The earliest computer storage was paper punch cards,

最早的存储介质是 打孔纸卡 \N 以及纸卡的亲戚 打孔纸带

and its close cousin, punched paper tape.

最早的存储介质是 打孔纸卡 \N 以及纸卡的亲戚 打孔纸带

By the 1940s, punch cards had largely standardized into a grid of 80 columns and 12 rows,

到1940年代，纸卡标准是 80列x12行

allowing for a maximum of 960 bits of data to be stored on a single card.

一张卡能存 960 位数据 (80x12=960)

The largest program ever punched onto cards, that we know of,

据我们所知的 最大纸卡程序

was the US Military's Semi-Automatic Ground Environment, or SAGE,

是美国军方的"半自动地面防空系统" 简称 SAGE

an Air Defense System that became operational in 1958.

一个在 1958 年投入使用的防空系统

The main program was stored on 62,500 punchcards,

主程序存储在 62,500 个纸卡上

roughly equivalent to 5 megabytes of data,

大小 5MB 左右, 相当如今手机拍张照

that's the size of an average smartphone photo today.

大小 5MB 左右, 相当如今手机拍张照

Punch cards were a useful and popular form of storage for decades,

纸卡用了十几年，因为不用电而且便宜耐用

they didn't need power, plus paper was cheap and reasonably durable.

纸卡用了十几年，因为不用电而且便宜耐用

However, punchcards were slow and write-once,

然而坏处是读取慢，只能写入一次

you can't easily un-punch a hole.

打的孔无法轻易补上

So they were a less useful form of memory,

对于存临时值，纸卡不好用

where a value might only be needed for a fraction of a second during a program's execution,

对于存临时值，纸卡不好用

and then discarded.

对于存临时值，纸卡不好用

A faster, larger and more flexible form of computer memory was needed.

我们需要更快更大更灵活的存储方式

An early and practical approach was developed by J. Presper Eckert,

J. Presper Eckert 在 1944 年建造 ENIAC 时发明了一种方法

as he was finishing work on ENIAC in 1944.

J. Presper Eckert 在 1944 年建造 ENIAC 时发明了一种方法

His invention was called Delay Line Memory, and it worked like this.

叫"延迟线存储器"（Delay Line Memory）原理如下

You take a tube and fill it with a liquid, like mercury.

拿一个管子装满液体，如水银

Then, you put a speaker at one end and microphone at the other.

管子一端放扬声器，另一端放麦克风

When you pulse the speaker, it creates a pressure wave.

扬声器发出脉冲时 会产生压力波

This takes time to propagate to the other end of the tube,

压力波需要时间 传播到另一端的麦克风

where it hits the microphone,

压力波需要时间 传播到另一端的麦克风

converting it back into an electrical signal.

麦克风将压力波 转换回电信号.

And we can use this propagation delay to store data!

我们可以用压力波的传播延迟 来存储数据！

Imagine that the presence of a pressure wave is a 1

假设有压力波代表 1，没有代表 0

and the absence of a pressure wave is a 0.

假设有压力波代表 1，没有代表 0

Our speaker can output a binary sequence like 1010 0111.

扬声器可以输出 1​​010 0111

The corresponding waves will travel down the tube, in order,

压力波沿管子传播，过了一会儿，撞上麦克风，

and a little while later, hit the microphone,

压力波沿管子传播，过了一会儿，撞上麦克风，

which converts the signal back into 1's and 0's.

将信号转换回 1 和 0

If we create a circuit that connects the microphone to the speaker,

如果加一个电路，连接麦克风和扬声器

plus a little amplifier to compensate for any loss,

再加一个放大器（Amplifier）来弥补信号衰弱

we can create a loop that stores data.

就能做一个存储数据的循环

The signal traveling along the wire is near instantaneous,

信号沿电线传播几乎是瞬时的,

so there's only ever one bit of data showing at any moment in time.

所以任何时间点只显示 1 bit 数据

But in the tube, you can store many bits!

但管子中可以存储多个位(bit)

After working on ENIAC, Eckert and his colleague John Mauchly,

忙完 ENIAC 后，Eckert 和同事 John Mauchly

set out to build a bigger and better computer called EDVAC, incorporating Delay Line Memory.

着手做一个更大更好的计算机叫 EDVAC，使用了延迟线存储器

In total, the computer had 128 Delay Lines,

总共有 128 条延迟线，每条能存 352 位（bits）

each capable of storing 352 bits.

总共有 128 条延迟线，每条能存 352 位（bits）

That's a grand total of 45 thousands bits of memory,

总共能存 45,000 位(bit)

not too shabby for 1949!

对 1949 年来说还不错！

This allowed EDVAC to be one of the very earliest Stored-Program Computers,

这使得 EDVAC 成为最早的 "存储程序计算机" 之一

which we talked about in Episode 10.

我们在第 10 集讨论过

However, a big drawback with delay line memory

但"延迟线存储器"的一大缺点是

is that you could only read one bit of data from a tube at any given instant.

每一个时刻只能读一位 (bit) 数据

If you wanted to access a specific bit, like bit 112,

如果想访问一个特定的 bit，比如第 112 位(bit) \N 你得等待它从循环中出现

you'd have to wait for it to come around in the loop,

如果想访问一个特定的 bit，比如第 112 位(bit) \N 你得等待它从循环中出现

what's called sequential or cyclic-access memory,

所以又叫 "顺序存储器"或"循环存储器"

whereas we really want random access memory,

而我们想要的是 "随机存取存储器" \N 可以随时访问任何位置

where we can access any bit at any time.

而我们想要的是 "随机存取存储器" \N 可以随时访问任何位置

It also proved challenging to increase the density of the memory,

增加内存密度也是一个挑战

packing waves closer together meant they were more easily mixed up.

把压力波变得更紧密 意味着更容易混在一起

In response, new forms of delay line memory were invented,

所以出现了其他类型的 "延迟线存储器"

such as magnetostrictive delay lines .

如 "磁致伸缩延迟存储器"

These delay lines use a metal wire that could be twisted,

用金属线的振动来代表数据

creating little torsional waves that represented data.

用金属线的振动来代表数据

By forming the wire into a coil, you could store around 1000 bits in a 1 foot by 1 foot square.

通过把线卷成线圈，1英尺×1英尺的面积能存储大概 1000位(bit)

However, delay line memory was largely obsolete by the mid 1950s,

然而，延迟线存储器在 1950 年代中期就基本过时了

surpassed in performance, reliability and cost by a new kid on the block:

因为出现了新技术，性能,可靠性和成本都更好

magnetic core memory which was constructed out of little magnetic donuts,

"磁芯存储器"，用了像甜甜圈的小型磁圈

called cores.

"磁芯存储器"，用了像甜甜圈的小型磁圈

If you loop a wire around this core.

如果给磁芯绕上电线，并施加电流，可以将磁化在一个方向

and run an electrical current through the wire,

如果给磁芯绕上电线，并施加电流，可以将磁化在一个方向

we can magnetize the core in a certain direction.

如果给磁芯绕上电线，并施加电流，可以将磁化在一个方向

If we turn the current off, the core will stay magnetized.

如果关掉电流，磁芯保持磁化

If we pass current through the wire in the opposite direction,

如果沿相反方向施加电流

the magnetization direction, called polarity,

磁化的方向（极性）会翻转

flips the other way.

磁化的方向（极性）会翻转

In this way, we can store 1's and 0's!

这样就可以存 1 和 0！

1 bit of memory isn't very useful, so these little donuts were arranged into grids.

如果只存 1 位不够有用，所以把小甜甜圈排列成网格

There were wires for selecting the right row and column, and a wire that ran through every core,

有电线负责选行和列 \N 也有电线贯穿每个磁芯, 用于读写一位(bit)

which could be used to read or write a bit.

有电线负责选行和列 \N 也有电线贯穿每个磁芯, 用于读写一位(bit)

Here is an actual piece of core memory!

我手上有一块磁芯存储器

In each of these little yellow squares, there are 32 rows and 32 columns of tiny cores,

每个黄色方格 有32行x32列的磁芯 \N 每个磁芯存 1 位数据

each one holding 1 bit of data.

每个黄色方格 有32行x32列的磁芯 \N 每个磁芯存 1 位数据

So, each of these yellow squares could hold 1024 bits.

所以能存 1024 位(bit) (32x32=1024)

In total, there are 9 of these,

总共 9 个黄色方格

so this memory board could hold a maximum of 9216 bits,

所以这块板子最多能存 9216 位(bit) (1024x9=9216)

which is around 9 kilobytes.

换算过来大约是 9 千字节 \N (9216 bit ~= 9 kb)

The first big use of core memory was MIT's Whirlwind 1 computer, in 1953,

磁芯内存的第一次大规模运用\N 是 1953 年麻省理工学院的 Whirlwind 1 计算机

which used a 32 by 32 core arrangement.

磁芯排列是 32×32

And, instead of just a single plane of cores, like this,

用了 16 块板子，能存储大约 16000 位(bit)

it was 16 boards deep, providing roughly 16 thousand bits of storage.

用了 16 块板子，能存储大约 16000 位(bit)

Importantly, unlike delay line memory,

更重要的是，不像"延迟线存储器" \N 磁芯存储器能随时访问任何一位(bit)

any bit could be accessed at any time.

更重要的是，不像"延迟线存储器" \N 磁芯存储器能随时访问任何一位(bit)

This was a killer feature,

这在当时非常了不起

and magnetic core memory became the predominant Random Access Memory technology

"磁芯存储器" 从 1950 年代中期开始成为主流 \N 流行了 20 多年

for two decades, beginning in the mid 1950

"磁芯存储器" 从 1950 年代中期开始成为主流 \N 流行了 20 多年

even though it was typically woven by hand!

而且一般还是手工编织的！

Although starting at roughly 1 dollar per bit,

刚开始时 存储成本大约 1 美元 1 位(bit) \N 到1970年代，下降到 1 美分左右

the cost fell to around 1 cent per bit by the 1970s.

刚开始时 存储成本大约 1 美元 1 位(bit) \N 到1970年代，下降到 1 美分左右

Unfortunately, even 1 cent per bit isn't cheap enough for storage.

不幸的是，即使每位 1 美分也不够便宜

As previously mentioned,

之前提过，现代手机随便拍张照片都有 5 MB

an average smartphone photo is around 5 megabytes in size,

之前提过，现代手机随便拍张照片都有 5 MB

that's roughly 40 million bits.

5MB 约等于 4000 万 bit

Would you pay 4 hundred thousand dollars to store a photo on core memory?

你愿意花 40 万美元在"磁芯存储器"上存照片吗？

If you have that kind of money to drop,

如果你有这么多钱

did you know that Crash Course is on Patreon?

你知道 Crash Course 在 Patreon 有赞助页吗？

Right? Wink wink.

对吧？你懂的

Anyway, there was tremendous research into storage technologies happening at this time.

总之，当时对存储技术进行了大量的研究

By 1951, Eckert and Mauchly had started their own company,

到 1951 年，Eckert 和 Mauchly 创立了自己的公司

and designed a new computer called UNIVAC,

设计了一台叫 UNIVAC 的新电脑

one of the earliest commercially sold computers.

最早进行商业销售的电脑之一

It debuted with a new form of computer storage:

它推出了一种新存储：磁带

magnetic tape.

它推出了一种新存储：磁带

This was a long, thin and flexible strip of magnetic material, stored in reels.

磁带是纤薄柔软的一长条磁性带子 卷在轴上

The tape could be moved forwards or backwards inside of a machine called a tape drive.

磁带可以在"磁带驱动器"内前后移动

Inside is a write head,

里面有一个"写头"绕了电线，电流通过产生磁场

which passes current through a wound wire to generate a magnetic field,

里面有一个"写头"绕了电线，电流通过产生磁场

causing a small section of the tape to become magnetized.

导致磁带的一小部分被磁化

The direction of the current sets the polarity, again, perfect for storing 1's and 0's.

电流方向决定了极性，代表 1 和 0

There was also a separate read head could detect the polarity non-destructively.

还有一个"读头"，可以非破坏性地检测极性

The UNIVAC used half-inch-wide tape with 8 parallel data tracks,

UNIVAC 用了半英寸宽，8条并行的磁带

each able to store 128 bits of data per inch.

磁带每英寸可存 128 位数据

With each reel containing 1200 feet of tape,

每卷有 1200 英尺长

it meant you could store roughly 15 million bits

意味着一共可以存 1500 万位左右

- that's almost 2 megabytes!

- 接近2兆字节！（2 MB）

Although tape drives were expensive,

虽然磁带驱动器很贵，但磁带又便宜又小

the magnetic tape itself was cheap and compact,

虽然磁带驱动器很贵，但磁带又便宜又小

and for this reason, they're still used today for archiving data.

因此磁带至今仍用于存档

The main drawback is access speed.

磁带的主要缺点是访问速度

Tape is inherently sequential,

磁带是连续的，必须倒带或快进到达特定位置

you have to rewind or fast-forward to get to data you want.

磁带是连续的，必须倒带或快进到达特定位置

This might mean traversing hundreds of feet of tape to retrieve a single byte,

可能要几百英尺才能得到某个字节(byte)，这很慢

which is slow.

可能要几百英尺才能得到某个字节(byte)，这很慢

A related popular technology in the 1950s and 60s was Magnetic Drum Memory.

1950,60年代，有个类似技术是 "磁鼓存储器"

This was a metal cylinder - called a drum - coated in a magnetic material for recording data

有金属圆筒，盖满了磁性材料以记录数据

The drum was rotated continuously,

滚筒会持续旋转，周围有数十个读写头

and positioned along its length were dozens of read and write heads.

滚筒会持续旋转，周围有数十个读写头

These would wait for the right spot to rotate underneath them to read or write a bit of data.

等滚筒转到正确的位置\N 读写头会读或写 1 位(bit) 数据

To keep this delay as short as possible,

为了尽可能缩短延迟, 鼓轮每分钟上千转！

drums were rotated thousand of revolutions per minute!

为了尽可能缩短延迟, 鼓轮每分钟上千转！

By 1953, when the technology started to take off,

到 1953 年，磁鼓技术飞速发展 \N 可以买到存 80,000 位的"磁鼓存储器"

you could buy units able to record 80,000 bits of data

到 1953 年，磁鼓技术飞速发展 \N 可以买到存 80,000 位的"磁鼓存储器"

- that's 10 kilobytes,

- 也就是 10 KB

but the manufacture of drums ceased in the 1970s.

但到 1970 年代 "磁鼓存储器" 不再生产

However, Magnetic Drums did directly lead to the development of Hard Disk Drives,

然而，磁鼓导致了硬盘的发展 \N 硬盘和磁鼓很相似

which are very similar, but use a different geometric configuration.

然而，磁鼓导致了硬盘的发展 \N 硬盘和磁鼓很相似

Instead of large cylinder, hard disks use,

不过硬盘用的是盘，不像磁鼓用圆柱体，因此得名

well disks that are hard.

不过硬盘用的是盘，不像磁鼓用圆柱体，因此得名

Hence the name!

不过硬盘用的是盘，不像磁鼓用圆柱体，因此得名

The storage principle is the same,

原理是一样的，磁盘表面有磁性

the surface of a disk is magnetic,

原理是一样的，磁盘表面有磁性

allowing write and read heads to store and retrieve 1's and 0's.

写入头和读取头 可以处理上面的 1 和 0

The great thing about disks is that they are thin,

硬盘的好处是薄，可以叠在一起

so you can stack many of them together,

硬盘的好处是薄，可以叠在一起

providing a lot of surface area for data storage.

提供更多表面积来存数据

That's exactly what IBM did for the world's first computer with a disk drive:

IBM 对世上第一台磁盘计算机就是这样做的

the RAMAC 305.

RAMAC 305

Sweet name BTW.

顺便一说名字不错

It contained fifty, 24-inch diameter disks,

它有 50 张 24 英寸直径的磁盘，总共能存 5 MB 左右

offering a total storage capacity of roughly 5 megabytes.

它有 50 张 24 英寸直径的磁盘，总共能存 5 MB 左右

Yess!! We've finally gotten to a technology that can store a single smartphone photo!

太棒啦! 终于能存一张现代手机的照片了！这年是 1956 年

The year was 1956.

太棒啦! 终于能存一张现代手机的照片了！这年是 1956 年

To access any bit of data,

要访问某个特定 bit

a read/write head would travel up or down the stack to the right disk,

一个读/写磁头会向上或向下移动，找到正确的磁盘

and then slide in between them.

然后磁头会滑进去

Like drum memory, the disks are spinning,

就像磁鼓存储器一样，磁盘也会高速旋转

so the head has to wait for the right section to come around.

所以读写头要等到正确的部分转过来

The RAMAC 305 could access any block of data, on average, in around 6/10ths of a second,

RAMAC 305 访问任意数据，平均只要六分之一秒左右

what's called the seek time.

也叫寻道时间

While great for storage, this was not nearly fast enough for memory,

虽然六分之一秒对存储器来说算不错 \N 但对内存来说还不够快

so the RAMAC 305 also had drum memory and magnetic core memory.

所以 RAMAC 305 还有"磁鼓存储器"和"磁芯存储器"

This is an example of a memory hierarchy,

这是"内存层次结构"的一个例子

where you have a little bit of fast memory, which is expensive,

一小部分高速+昂贵的内存

slightly more medium-speed memory, which is less expensive,

一部分稍慢+相对便宜些的内存

and then a lot of slowish memory, which is cheap.

还有更慢+更便宜的内存

This mixed approach strikes a balance between cost and speed.

这种混合 在成本和速度间取得平衡

Hard disk drives rapidly improved and became commonplace by the 1970s.

1970 年代，硬盘大幅度改进并变得普遍

A hard disk like this can easily hold 1 terabyte of data today

如今的硬盘可以轻易容纳 1TB 的数据

- that's a trillion bytes - or roughly 200,000 five megabyte photos!

能存 20 万张 5MB 的照片！

And these types of drives can be bought online for as little as 40 US dollars.

网上最低 40 美元就可以买到

That's 0.0000000005 cents per bit.

每 bit 成本 0.0000000005 美分

A huge improvement over core memory's 1 cent per bit!

比磁芯内存 1 美分 1 bit 好多了！

Also, modern drives have an average seek time of under 1/100th of a second.

另外，现代硬盘的平均寻道时间低于 1/100 秒

I should also briefly mention a close cousin of hard disks, the floppy disk,

我简单地提一下硬盘的亲戚，软盘

which is basically the same thing, but uses a magnetic medium that's, floppy.

除了磁盘是软的，其他基本一样

You might recognise it as the save icon on some of your applications,

你可能见过某些程序的保存图标是一个软盘

but it was once a real physical object!

软盘曾经是真实存在的东西！

It was most commonly used for portable storage,

软盘是为了便携，在 1970~1990 非常流行

and became near ubiquitous from the mid 1970s up to the mid 90s.

软盘是为了便携，在 1970~1990 非常流行

And today it makes a pretty good coaster.

如今当杯垫挺不错的

Higher density floppy disks, like Zip Disks,

密度更高的软盘，如 Zip Disks，在90年代中期流行起来

became popular in the mid 1990s,

密度更高的软盘，如 Zip Disks，在90年代中期流行起来

but fell out of favor within a decade.

但十年内就消失了

Optical storage came onto the scene in 1972, in the form of a 12-inch "laser disc."

光学存储器于 1972 年出现，12 英寸的"激光盘"

However, you are probably more familiar with its later, smaller, are more popular cousin,

你可能对后来的产品更熟：光盘（简称 CD）

the Compact Disk, or CD,

你可能对后来的产品更熟：光盘（简称 CD）

as well as the DVD which took off in the 90s.

以及 90 年代流行的 DVD

Functionally, these technologies are pretty similar to hard disks and floppy disks,

功能和硬盘软盘一样，都是存数据.

but instead of storing data magnetically,

但用的不是磁性

optical disks have little physical divots in their surface that cause light to be reflected differently,

光盘表面有很多小坑，造成光的不同反射

which is captured by an optical sensor, and decoded into 1's and 0's.

光学传感器会捕获到，并解码为 1 和 0

However, today, things are moving to solid state technologies, with no moving parts,

如今，存储技术在朝固态前进，没有机械活动部件

like this hard drive and also this USB stick.

比如这个硬盘，以及 U 盘

Inside are Integrated Circuits,

里面是集成电路，我们在第 15 集讨论过

which we talked about in Episode 15.

里面是集成电路，我们在第 15 集讨论过

The first RAM integrated circuits became available in 1972 at 1 cent per bit,

第一个 RAM 集成电路出现于 1972 年 \N 成本每比特 1 美分

quickly making magnetic core memory obsolete.

使"磁芯存储器"迅速过时

Today, costs have fallen so far,

如今成本下降了更多 \N 机械硬盘 被 固态硬盘 逐渐替代，简称 SSD

that hard disk drives are being replaced with non-volatile,

如今成本下降了更多 \N 机械硬盘 被 固态硬盘 逐渐替代，简称 SSD

Solid State Drives, or SSDs, as the cool kids say.

如今成本下降了更多 \N 机械硬盘 被 固态硬盘 逐渐替代，简称 SSD

Because they contain no moving parts,

由于 SSD 没有移动部件

they don't really have to seek anywhere,

磁头不用等磁盘转

so SSD access times are typically under 1/1000th of a second.

所以 SSD 访问时间低于 1/1000 秒

That's fast!

这很快！

But it's still many times slower than your computer's RAM.

但还是比 RAM 慢很多倍

For this reason, computers today still use memory hierarchies.

所以现代计算机 仍然用存储层次结构

So, we've come along way since the 1940s.

我们从 1940 年代到现在进步巨大

Much like transistor count and Moore's law,

就像在第 14 集讨论过的 晶体管数量和摩尔定律

which we talked about in Episode 14,

就像在第 14 集讨论过的 晶体管数量和摩尔定律

memory and storage technologies have followed a similar exponential trend.

内存和存储技术也有类似的趋势

From early core memory costing millions of dollars per megabyte, we're steadily fallen,

从早期每 MB 成本上百万美元，下滑到

to mere cents by 2000, and only fractions of a cent today.

2000 年只要几分钱，如今远远低于 1 分钱

Plus, there's WAY less punch cards to keep track of.

完全没有打孔纸卡

Seriously, can you imagine if there was a slight breeze in that room containing the SAGE program?

你能想象 SEGA 的纸卡房间风一吹会怎样吗？

62,500 punch cards.

62,500 张卡

I don't even want to think about it.

我想都不敢想

I'll see you next week.

我们下周见

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Last episode we talked about data storage, how technologies like magnetic tape and hard

上集我们讲了数据存储，磁带和硬盘这样的技术

disks can store trillions of bits of data,

可以在断电状态长时间存上万亿个位

for long durations, even without power.

可以在断电状态长时间存上万亿个位

Which is perfect for recording "big blobs" of related data,

非常合适存一整块有关系的数据，或者说"文件"

what are more commonly called computer files.

非常合适存一整块有关系的数据，或者说"文件"

You've no doubt encountered many types,

你肯定见过很多种文件 \N 比如文本文件，音乐文件，照片和视频

like text files, music files, photos and videos.

你肯定见过很多种文件 \N 比如文本文件，音乐文件，照片和视频

Today, we're going to talk about how files work,

今天，我们要讨论文件到底是什么 \N 以及计算机怎么管理文件

and how computers keep them all organized with File Systems.

今天，我们要讨论文件到底是什么 \N 以及计算机怎么管理文件

It's perfectly legal for a file to contain arbitrary, unformatted data,

随意排列文件数据完全没问题，但按格式排会更好

but it's most useful and practical if the data inside the file is organized somehow.

随意排列文件数据完全没问题，但按格式排会更好

This is called a file format.

这叫 "文件格式"

You can invent your own, and programmers do that from time to time,

你可以发明自己的文件格式，程序员偶尔会这样做

but it's usually best and easiest to use an existing standard, like JPEG and MP3.

但最好用现成标准，比如 JPEG 和 MP3

Let's look at some simple file formats.

来看一些简单文件格式，最简单的是文本文件

The most straightforward are text files,

来看一些简单文件格式，最简单的是文本文件

also know as TXT file, which contain...surprise! text.

也叫 TXT 文件, 里面包含的是... 文字 （惊喜吧）

Like all computer files, this is just a huge list of numbers, stored as binary.

就像所有其它文件，文本文件只是一长串二进制数

If we look at the raw values of a text file in storage, it would look something like this:

原始值看起来会像这样：

We can view this as decimal numbers instead of binary,

可以转成十进制看，但帮助不大

but that still doesn't help us read the text.

可以转成十进制看，但帮助不大

The key to interpreting this data is knowing that TXT files use ASCII,

解码数据的关键是 ASCII 编码

a character encoding standard we discussed way back in Episode 4.

一种字符编码标准，第 4 集讨论过.

So, in ASCII, our first value, 72, maps to the capital letter H.

第一个值 72 \N 在 ASCII 中是大写字母 H

And in this way, we decode the whole file.

以此类推 解码其他数字

Let's look at a more complicated example: a WAVE File, also called a WAV,

来看一个更复杂的例子：波形(Wave)文件，也叫 WAV \N 它存音频数据

which stores audio.

来看一个更复杂的例子：波形(Wave)文件，也叫 WAV \N 它存音频数据

Before we can correctly read the data, we need to know some information,

在正确读取数据前，需要知道一些信息

like the bit rate and whether it's a single track or stereo.

比如码率(bit rate)，以及是单声道还是立体声

Data about data, is called meta data.

关于数据的数据，叫"元数据"(meta data)

This metadata is stored at the front of the file, ahead of any actual data,

元数据存在文件开头，在实际数据前面 \N 因此也叫 文件头(Header)

in what's known as a Header.

元数据存在文件开头，在实际数据前面 \N 因此也叫 文件头(Header)

Here's what the first 44 bytes of a WAV file looks like.

WAV 文件的前 44 个字节长这样

Some parts are always the same, like where it spells out W-A-V-E.

有的部分总是一样的，比如写着 WAVE 的部分

Other parts contain numbers that change depending on the data contained within.

其他部分的内容，会根据数据变化

The audio data comes right behind the metadata, and it's stored as a long list of numbers.

音频数据紧跟在元数据后面，是一长串数字

These values represent the amplitude of sound captured many times per second, and if you

数字代表每秒捕获多次的声音幅度

want a primer on sound, check out our video all about it in Crash Course Physics.

如果想学声音的基础知识 \N 可以看物理速成课第18集

Link in the dobblydoo.

如果想学声音的基础知识 \N 可以看物理速成课第18集

As an example, let's look at a waveform of me saying: "hello!" Hello!

举个例子，看一下"你好"的波形

Now that we've captured some sound, let's zoom into a little snippet.

现在捕获到了一些声音，我们放大看一下

A digital microphone, like the one in your computer or smartphone,

电脑和手机麦克风，每秒可以对声音进行上千次采样

samples the sound pressure thousands of times.

电脑和手机麦克风，每秒可以对声音进行上千次采样

Each sample can be represented as a number.

每次采样可以用一个数字表示

Larger numbers mean higher sound pressure, what's called amplitude.

声压越高数字越大，也叫"振幅"

And these numbers are exactly what gets stored in a WAVE file!

WAVE 文件里存的就是这些数据！

Thousands of amplitudes for every single second of audio!

每秒上千次的振幅！

When it's time to play this file, an audio program needs to actuate the computer's speakers

播放声音文件时，扬声器会产生相同的波形

such that the original waveform is emitted.

播放声音文件时，扬声器会产生相同的波形

"Hello!"

"你好！"

So, now that you're getting the hang of file formats, let's talk about bitmaps or

现在来谈谈 位图(Bitmap)，后缀 .bmp, 它存图片

BMP, which store pictures.

现在来谈谈 位图(Bitmap)，后缀 .bmp, 它存图片

On a computer, Pictures are made up of little tiny square elements called pixels.

计算机上，图片由很多个叫"像素"的方块组成

Each pixel is a combination of three colors: red, green and blue.

每个像素由三种颜色组成：红，绿，蓝

These are called additive primary colors, and they can be mixed together to create any

叫"加色三原色"，混在一起可以创造其它颜色

other color on our electronic displays.

叫"加色三原色"，混在一起可以创造其它颜色

Now, just like WAV files, BMPs start with metadata,

就像 WAV 文件一样，BMP 文件开头也是元数据 \N 有图片宽度，图片高度，颜色深度

including key values like image width, image height, and color depth.

就像 WAV 文件一样，BMP 文件开头也是元数据 \N 有图片宽度，图片高度，颜色深度

As an example, let's say the metadata specified an image 4 pixels wide, by 4 pixels tall,

举例，假设元数据说图是 4像素宽 x 4像素高

with a 24-bit color depth - that's 8-bits for red, 8-bits for green, and 8-bits for blue.

颜色深度 24 位\N 8 位红色，8 位绿色，8 位蓝色

As a reminder, 8 bits is the same as one byte.

提醒一下，8位 (bit) 和 1字节(byte)是一回事

The smallest number a byte can store is 0, and the largest is 255.

一个字节能表示的最小数是 0，最大 255

Our image data is going to look something like this:

图像数据看起来会类似这样：\N 来看看第一个像素的颜色

Let's look at the color of our first pixel.

图像数据看起来会类似这样：\N 来看看第一个像素的颜色

It has 255 for its red value, 255 for green and 255 for blue.

红色是255，绿色是255，蓝色也是255

This equates to full intensity red, full intensity green and full intensity blue.

这等同于全强度红色，全强度绿色和全强度蓝色

These colors blend together on your computer monitor to become white.

混合在一起变成白色

So our first pixel is white!

所以第一个像素是白色！

The next pixel has a Red-Green-Blue, or RGB value of 255, 255, 0.

下一个像素的红绿蓝值，或 RGB 值 \N 255,255,0 是黄色！

That's the color yellow!

下一个像素的红绿蓝值，或 RGB 值 \N 255,255,0 是黄色！

The pixel after that has a RGB value of 0,0,0 - that's zero intensity everything, which is black.

下一个像素是 0,0,0 ，黑色

And the next one is yellow.

下一个是黄色

Because the metadata specified this was a 4 by 4 image, we know that we've reached

因为元数据说图片是 4x4 \N 我们知道现在到了第一行结尾

the end of our first row of pixels.

因为元数据说图片是 4x4 \N 我们知道现在到了第一行结尾

So, we need to drop down a row.

所以换一行

The next RGB value is 255,255,0 yellow again.

下一个 RGB 值是 255,255,0，又是黄色

Okay, let's go ahead and read all the pixels in our 4x4 image tada!

好，我们读完剩下的像素

A very low resolution pac-man!

一个低分辨率的吃豆人

Obviously this is a simple example of a small image,

刚才显然只是一个简单例子，但这张图片也可以用 BMP 存

but we could just as easily store this image in a BMP.

刚才显然只是一个简单例子，但这张图片也可以用 BMP 存

I want to emphasize again that it doesn't matter if it's a text file, WAV,

我想再次强调，不管是文本文件，WAV，BMP

BMP, or fancier formats we don't have time to discuss,

或是我们没时间讨论的其他格式

Under the hood, they're all the same: long lists of numbers, stored as binary, on a storage device.

文件在底层全是一样的： 一长串二进制

File formats are the key to reading and understanding the data inside.

为了知道文件是什么，文件格式至关重要

Now that you understand files a little better, let's move on to

现在你对文件更了解了 \N 我们接下来讨论计算机怎么存文件

how computers go about storing them.

现在你对文件更了解了 \N 我们接下来讨论计算机怎么存文件

Even though the underlying storage medium might be

虽然硬件可能是磁带，磁鼓，磁盘或集成电路

a strip of tape, a drum, a disk, or integrated circuits...

虽然硬件可能是磁带，磁鼓，磁盘或集成电路

hardware and software abstractions let us think of storage as a

通过软硬件抽象后，可以看成一排能存数据的桶

long line of little buckets that store values.

通过软硬件抽象后，可以看成一排能存数据的桶

In the early days, when computers only performed one computation

在很早期时，计算机只做一件事，比如算火炮射程表 \N 整个储存器就像一整个文件

like calculating artillery range tables. the entire storage operated like one big file.

在很早期时，计算机只做一件事，比如算火炮射程表 \N 整个储存器就像一整个文件

Data started at the beginning of storage, and then filled it up in order as output was

数据从头存到尾，直到占满

produced, up to the storage capacity.

数据从头存到尾，直到占满

However, as computational power and storage capacity improved, it became possible, and

但随着计算能力和存储容量的提高 \N 存多个文件变得非常有用

useful, to store more than one file at a time.

但随着计算能力和存储容量的提高 \N 存多个文件变得非常有用

The simplest option is to store files back-to-back.

最简单的方法 是把文件连续存储

This can work... but how does the computer know where files begin and end?

这样能用, \N 但怎么知道文件开头和结尾在哪里？

Storage devices have no notion of files C they're just a mechanism for storing lots of bits.

储存器没有文件的概念，只是存储大量位

So, for this to work, we need to have a special file that records where other ones are located.

所以为了存多个文件 \N 需要一个特殊文件，记录其他文件的位置

This goes by many names, but a good general term is Directory File.

这个特殊文件有很多名字，这里泛称 "目录文件"

Most often, it's kept right at the front of storage, so we always know where to access it.

这个文件经常存在最开头，方便找

Location zero!

位置 0！

Inside the Directory File are the names of all the other files in storage.

目录文件里，存所有其他文件的名字

In our example, they each have a name, followed by a period

格式是文件名 + 一个句号 + 扩展名，比如 BMP 或 WAV

and end with what's called a File Extension, like "BMP" or "WAV".

格式是文件名 + 一个句号 + 扩展名，比如 BMP 或 WAV

Those further assist programs in identifying file types.

扩展名帮助得知文件类型

The Directory File also stores metadata about these files, like when they were created and

目录文件还存文件的元数据，比如创建时间

last modified, who the owner is, and if it can be read, written or both.

最后修改时间，文件所有者是谁\N是否能读/写 或读写都行

But most importantly, the directory file contains where these files

最重要的是，目录文件有文件起始位置和长度

begin in storage, and how long they are.

最重要的是，目录文件有文件起始位置和长度

If we want to add a file, remove a file, change a filename, or similar,

如果要添加文件，删除文件，更改文件名等

we have to update the information in the Directory File.

必须更新目录文件

It's like the Table of Contents in a book, if you make a chapter shorter, or move it

就像书的目录，如果缩短或移动了一个章节 \N 要更新目录，不然页码对不上

somewhere else, you have to update the table of contents, otherwise the page numbers won't match!

就像书的目录，如果缩短或移动了一个章节 \N 要更新目录，不然页码对不上

The Directory File, and the maintenance of it, is an example of a very basic File System,

目录文件，以及对目录文件的管理 \N 是一个非常简单的文件系统例子

the part of an Operating System that manages and keep track of stored files.

文件系统专门负责管理文件

This particular example is a called a Flat File System, because they're all stored at one level.

刚刚的例子叫"平面文件系统" \N因为文件都在同一个层次

It's flat!

平的！

Of course, packing files together, back-to-back, is a bit of a problem,

当然，把文件前后排在一起 有个问题

because if we want to add some data to let's say "todo.txt",

如果给 todo.txt 加一点数据 \N 会覆盖掉后面 carrie.bmp 的一部分

there's no room to do it without overwriting part of "carrie.bmp".

如果给 todo.txt 加一点数据 \N 会覆盖掉后面 carrie.bmp 的一部分

So modern File Systems do two things.

所以现代文件系统会做两件事

First, they store files in blocks.

1. 把空间划分成一块块 \N 导致有一些 "预留空间" 可以方便改动

This leaves a little extra space for changes, called slack space.

1. 把空间划分成一块块 \N 导致有一些 "预留空间" 可以方便改动

It also means that all file data is aligned to a common size, which simplifies management.

同时也方便管理

In a scheme like this, our Directory File needs to keep track of

用这样的方案，目录文件要记录文件在哪些块里

what block each one is stored in.

用这样的方案，目录文件要记录文件在哪些块里

The second thing File Systems do, is allow files to be broken up into chunks

2. 拆分文件，存在多个块里

and stored across many blocks.

2. 拆分文件，存在多个块里

So let's say we open "todo.txt", and we add a few more items then the file becomes

假设打开 todo.txt 加了些内容\N 文件太大存不进一块里

too big to be saved in its one block.

假设打开 todo.txt 加了些内容\N 文件太大存不进一块里

We don't want to overwrite the neighboring one, so instead, the File System allocates

我们不想覆盖掉隔壁的块，所以文件系统会分配 \N 一个没使用的块，容纳额外的数据

an unused block, which can accommodate extra data.

我们不想覆盖掉隔壁的块，所以文件系统会分配 \N 一个没使用的块，容纳额外的数据

With a File System scheme like this, the Directory File needs to store

目录文件会记录不止一个块，而是多个块

not just one block per file, but rather a list of blocks per file.

目录文件会记录不止一个块，而是多个块

In this way, we can have files of variable sizes that can be easily

只要分配块，文件可以轻松增大缩小

expanded and shrunk, simply by allocating and deallocating blocks.

只要分配块，文件可以轻松增大缩小

If you watched our episode on Operating Systems, this should sound a lot like Virtual Memory.

如果你看了第18集 操作系统 \N 这听起来很像"虚拟内存"

Conceptually it's very similar!

概念上讲的确很像！

Now let's say we want to delete "carrie.bmp".

假设想删掉 carrie.bmp \N 只需要在目录文件删掉那条记录

To do that, we can simply remove the entry from the Directory File.

假设想删掉 carrie.bmp \N 只需要在目录文件删掉那条记录

This, in turn, causes one block to become free.

让一块空间变成了可用

Note that we didn't actually erase the file's data in storage, we just deleted the record of it.

注意这里没有擦除数据，只是把记录删了

At some point, that block will be overwritten with new data, but until then, it just sits there.

之后某个时候，那些块会被新数据覆盖 \N 但在此之前，数据还在原处

This is one way that computer forensic teams can "recover" data from computers even

所以计算机取证团队可以"恢复"数据

though people think it has been deleted. Crafty!

虽然别人以为数据已经"删了", 狡猾！

Ok, let's say we add even more items to our todo list, which causes the File System

假设往 todo.txt 加了更多数据 \N 所以操作系统分配了一个新块，用了刚刚 carrie.bmp 的块

to allocate yet another block to the file, in this case,

假设往 todo.txt 加了更多数据 \N 所以操作系统分配了一个新块，用了刚刚 carrie.bmp 的块

recycling the block freed from carrie.bmp.

假设往 todo.txt 加了更多数据 \N 所以操作系统分配了一个新块，用了刚刚 carrie.bmp 的块

Now our "todo.txt" is stored across 3 blocks, spaced apart, and also out of order.

现在 todo.txt 在 3 个块里，隔开了，顺序也是乱的

Files getting broken up across storage like this is called fragmentation.

这叫碎片

It's the inevitable byproduct of files being created, deleted and modified.

碎片是增/删/改文件导致的，不可避免

For many storage technologies, this is bad news.

对很多存储技术来说，碎片是坏事

On magnetic tape, reading todo.txt into memory would require

如果 todo.txt 存在磁带上，读取文件要

seeking to block 1, then fast forwarding to block 5, and then rewinding to block 3

先读块1, 然后快进到块5，然后往回转到块2

that's a lot of back and forth!

来回转个半天

In real world File Systems, large files might be stored across hundreds of blocks,

现实世界中，大文件可能存在数百个块里

and you don't want to have to wait five minutes for your files to open.

你可不想等五分钟才打开文件

The answer is defragmentation!

答案是碎片整理！

That might sound like technobabble, but the process is really simple,

这个词听起来好像很复杂，但实际过程很简单

and once upon a time it was really fun to watch!

以前看计算机做碎片整理 真的很有趣！

The computer copies around data so that files have blocks located together

计算机会把数据来回移动，排列成正确的顺序

in storage and in the right order.

计算机会把数据来回移动，排列成正确的顺序

After we've defragged, we can read our todo file,

整理后 todo.txt 在 1 2 3，方便读取.

now located in blocks 1 through 3, in a single, quick read pass.

整理后 todo.txt 在 1 2 3，方便读取.

So far, we've only been talking about Flat File Systems,

目前只说了平面文件系统，文件都在同一个目录里.

where they're all stored in one directory.

目前只说了平面文件系统，文件都在同一个目录里.

This worked ok when computers only had a little bit of storage,

如果存储空间不多，这可能就够用了 \N 因为只有十几个文件

and you might only have a dozen or so files.

如果存储空间不多，这可能就够用了 \N 因为只有十几个文件

But as storage capacity exploded, like we discussed last episode,

但上集说过，容量爆炸式增长，文件数量也飞速增长

so did the number of files on computers.

但上集说过，容量爆炸式增长，文件数量也飞速增长

Very quickly, it became impractical to store all files together at one level.

很快，所有文件都存在同一层变得不切实际

Just like documents in the real world, it's handy to store related files together in folders.

就像现实世界\N 相关文件放在同一个文件夹会方便很多

Then we can put connected folders into folders, and so on.

然后文件夹套文件夹.

This is a Hierarchical File System, and its what your computer uses.

这叫"分层文件系统"，你的计算机现在就在用这个.

There are a variety of ways to implement this, but let's stick with the File System example

实现方法有很多种，我们用之前的例子来讲重点好了

we've been using to convey the main idea.

实现方法有很多种，我们用之前的例子来讲重点好了

The biggest change is that our Directory File needs to be able to point not just to files,

最大的变化是 目录文件不仅要指向文件, 还要指向目录

but also other directories.

最大的变化是 目录文件不仅要指向文件, 还要指向目录

To keep track of what's a file and what's a directory, we need some extra metadata.

我们需要额外元数据 来区分开文件和目录，

This Directory File is the top-most one, known as the Root Directory.

这个目录文件在最顶层，因此叫根目录

All other files and folders lie beneath this directory along various file paths.

所有其他文件和文件夹，都在根目录下

We can see inside of our "Root" Directory File that we have 3 files

图中可以看到根目录文件有3个文件 \N 2个子文件夹："音乐"和"照片"

and 2 subdirectories: music and photos.

图中可以看到根目录文件有3个文件 \N 2个子文件夹："音乐"和"照片"

If we want to see what's stored in our music directory, we have to go to that block and

如果想知道"音乐"文件夹里有什么 \N 必须去那边读取目录文件（格式和根目录文件一样）

read the Directory File located there; the format is the same as our root directory.

如果想知道"音乐"文件夹里有什么 \N 必须去那边读取目录文件（格式和根目录文件一样）

There's a lot of great songs in there!

有很多好歌啊！

In addition to being able to create hierarchies of unlimited depth,

除了能做无限深度的文件夹 \N 这个方法也让我们可以轻松移动文件

this method also allows us to easily move around files.

除了能做无限深度的文件夹 \N 这个方法也让我们可以轻松移动文件

So, if we wanted to move "theme.wav" from our root directory to the music directory,

如果想把 theme.wav 从根目录移到音乐目录

we don't have to re-arrange any blocks of data.

不用移动任何数据块

We can simply modify the two Directory Files, removing an entry from one and adding it to another.

只需要改两个目录文件 \N 一个文件里删一条记录，另一个文件里加一条记录

Importantly, the theme.wav file stays in block 5.

theme.wav 依然在块5

So that's a quick overview of the key principles of File Systems.

文件系统的几个重要概念 现在介绍完了.

They provide yet another way to move up a new level of abstraction.

它提供了一层新抽象！

File systems allow us to hide the raw bits stored on magnetic tape, spinning disks and

文件系统使我们不必关心 \N 文件在磁带或磁盘的具体位置

the like, and they let us think of data as neatly organized and easily accessible files.

整理和访问文件更加方便

We even started talking about users, not programmers, manipulating data,

我们像普通用户一样直观操纵数据，比如打开和整理文件

like opening files and organizing them,

我们像普通用户一样直观操纵数据，比如打开和整理文件

foreshadowing where the series will be going in a few episodes.

接下来几集也会从用户角度看问题

I'll see you next week.

下周见

This episode is brought to you by Curiosity Stream.

本集由 Curiosity Stream 赞助播出

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Last episode we talked about Files, bundles of data, stored on a computer, that

上集我们讨论了文件格式，如何编码文字，声音，图片

are formatted and arranged to encode information, like text, sound or images.

上集我们讨论了文件格式，如何编码文字，声音，图片

We even discussed some basic file formats, like text, wave, and bitmap.

还举了具体例子 .txt .wav .bmp

While these formats are perfectly fine and still used today,

这些格式虽然管用，而且现在还在用 \N 但它们的简单性意味着效率不高

their simplicity also means they're not very efficient.

这些格式虽然管用，而且现在还在用 \N 但它们的简单性意味着效率不高

Ideally, we want files to be as small as possible, so we can store lots of them without filling

我们希望文件能小一点，这样能存大量文件，传输也会快一些

up our hard drives, and also transmit them more quickly.

我们希望文件能小一点，这样能存大量文件，传输也会快一些

Nothing is more frustrating than waiting for an email attachment to download. Ugh!

等邮件附件下载烦死人了

The answer is compression, which literally squeezes data into a smaller size.

解决方法是 压缩，把数据占用的空间压得更小

To do this, we have to encode data using fewer bits than the original representation.

用更少的位(bit)来表示数据

That might sound like magic, but it's actually computer science!

听起来像魔法，但其实是计算机科学！

Lets return to our old friend from last episode, Mr. Pac-man!

我们继续用上集的 吃豆人例子，图像是 4像素x4像素

This image is 4 pixels by 4 pixels.

我们继续用上集的 吃豆人例子，图像是 4像素x4像素

As we discussed, image data is typically stored as a list of pixel values.

之前说过，图像一般存成一长串像素值

To know where rows end, image files have metadata, which defines properties like dimensions.

为了知道一行在哪里结束 \N 图像要有元数据，写明尺寸等属性

But, to keep it simple today, we're not going to worry about it.

但为了简单起见，今天忽略这些细节

If you mix full intensity red, green and blue - that's 255 for all

如果红绿蓝都是 255 会得到白色

three values - you get the color white.

如果红绿蓝都是 255 会得到白色

If you mix full intensity red and green, but no blue (it's 0), you get yellow.

如果混合 255红色和255绿色，会得到黄色

We have 16 pixels in our image, and each of those needs 3 bytes of color data.

这个图像有16个像素(4x4), 每个像素3个字节

That means this image's data will consume 48 bytes of storage.

总共占48个字节（16x3=48）

But, we can compress the data and pack it into a smaller number of bytes than 48!

但我们可以压缩到少于 48 个字节

One way to compress data is to reduce repeated or redundant information.

一种方法是 减少重复信息

The most straightforward way to do this is called Run-Length Encoding.

最简单的方法叫 游程编码(Run-Length Encoding)

This takes advantage of the fact that there are often runs of identical values in files.

适合经常出现相同值的文件

For example, in our pac-man image, there are 7 yellow pixels in a row.

比如吃豆人 有7个连续黄色像素

Instead of encoding redundant data: yellow pixel, yellow pixel, yellow pixel, and so

与其全存下来：黄色，黄色，黄色...

on, we can just say "there's 7 yellow pixels in a row" by inserting

可以插入一个额外字节，代表有7个连续黄色像素

an extra byte that specifies the length of the run, like so:

可以插入一个额外字节，代表有7个连续黄色像素

And then we can eliminate the redundant data behind it.

然后删掉后面的重复数据.

To ensure that computers don't get confused with which bytes are run lengths and which

为了让计算机能分辨哪些字节是"长度" 哪些字节是"颜色" \N 格式要一致

bytes represent color, we have to be consistent in how we apply this scheme.

为了让计算机能分辨哪些字节是"长度" 哪些字节是"颜色" \N 格式要一致

So, we need to preface all pixels with their run-length.

所以我们要给所有像素前面标上长度

In some cases, this actually adds data, but on the whole, we've dramatically reduced

有时候数据反而会变多，但就这个例子而言

the number of bytes we need to encode this image.

我们大大减少了字节数，之前是48 现在是24

We're now at 24 bytes, down from 48.

我们大大减少了字节数，之前是48 现在是24

That's 50% smaller!

小了50％！省了很多空间！

A huge saving!

小了50％！省了很多空间！

Also note that we haven't lost any data.

还有，我们没有损失任何数据 \N 我们可以轻易恢复到原来的数据

We can easily expand this back to the original form without any degradation.

还有，我们没有损失任何数据 \N 我们可以轻易恢复到原来的数据

A compression technique that has this characteristic is called

这叫"无损压缩"，没有丢失任何数据

lossless compression, because we don't lose anything.

这叫"无损压缩"，没有丢失任何数据

The decompressed data is identical to the original before compression, bit for bit.

解压缩后，数据和压缩前完全一样

Let's take a look at another type of lossless compression, where

我们来看另一种无损压缩，它用更紧凑的方式表示数据块

blocks of data are replaced by more compact representations.

我们来看另一种无损压缩，它用更紧凑的方式表示数据块

This is sort of like " don't forget to be awesome " being replaced by DFTBA.

有点像 "别忘了变厉害" 简写成 DFTBA

To do this, we need a dictionary that stores the mapping from codes to data.

为此，我们需要一个字典，存储"代码"和"数据"间的对应关系

Lets see how this works for our example.

我们看个例子

We can view our image as not just a string of individual pixels,

我们可以把图像看成一块块，而不是一个个像素

but as little blocks of data.

我们可以把图像看成一块块，而不是一个个像素

For simplicity, we're going to use pixel pairs, which are 6 bytes long,

为了简单，我们把2个像素当成1块（占6个字节）

but blocks can be any size.

但你也可以定成其他大小

In our example, there are only four pairings: White-yellow, black-yellow,

我们只有四对： 白黄 黑黄 黄黄 白白

yellow-yellow and white-white.

我们只有四对： 白黄 黑黄 黄黄 白白

Those are the data blocks in our dictionary we want to generate compact codes for.

我们会为这四对 生成紧凑代码(compact codes)

What's interesting, is that these blocks occur at different frequencies.

有趣的是，这些块的出现频率不同

One method for generating efficient codes is building a Huffman Tree, invented by David

1950年代 大卫·霍夫曼 发明了一种高效编码方式叫 \N "霍夫曼树"（Huffman Tree） 当时他是麻省理工学院的学生

Huffman while he was a student at MIT in the 1950s.

1950年代 大卫·霍夫曼 发明了一种高效编码方式叫 \N "霍夫曼树"（Huffman Tree） 当时他是麻省理工学院的学生

His algorithm goes like this.

算法是这样的

First, you layout all the possible blocks and their frequencies.

首先，列出所有块和出现频率，每轮选两个最低的频率

At every round, you select the two with the lowest frequencies.

首先，列出所有块和出现频率，每轮选两个最低的频率

Here, that's Black-Yellow and White-White, each with a frequency of 1.

这里 黑黄 和 白白 的频率最低，它们都是 1

You combine these into a little tree. which have a combined frequency of 2,

可以把它们组成一个树，总频率 2

so we record that.

可以把它们组成一个树，总频率 2

And now one step of the algorithm done.

现在完成了一轮算法

Now we repeat the process.

现在我们重复这样做

This time we have three things to choose from.

这次有3个可选

Just like before, we select the two with the lowest frequency, put them into a little tree,

就像上次一样，选频率最低的两个，放在一起，并记录总频率

and record the new total frequency of all the sub items.

就像上次一样，选频率最低的两个，放在一起，并记录总频率

Ok, we're almost done.

好，我们快完成了

This time it's easy to select the two items with the lowest frequency

这次很简单，因为只有2个选择

because there are only two things left to pick.

这次很简单，因为只有2个选择

We combine these into a tree, and now we're done!

把它们组合成一棵树就完成了！

Our tree looks like this, and it has a very cool property: it's arranged by frequency,

现在看起来像这样，它有一个很酷的属性：按频率排列

with less common items lower down.

频率低的在下面

So, now we have a tree, but you may be wondering how this gets us to a dictionary.

现在有了一棵树，你可能在想 "怎么把树变成字典？"

Well, we use our frequency-sorted tree to generate the codes we need

我们可以把每个分支用 0 和 1 标注，就像这样

by labeling each branch with a 0 or a 1, like so.

我们可以把每个分支用 0 和 1 标注，就像这样

With this, we can write out our code dictionary.

现在可以生成字典

Yellow-yellow is encoded as just a single 0. White-yellow is encoded as 10

黄黄 编码成 0 \N 白黄 编码成 10 \N 黑黄 编码成 110 \N 白白 编码成 111

Black-Yellow is 1 1 0. and finally white-white is 1 1 1.

黄黄 编码成 0 \N 白黄 编码成 10 \N 黑黄 编码成 110 \N 白白 编码成 111

The really cool thing about these codewords is that there's no way to

酷的地方是 它们绝对不会冲突

have conflicting codes, because each path down the tree is unique.

因为树的每条路径是唯一的

This means our codes are prefix-free, that is no code starts with another complete code.

意味着代码是"无前缀"的，没有代码是以另一个代码开头的

Now, let's return to our image data and compress it!

现在我们来压缩！

-- NOT BYTES -- BITS!! That's less than 2 bytes of data!

注意是位(bit)！ 不是字节(byte)！ \N 14位(bit) 还不到2个字节(byte)！

But, don't break out the champagne quite yet!

但，先别急着开香槟！

This data is meaningless unless we also save our code dictionary.

字典也要保存下来，否则 14 bit 毫无意义

So, we'll need to append it to the front of the image data, like this.

所以我们把字典 加到 14 bit 前面，就像这样

Now, including the dictionary, our image data is 30 bytes long.

现在加上字典，图像是 30 个字节(bytes) \N 比 48 字节好很多

That's still a significant improvement over 48 bytes.

现在加上字典，图像是 30 个字节(bytes) \N 比 48 字节好很多

The two approaches we discussed,

"消除冗余"和"用更紧凑的表示方法"，这两种方法通常会组合使用

removing redundancies and using more compact representations, are often combined,

"消除冗余"和"用更紧凑的表示方法"，这两种方法通常会组合使用

and underlie almost all lossless compressed file formats,

几乎所有无损压缩格式都用了它们 \N 比如 GIF, PNG, PDF, ZIP

like GIF, PNG, PDF and ZIP files.

几乎所有无损压缩格式都用了它们 \N 比如 GIF, PNG, PDF, ZIP

Both run-length encoding and dictionary coders are lossless compression techniques.

游程编码 和 字典编码 都是无损压缩

No information is lost; when you decompress, you get the original file.

压缩时不会丢失信息，解压后，数据和之前完全一样

That's really important for many types of files.

无损对很多文件很重要

Like, it'd be very odd if I zipped up a word document to send to you,

比如我给你发了个压缩的 word 文档\N 你解压之后发现内容变了，这就很糟糕了

and when you decompressed it on your computer, the text was different.

比如我给你发了个压缩的 word 文档\N 你解压之后发现内容变了，这就很糟糕了

But, there are other types of files where we can get away with little changes, perhaps

但其他一些文件，丢掉一些数据没什么关系

by removing unnecessary or less important information, especially information

丢掉那些人类看不出区别的数据

that human perception is not good at detecting.

丢掉那些人类看不出区别的数据

And this trick underlies most lossy compression techniques.

大多数有损压缩技术，都用到了这点

These tend to be pretty complicated, so we're going to attack this at a conceptual level.

实际细节比较复杂，所以我们讲概念就好

Let's take sound as an example.

以声音为例，你的听力不是完美的

Your hearing is not perfect.

以声音为例，你的听力不是完美的

We can hear some frequencies of sound better than others.

有些频率我们很擅长，其他一些我们根本听不见，比如超声波

And there are some we can't hear at all, like ultrasound.

有些频率我们很擅长，其他一些我们根本听不见，比如超声波

Unless you're a bat.

除非你是蝙蝠

Basically, if we make a recording of music, and there's data in the ultrasonic frequency range,

举个例子，如果录音乐，超声波数据都可以扔掉 \N 因为人类听不到超声波

we can discard it, because we know that humans can't hear it.

举个例子，如果录音乐，超声波数据都可以扔掉 \N 因为人类听不到超声波

On the other hand, humans are very sensitive to frequencies in the vocal range, like people

另一方面，人类对人声很敏感，所以应该尽可能保持原样

singing, so it's best to preserve quality there as much as possible.

另一方面，人类对人声很敏感，所以应该尽可能保持原样

Deep bass is somewhere in between.

低音介于两者之间，人类听得到，但不怎么敏感

Humans can hear it, but we're less attuned to it.

低音介于两者之间，人类听得到，但不怎么敏感

We mostly sense it.

一般是感觉到震动

Lossy audio compressors takes advantage of this, and encode different

有损音频压缩利用这一点，用不同精度编码不同频段

frequency bands at different precisions.

有损音频压缩利用这一点，用不同精度编码不同频段

Even if the result is rougher, it's likely that users won't perceive the difference.

听不出什么区别，不会明显影响体验

Or at least it doesn't dramatically affect the experience.

听不出什么区别，不会明显影响体验

And here comes the hate mail from the audiophiles!

音乐发烧友估计要吐槽了！

You encounter this type of audio compression all the time.

日常生活中你会经常碰到这类音频压缩

It's one of the reasons you sound different on a cellphone versus in person.

所以你在电话里的声音 和现实中不一样

The audio data is being compressed, allowing more people to take calls at once.

压缩音频是为了让更多人能同时打电话

As the signal quality or bandwidth get worse, compression algorithms remove more data,

如果网速变慢了，压缩算法会删更多数据

further reducing precision, which is why Skype calls sometimes sound like robots talking.

进一步降低声音质量，所以 Skype 通话有时听起来像机器人

Compared to an uncompressed audio format, like a WAV or FLAC (there we go, got the audiophiles back)

和没压缩的音频格式相比，比如 WAV 或 FLAC \N ( 这下音乐发烧友满意了）

compressed audio files, like MP3s, are often 10 times smaller.

压缩音频文件如 MP3，能小10倍甚至更多.

That's a huge saving!

省了超多空间！

And it's why I've got a killer music collection on my retro iPod.

所以我的旧 iPod 上有一堆超棒的歌

Don't judge.

别批判我

This idea of discarding or reducing precision in a manner that aligns with human perception

这种删掉人类无法感知的数据的方法，叫"感知编码"

is called perceptual coding,

这种删掉人类无法感知的数据的方法，叫"感知编码"

and it relies on models of human perception,

它依赖于人类的感知模型，模型来自"心理物理学"领域

which come from a field of study called Psychophysics.

它依赖于人类的感知模型，模型来自"心理物理学"领域

This same idea is the basis of lossy compressed image formats, most famously JPEGs.

这是各种"有损压缩图像格式"的基础，最著名的是 JPEG

Like hearing, the human visual system is imperfect.

就像听力一样，人的视觉系统也不是完美的.

We're really good at detecting sharp contrasts, like the edges of objects,

我们善于看到尖锐对比，比如物体的边缘

but our perceptual system isn't so hot with subtle color variations.

但我们看不出颜色的细微变化

JPEG takes advantage of this by breaking images up into blocks of 8x8 pixels,

JPEG 利用了这一点，把图像分解成 8x8 像素块

then throwing away a lot of the high-frequency spatial data.

然后删掉大量高频率空间数据

For example, take this photo of our directors dog - Noodle.

举个例子，这是导演的狗，面面

So cute!

超可爱！

Let's look at a patch of 8x8 pixels.

我们来看其中一个 8x8 像素

Pretty much every pixel is different from its neighbor,

几乎每个像素都和相邻像素不同，用无损技术很难压缩 \N 因为太多不同点了

making it hard to compress with loss-less techniques because there's just a lot going on.

几乎每个像素都和相邻像素不同，用无损技术很难压缩 \N 因为太多不同点了

Lots of little details.

很多小细节

But human perception doesn't register all those details.

但人眼看不出这些细节

So, we can discard a lot of that detail, and replace it with a simplified patch like this.

因此可以删掉很多，用这样一个简单的块来代替

This maintains the visual essence, but might only use 10% of the data.

这看起来一样，但可能只占10％的原始数据

We can do this for all the patches in the image and get this result.

我们可以对所有 8x8 块做一样的操作

You can still see it's a dog, but the image is rougher.

图片依然可以认出是一只狗，只是更粗糙一些

So, that's an extreme example, going from a slightly compressed JPEG to a highly compressed one,

以上例子比较极端，进行了高度压缩，只有原始大小的八分之一

one-eighth the original file size.

以上例子比较极端，进行了高度压缩，只有原始大小的八分之一

Often, you can get away with a quality somewhere in between, and perceptually,

通常你可以取得平衡，图片看起来差不多，但文件小不少

it's basically the same as the original.

通常你可以取得平衡，图片看起来差不多，但文件小不少

Can you tell the difference between the two?

你看得出两张图的区别吗？

Probably not, but I should mention that video compression plays a role in that too,

估计看不出，但我想提一下，视频压缩也造成了影响

since I'm literally being compressed in a video right now.

毕竟你现在在看视频啊

Videos are really just long sequences of images, so a lot of what I said

视频只是一长串连续图片 \N 所以图片的很多方面也适用于视频

about them applies here too.

视频只是一长串连续图片 \N 所以图片的很多方面也适用于视频

But videos can do some extra clever stuff, because between frames,

但视频可以做一些小技巧 \N 因为帧和帧之间很多像素一样

a lot of pixels are going to be the same.

但视频可以做一些小技巧 \N 因为帧和帧之间很多像素一样

Like this whole background behind me!

比如我后面的背景！

This is called temporal redundancy.

这叫 时间冗余

We don't need to re-transmit those pixels every frame of the video.

视频里不用每一帧都存这些像素 \N 可以只存变了的部分

We can just copy patches of data forward.

视频里不用每一帧都存这些像素 \N 可以只存变了的部分

When there are small pixel differences, like the readout on this frequency generator behind me,

当帧和帧之间有小小的差异时，比如后面这个频率发生器

most video formats send data that encodes just the difference between patches,

很多视频编码格式，只存变化的部分

which is more efficient than re-transmitting all the pixels afresh, again taking advantage

这比存所有像素更有效率 \N 利用了帧和帧之间的相似性

of inter-frame similarity.

这比存所有像素更有效率 \N 利用了帧和帧之间的相似性

The fanciest video compression formats go one step further.

更高级的视频压缩格式 会更进一步

They find patches that are similar between frames, and not only copy them forward, with

找出帧和帧之间相似的补丁 \N 然后用简单效果实现，比如移动和旋转

or without differences, but also can apply simple effects to them, like a shift or rotation.

找出帧和帧之间相似的补丁 \N 然后用简单效果实现，比如移动和旋转

They can also lighten or darken a patch between frames.

变亮和变暗

So, if I move my hand side to side like this the video compressor will identify the similarity,

如果我这样摆手，视频压缩器会识别到相似性

capture my hand in one or more patches, then just move these patches around between frames.

用一个或多个补丁代表我的手，然后帧之间直接移动这些补丁

You're actually seeing my hand from the past kinda freaky, but it uses a lot less data.

所以你看到的是我过去的手（不是实时的）\N 有点可怕 但数据量少得多

MPEG-4 videos, a common standard, are often 20 to 200 times

MPEG-4 是常见标准，可以比原文件小20倍到200倍

smaller than the original, uncompressed file.

MPEG-4 是常见标准，可以比原文件小20倍到200倍

However, encoding frames as translations and rotations of patches from previous frames

但用补丁的移动和旋转 来更新画面

can go horribly wrong when you compress too heavily, and there isn't

当压缩太严重时会出错 \N 没有足够空间更新补丁内的像素

enough space to update pixel data inside of the patches.

当压缩太严重时会出错 \N 没有足够空间更新补丁内的像素

The video player will forge ahead, applying the right motions,

即使补丁是错的，视频播放器也会照样播放

even if the patch data is wrong.

即使补丁是错的，视频播放器也会照样播放

And this leads to some hilarious and trippy effects, which I'm sure you've seen.

导致一些怪异又搞笑的结果，你肯定见过这些.

Overall, it's extremely useful to have compression techniques for all the types of data I discussed today.

总的来说，压缩对大部分文件类型都有用

(I guess our imperfect vision and hearing are "useful," too.)

从这个角度来讲，人类不完美的视觉和听觉 也算有用

And it's important to know about compression because it allows users to

学习压缩非常重要 \N 因为可以高效存储图片，音乐，视频

store pictures, music, and videos in efficient ways.

学习压缩非常重要 \N 因为可以高效存储图片，音乐，视频

Without it, streaming your favorite Carpool Karaoke videos on YouTube would be nearly impossible,

如果没有压缩 \N 在 YouTube 看"明星拼车唱歌"几乎不可能

due to bandwidth and the economics of transmitting that volume of data for free.

因为你的带宽可能不够（会很卡） \N 而且供应商不愿意免费传输那么多数据

And now when your Skype calls sound like they're being taken over by demons,

现在你知道为什么打 Skype 电话 \N 有时像在和恶魔通话

you'll know what's really going on.

现在你知道为什么打 Skype 电话 \N 有时像在和恶魔通话

I'll see you next week.

下周见

Hi, I'm Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

We've talked a lot about inputs and outputs in this series,

我们之前讨论过 输入输出 \N 但都是计算机组件互相输入输出

but they've mostly been between different parts of a computer

我们之前讨论过 输入输出 \N 但都是计算机组件互相输入输出

like outputting data from RAM or inputting instructions to a CPU.

比如 RAM 输出数据，或输指令进 CPU

We haven't discussed much about inputs coming from humans.

我们还没讲 来自人类的输入

We also haven't learned how people get information out of a computer,

也没讲怎么从电脑中拿出信息，除了用打孔纸卡

other than by printing or punching it onto paper.

也没讲怎么从电脑中拿出信息，除了用打孔纸卡

Of course, there's a wide variety of input and output devices that

当然，有很多种 "输入输出设备" \N 让我们和计算机交互

allow us users to communicate with computers.

当然，有很多种 "输入输出设备" \N 让我们和计算机交互

They provide an interface between human and computer.

它们在人类和机器间提供了界面

And today, there's a whole field of study called Human-Computer Interaction.

如今有整个学科专门研究这个，叫 "人机交互"

These interfaces are so fundamental to the user experience

界面对用户体验非常重要 \N 所以是我们接下来几集的重点

that they're the focus of the next few episodes.

界面对用户体验非常重要 \N 所以是我们接下来几集的重点

As we discussed at the very beginning of the series,

在系列开头的几集，我们提过

the earliest mechanical and electro-mechanical computing devices

早期机械计算设备 \N 用齿轮，旋钮和开关等机械结构来输入输出

used physical controls for inputs and outputs, like gears, knobs and switches,

早期机械计算设备 \N 用齿轮，旋钮和开关等机械结构来输入输出

and this was pretty much the extent of the human interface.

这些就是交互界面

Even the first electronic computers, like Colossus and ENIAC,

甚至早期电子计算机 比如 Colossus 和 ENIAC

were configured using huge panels of mechanical controls and patch wires.

也是用一大堆机械面板和线 来操作

It could take weeks to enter in a single program, let alone run it,

输入一个程序可能要几星期，还没提运行时间.

and to get data out after running a program, results were most often printed to paper.

运行完毕后想拿出数据，一般是打印到纸上

Paper printers were so useful

打印机超有用

that even Babbage designed one for his Difference Engine,

甚至 查尔斯·巴贝奇 都给差分机专门设计了一个

and that was in the 1820s!

那可是 1820 年代!

However, by the 1950s, mechanical inputs were rendered obsolete

然而，到 1950 年代，机械输入完全消失

by programs and data stored entirely on mediums like punch cards and magnetic tape.

因为出现了打孔纸卡和磁带

Paper printouts were still used for the final output,

但输出仍然是打印到纸上

and huge banks of indicator lights were developed

还有大量指示灯，在运行中提供实时反馈

to provide real time feedback while the program was in progress.

还有大量指示灯，在运行中提供实时反馈

It's important to recognize that computer input of this era was

那个时代的特点是 尽可能迁就机器 \N 对人类好不好用是其次

designed to be as simple and robust as possible for computers.

那个时代的特点是 尽可能迁就机器 \N 对人类好不好用是其次

Ease and understanding for users was a secondary concern.

那个时代的特点是 尽可能迁就机器 \N 对人类好不好用是其次

Punch tape is a great example

打孔纸带就是个好例子

this was explicitly designed to be easy for computers to read.

就是为了方便计算机读取

The continuous nature of tape made it easy to handle mechanically,

纸带是连续的，方便机器处理

and the holes could be reliably detected with a mechanical or optical system,

纸孔可以方便地 用机械或光学手段识别

which encoded instructions and data.

纸孔可以编码程序和数据

But of course, humans don't think in terms of little punched holes on strips of paper.

当然, 人类不是以纸孔方式思考的.

So, the burden was on programmers.

所以负担放到了程序员身上

They had to spend the extra time and effort to convert their ideas and programs

他们要花额外时间和精力 \N 转成计算机能理解的格式

into a language and a format that was easy for computers of the era to understand

他们要花额外时间和精力 \N 转成计算机能理解的格式

often with the help of additional staff and auxiliary devices.

一般需要额外人员和设备帮忙

It's also important to note that early computers, basically pre-1950,

要注意的是，基本上 1950 年前的早期计算机 \N "输入"的概念很原始

had an extremely simple notion of human input.

要注意的是，基本上 1950 年前的早期计算机 \N "输入"的概念很原始

Yes, humans input programs and data into computers,

是的，的确是人类负责输入程序和数据 \N 但计算机不会交互式回应

but these machines generally didn't respond interactively to humans.

是的，的确是人类负责输入程序和数据 \N 但计算机不会交互式回应

Once a program was started, it typically ran until it was finished.

程序开始运行后会一直运行 直到结束

That's because these machines were way too expensive to be

因为机器太贵了 \N 不能等人类慢慢敲命令和给数据

waiting around for humans to type a command or enter data.

因为机器太贵了 \N 不能等人类慢慢敲命令和给数据

Any input needed for a computation was fed in at the same time as the program.

要同时放入程序和数据

This started to change in the late 1950s.

这在 1950 年代晚期开始发生变化

On one hand, smaller-scale computers started to become cheap enough

一方面，小型计算机变得足够便宜

that it was feasible to have a human-in-the loop;

让人类来回和计算机交互 变得可以接受

that is, a back and forth between human and computer.

交互式就是人和计算机之间来回沟通

And on the other hand,

而另一方面

big fancy computers became fast and sophisticated enough to support many programs and users at once,

大型计算机变得更快，能同时支持多个程序和多个用户

what were called multitasking and time-sharing systems .

这叫"多任务"和"分时系统"

But these computers needed a way to get input from users.

但交互式操作时 \N 计算机需要某种方法来获得用户输入

For this, computers borrowed the ubiquitous data entry mechanism of the era: keyboards.

所以借用了当时已经存在的数据录入机制：键盘

At this point, typing machines had already been in use for a few centuries,

当时，打字机已经存在几个世纪了

but it was Christopher Latham Sholes, who invented the modern typewriter in 1868.

但现代打字机是 \N克里斯托弗·莱瑟姆·肖尔斯 在 1868 年发明的

It took until 1874 to refine the design and manufacture it,

虽然到 1874 年才完成设计和制造

but it went on to be a commercial success.

但之后取得了商业成功

Sholes' typewriter adopted an unusual keyboard layout that you know well QWERTY

肖尔斯 的打字机用了不寻常的布局，QWERTY

named for the top-left row of letter keys.

名字来自键盘左上角按键

There has been a lot of speculation as to why this design was used.

为什么这么设计 有很多猜测

The most prevalent theory is that it put common letter pairings in English far apart

最流行的理论是 这样设计是为了 \N 把常见字母放得远一些，避免按键卡住

to reduce the likelihood of typebars jamming when entered in sequence.

最流行的理论是 这样设计是为了 \N 把常见字母放得远一些，避免按键卡住

It's a convenient explanation, but it's also probably false,

这个解释虽然省事，但可能是错的，或至少不够全面

or at least not the full story.

这个解释虽然省事，但可能是错的，或至少不够全面

In fact, QWERTY puts many common letter pairs together,

事实上，QWERTY 把很多常见字母放在了一起 \N 比如 TH 和 ER

like "TH" and "ER".

事实上，QWERTY 把很多常见字母放在了一起 \N 比如 TH 和 ER

And we know that Sholes and his team went through many iterations

我们知道 肖尔斯和他的团队设计了很多版 \N 才进化到这个布局

before arriving at this iconic arrangement.

我们知道 肖尔斯和他的团队设计了很多版 \N 才进化到这个布局

Regardless of the reason, the commercial success of Sholes' typewriter meant

总之，肖尔斯 的打字机取得了成功 \N 所以其它公司很快开始抄他的设计

the competitor companies that soon followed duplicated his design.

总之，肖尔斯 的打字机取得了成功 \N 所以其它公司很快开始抄他的设计

Many alternative keyboard layouts have been proposed over the last century,

过去一个世纪有不少新的键盘布局被发明，宣称各种好处

claiming various benefits.

过去一个世纪有不少新的键盘布局被发明，宣称各种好处

But, once people had invested the time to learn QWERTY,

但人们已经熟悉了 QWERTY 布局 \N 根本不想学新布局

they just didn't want to learn something new.

但人们已经熟悉了 QWERTY 布局 \N 根本不想学新布局

This is what economists would call a switching barrier or switching cost.

这是经济学家所说的 转换成本

And it's for this very basic human reason

所以现在都快1个半世纪了 \N 我们还在用 QWERTY 键盘布局

that we still use QWERTY keyboards almost a century and a half later!

所以现在都快1个半世纪了 \N 我们还在用 QWERTY 键盘布局

I should mention that QWERTY isn't universal.

我应该提一下，QWERTY 不是通用的

There are many international variants,

有很多变体，比如法国 AZERTY 布局

like the French AZERTY layout,

有很多变体，比如法国 AZERTY 布局

or the QWERTZ layout common in central Europe.

以及中欧常见的 QWERTZ 布局

Interestingly, Sholes didn't envision that typing would ever be faster than handwriting,

有趣的是，肖尔斯 根本没想到打字会比手写快

which is around 20 words per minute.

手写速度大约是每分钟 20 个

Typewriters were introduced chiefly for legibility and standardization of documents, not speed.

打字机主要为了易读性和标准化，而不是速度

However, as they became standard equipment in offices, the desire for speedy typing grew,

然而随着打字机成为办公室标配 \N 对快速打字的渴望越来越大

and there were two big advances that unlocked typing's true potential.

有两个重大进步 解放了打字的潜力

Around 1880, Elizabeth Longley, a teacher at the Cincinnati Shorthand and Type-Writer Institute,

1880年左右，辛辛那提速记学院 \N 一名叫 伊丽莎白·朗利 的老师

started to promote ten-finger typing.

开始推广 十指打字

This required much less finger movement than hunt-and-peck,

比一个手指打字要移动的距离短得多，所以速度更快

so it offered enhanced typing speeds.

比一个手指打字要移动的距离短得多，所以速度更快

Then, a few years later, Frank Edward McGurrin, a federal court clerk in Salt Lake City,

几年后，弗兰克·爱德华·麦克格林 \N 盐湖城的一位联邦法庭书记

taught himself to touch-type; as in, he didn't need to look at the keys while typing.

学会了盲打，打字时不用看键盘

In 1888, McGurrin won a highly publicized typing-speed contest,

1888年，麦格高林 赢了备受关注的打字速度比赛

after which ten-finger, touch-typing began to catch on.

之后"十指盲打"开始流行

Professional typists were soon able to achieve speeds upwards of 100 words per minute,

专业打字员 每分钟 100 字以上

much faster than handwriting!

比手写快多了！而且清晰又整洁！

And nice and neat too!

比手写快多了！而且清晰又整洁！

So, humans are pretty good with typewriters,

虽然人类擅长用打字机

but we can't just plunk down a typewriter in front of a computer and have it type

但我们没法把打字机塞到计算机面前，让它打字

they have no fingers!

计算机又没有手指

Instead, early computers adapted a special type of typewriter that was used for telegraphs,

所以早期计算机用了一种特殊打字机，是专门用来发电报的,

called a teletype machine.

叫 电传打字机

These were electromechanically-augmented typewriters

这些打字机是强化过的，可以用电报线发送和接收文本

that can send and receive text over telegraph lines.

这些打字机是强化过的，可以用电报线发送和接收文本

Pressing a letter on one teletype keyboard would cause a signal to be sent,

按一个字母，信号会通过电报线，发到另一端

over telegraph wires, to a teletype machine on the other end,

按一个字母，信号会通过电报线，发到另一端

which would then electromechanically type that letter.

另一端的电传打字机会打出来

This allowed two humans to type to one another over long distances.

使得两人可以长距离沟通

basically a steampunk version of a chat room.

基本是个蒸汽朋克版聊天室

Since these teletype machines already had an electronic interface,

因为电传打字机有电子接口，稍作修改就能用于计算机

they were easily adapted for computer use,

因为电传打字机有电子接口，稍作修改就能用于计算机

and teletype computer interfaces were common in the 1960s and 70s.

电传交互界面在 1960~1970 很常见

Interaction was pretty straightforward.

用起来很简单

Users would type a command, hit enter, and then the computer would type back.

输入一个命令，按回车，然后计算机会输回来

This text "conversation" between a user and a computer went back and forth.

用户和计算机来回"对话"

These were called command line interfaces,

这叫"命令行界面"

and they remained the most prevalent form of human-computer interaction

它是最主要的人机交互方式，一直到 1980 年代

up until around the 1980s.

它是最主要的人机交互方式，一直到 1980 年代

Command Line interaction on a teletype machine looks something like this.

用电传打字机的命令行交互 类似这样：

A user can type any number of possible commands.

用户可以输入各种命令

Let's check out a few,

我们来看几个命令，先看当前目录有什么文件

beginning with seeing all of the files in the current directory we're in.

我们来看几个命令，先看当前目录有什么文件

For this, we would type the command, "ls", which is short for list,

输入命令 ls，名字来自 list 的缩写

and the computer replies with a list of the files in our current directory.

然后计算机会列出 当前目录里的所有文件

If we want to see what's in our "secretStarTrekDiscoveryCast.txt.txt file",

如果想看 secretStarTrekDiscoveryCast.txt 有什么

we use yet another command to display the contents.

要用另一个命令 显示文件内容

In unix, we can call "cat" - short for concatenate.

unix 用 cat 命令显示文件内容 \N cat 是连接(concatenate)的缩写

We need to specify which file to display, so we include that after the command, called an argument.

然后指定文件名，指定的方法是写在 cat 命令后面 \N 传给命令的值叫 参数

If you're connected to a network with other users,

如果同一个网络里有其他人

you can use a primitive version of a Find My Friends app

你可以用 finger 命令找朋友 \N 就像是个很原始的"找朋友" App

to get more info on them with the command "finger".

你可以用 finger 命令找朋友 \N 就像是个很原始的"找朋友" App

Electromechanical teletype machines

电传打字机 直到1970年代左右都是主流交互方式

were the primary computing interface for most users up until around the 1970s.

电传打字机 直到1970年代左右都是主流交互方式

Although computer screens first emerged in the 1950s,

尽管屏幕最早出现在 1950 年代 \N 但对日常使用太贵 而且分辨率低

and were used for graphics they were too expensive and low resolution for everyday use.

尽管屏幕最早出现在 1950 年代 \N 但对日常使用太贵 而且分辨率低

However, mass production of televisions for the consumer market, and general improvements

然而因为针对普通消费者的电视机开始量产 \N 同时处理器与内存也在发展

in processors and memory, meant that by 1970, it was economically viable

到1970年代，屏幕代替电传打字机 变得可行

to replace electromechanical teletype machines with screen-based equivalents.

到1970年代，屏幕代替电传打字机 变得可行

But rather than build a whole new standard to interface computers with these screens,

但与其为屏幕 专门做全新的标准

engineers simply recycled the existing text-only, teletype protocol.

工程师直接用现有的 电传打字机协议

These machines used a screen, which simulated endless paper.

屏幕就像无限长度的纸 \N 除了输入和输出字，没有其它东西

It was text in and text out, nothing more.

屏幕就像无限长度的纸 \N 除了输入和输出字，没有其它东西

The protocol was identical, so computers couldn't even tell if it was paper or a screen.

协议是一样的，所以计算机分不出是纸还是屏幕

These virtual teletype or glass teletype machines became known as terminals .

这些"虚拟电传打字机"或"玻璃电传打字机"\N 叫 终端

By 1971, it was estimated, in the United States,

到1971年，美国大约有 7 万台电传打字机 \N 以及 7 万个终端

there was something on the order of 70,000 electro-mechanical teletype machines

到1971年，美国大约有 7 万台电传打字机 \N 以及 7 万个终端

and 70,000 screen-based terminals in use.

到1971年，美国大约有 7 万台电传打字机 \N 以及 7 万个终端

Screens were so much better, faster and more flexible, though.

屏幕又好又快又灵活

Like, you could delete a mistake and it would disappear.

如果删一个错别字 会立刻消失

So, by the end of the 1970s, screens were standard.

所以到 1970 年代末 屏幕成了标配

You might think that command line interfaces are way too primitive to do anything interesting.

你也许会想，命令行界面太原始了 \N 做不了什么有意思的事

But even when the only interaction was through text, programmers found a way to make it fun.

即便只有文字 \N 程序员也找到了一些方法，让它变得有趣一些

Early interactive, text-based computer games include famous titles like Zork,

早期的著名交互式文字游戏 Zork

created in 1977.

出现于 1977 年

Players of these sorts of early games were expected to engage their limitless imaginations

早期游戏玩家需要丰富的想象力

as they visualized the fictional world around them, like what terrifying monster confronted them

想像自己身在虚构世界，比如"四周漆黑一片

when it was pitch black and you were likely to be eaten by a grue.

附近可能有怪物会吃掉你"

Let's go back to our command line, now on a fancy screen-based terminal, and play!

我们用命令行玩玩看

Just like before, we can see what's in our current directory with the "ls" command.

就像之前，我们可以用 ls 命令，看当前目录有什么

Then, let's go into our games directory by using the "cd" command, for " change directory ".

然后用 cd 命令，进入游戏文件夹 \N cd 的意思是 "改变文件夹"

Now, we can use our "ls" command again to see what games are installed on our computer.

再用 ls 看有哪些游戏

Sweet, we have Adventure!

超棒！我们有"冒险旅程"！(adventure)

All we have to do to run this program is type its name.

想运行这个程序，只需要输入它的名字

Until this application halts, or we quit it, it takes over the command line.

在程序自行停止或我们主动退出前 \N 它会接管命令行

What you're seeing here is actual interaction from "Colossal Cave Adventure",

你现在看到的，是"巨大洞穴冒险"这款游戏的真实输出

first developed by Will Crowther in 1976.

由 Will Crowther 在 1976 年开发

In the game, players can type in one- or two-word commands to move around,

游戏中，玩家可以输入1个词或2个词的命令 \N 来移动人物，和其他东西交互，捡物品等

interact with objects, pickup items and so on.

游戏中，玩家可以输入1个词或2个词的命令 \N 来移动人物，和其他东西交互，捡物品等

The program acts as the narrator, describing locations, possible actions,

然后游戏会像旁白一样，输出你的当前位置， \N 告诉你能做什么动作，以及你的动作造成的结果

and the results of those actions.

然后游戏会像旁白一样，输出你的当前位置， \N 告诉你能做什么动作，以及你的动作造成的结果

Certain ones resulted in death!

有些动作会导致死亡！

The original version only had 66 locations to explore,

原始版本只有 66 个地方可供探索

but it's widely considered to be the first example of interactive fiction.

但它被广泛认为是最早的互动式小说

These text adventure games later became multiplayer, called MUDs or Multi-User Dungeons.

游戏后来从纯文字进化成多人游戏 \N 简称 MUD，或多人地牢游戏

And they're the great-forbearers of the awesome graphical MMORPG's

是如今 MMORPG 的前辈 \N （大型多人在线角色扮演游戏）

(massive, multiplayer online role playing games) we enjoy today.

是如今 MMORPG 的前辈 \N （大型多人在线角色扮演游戏）

And if you want to know more about the history of these and other games

如果你想了解游戏史，我们有游戏速成课 \N 主持人 Andre Meadows

we've got a whole series on it hosted by Andre Meadows!

如果你想了解游戏史，我们有游戏速成课 \N 主持人 Andre Meadows

Command line interfaces, while simple, are very powerful.

命令行界面虽然简单 但十分强大

Computer programming is still very much a written task, and as such,

编程大部分依然是打字活 \N 所以用命令行比较自然

command lines are a natural interface.

编程大部分依然是打字活 \N 所以用命令行比较自然

For this reason, even today, most programmers use

因此，即使是现在 \N 大多数程序员工作中依然用命令行界面

command line interfaces as part of their work.

因此，即使是现在 \N 大多数程序员工作中依然用命令行界面

And they're also the most common way to access computers that are far away,

而且用命令行访问远程计算机 是最常见的方式 \N 比如服务器在另一个国家

like a server in a different country.

而且用命令行访问远程计算机 是最常见的方式 \N 比如服务器在另一个国家

If you're running Windows, macOS or Linux,

如果你用 Windows, macOS, Linux

your computer has a command line interface one you may have never used.

你的计算机有命令行界面，但你可能从来没用过

Check it out by typing "cmd" in your Windows search bar,

你可以在 Windows 搜索栏中输入 cmd

or search for Terminal on Mac.

或在 Mac 上搜 Terminal

Then install a copy of Zork and play on!

然后你可以装 Zork 玩！

So, you can see how these early advancements still have an impact on computing today.

现在你知道了 \N 早期计算机的发展是如何影响到现在的.

Just imagine if your phone didn't have a good ol' fashioned QWERTY keyboard.

想想要是手机没有 QWERTY 键盘 \N 在 Instagram 给图片配标题可就麻烦了

It could take forever to type your Instagram captions.

想想要是手机没有 QWERTY 键盘 \N 在 Instagram 给图片配标题可就麻烦了

But, there's still something missing from our discussion.

但我们还有一个重要话题没讲

All the sweet sweet graphics!

美妙的图形界面！

That's our topic for next week.

这是下周的主题

See you soon.

下周见

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

This 1960 PDP-1 is a great example of early computing with graphics.

这台 1960 年的 PDP-1 \N 是一个早期图形计算机的好例子

You can see a cabinet-sized computer on the left,

你可以看到 左边是柜子大小的电脑

an electromechanical teletype machine in the middle,

中间是电传打字机

and a round screen on the right.

右边是一个圆形的屏幕

Note how they're separated.

注意它们是分开的

That's because text-based tasks and graphical tasks were often distinct back then.

因为当时文本任务和图形任务是分开的.

In fact, these early computer screens had a very hard time rendering crisp text, whereas

事实上，早期的屏幕无法显示清晰的文字

typed paper offered much higher contrast and resolution.

而打印到纸上 有更高的对比度和分辨率

The most typical use for early computer screens was to keep track of a program's operation,

早期屏幕的典型用途 是跟踪程序的运行情况

like values in registers.

比如寄存器的值

It didn't make sense to have a teletype machine print this on paper

如果用打印机 \N 一遍又一遍打印出来没有意义

over and over and over again -- that'd waste a lot of paper, and it was slow.

不仅费纸而且慢

On the other hand, screens were dynamic and quick to update -- perfect for temporary values.

另一方面，屏幕更新很快，对临时值简直完美

Computer screens were rarely considered for program output, though.

但屏幕很少用于输出计算结果，结果一般都打印到纸上

Instead, any results from a computation were typically written to paper

但屏幕很少用于输出计算结果，结果一般都打印到纸上

or some other more permanent medium.

或其它更永久的东西上

But, screens were so darn useful

但屏幕超有用 \N 到1960年代，人们开始用屏幕做很多酷炫的事情

that by the early 1960s, people started to use them for awesome things.

但屏幕超有用 \N 到1960年代，人们开始用屏幕做很多酷炫的事情

A lot of different display technologies have been created over the decades,

几十年间出现了很多显示技术

but the most influential, and also the earliest, were Cathode Ray Tubes, or CRT

但最早最有影响力的是 阴极射线管（CRT）

These work by shooting electrons out of an emitter at a phosphor-coated screen.

原理是把电子发射到 有磷光体涂层的屏幕上

When electrons hit the coating, it glows for a fraction of a second.

当电子撞击涂层时 会发光几分之一秒

Because electrons are charged particles,

由于电子是带电粒子，路径可以用磁场控制

their paths can be manipulated with electromagnetic fields.

由于电子是带电粒子，路径可以用磁场控制

Plates or coils are used inside to steer electrons to a desired position,

屏幕内用板子或线圈 把电子引导到想要的位置

both left-right and up-down.

上下左右都行

With this control, there are two ways you can draw graphics.

既然可以这样控制，有 2 种方法绘制图形 \N 1. 引导电子束描绘出形状

The first option is to direct the electron beam to trace out shapes.

既然可以这样控制，有 2 种方法绘制图形 \N 1. 引导电子束描绘出形状

This is called Vector Scanning.

这叫"矢量扫描"

Because the glow persists for a little bit, if you repeat the path quickly enough,

因为发光只持续一小会儿 \N 如果重复得足够快 可以得到清晰的图像

you create a solid image.

因为发光只持续一小会儿 \N 如果重复得足够快 可以得到清晰的图像

The other option is to repeatedly follow a fixed path, scanning line by line,

2. 按固定路径，一行行来 \N 从上向下，从左到右，不断重复

from top left to bottom right, and looping over and over again.

2. 按固定路径，一行行来 \N 从上向下，从左到右，不断重复

You only turn on the electron beam at certain points to create graphics.

只在特定的点打开电子束，以此绘制图形

This is called Raster Scanning.

这叫 "光栅扫描"

With this approach, you can display shapes... and even text... all made of little line segments.

用这种方法，可以用很多小线段绘制形状 甚至文字

Eventually, as display technologies improved,

最后，因为显示技术的发展

it was possible to render crisp dots onto the screen, aka pixels.

我们终于可以在屏幕上显示清晰的点，叫"像素"

The Liquid Crystal Displays, or LCDs,

液晶显示器，简称 LCD

that we use today are quite a different technology.

和以前的技术相当不同

But, they use raster scanning too,

但 LCD 也用光栅扫描 \N 每秒更新多次 像素里红绿蓝的颜色

updating the brightness of little tiny red, green and blue pixels many times a second.

但 LCD 也用光栅扫描 \N 每秒更新多次 像素里红绿蓝的颜色

Interestingly, most early computers didn't use pixels

有趣的是，很多早期计算机不用像素

-- not because they couldn't physically,

- 不是技术做不到, 而是因为像素占太多内存

but because it consumed way too much memory for computers of the time.

- 不是技术做不到, 而是因为像素占太多内存

A 200 by 200 pixel image contains 40,000 pixels.

200像素×200像素的图像，有 40,000 个像素

Even if you use just one bit of data for each pixel,

哪怕每个像素只用一个 bit 表示 \N 代表黑色或白色，连灰度都没有！

that's black OR white -- not grayscale!

哪怕每个像素只用一个 bit 表示 \N 代表黑色或白色，连灰度都没有！

-- the image would consume 40,000 bits of memory.

- 会占 40,000 bit 内存 \N 比 PDP-1 全部内存的一半还多

That would have gobbled up more than half of a PDP-1's entire RAM.

- 会占 40,000 bit 内存 \N 比 PDP-1 全部内存的一半还多

So, computer scientists and engineers had to come up with clever tricks to render graphics

所以计算机科学家和工程师，得想一些技巧来渲染图形 \N 等内存发展到足够用

until memory sizes caught up to our pixelicious ambitions.

所以计算机科学家和工程师，得想一些技巧来渲染图形 \N 等内存发展到足够用

Instead of storing tens of thousands of pixels,

所以早期计算机不存大量像素值 \N 而是存符号，80x25个符号最典型

early computers stored a much smaller grid of letters, most typically 80 by 25 characters.

所以早期计算机不存大量像素值 \N 而是存符号，80x25个符号最典型

That's 2000 characters in total.

总共 2000 个字符

And if each is encoded in 8 bits, using something like ASCII,

如果每个字符用 8 位表示，比如用 ASCII

it would consume 16,000 bits of memory for an entire screen full of text,

总共才 16000 位，这种大小更合理

which is way more reasonable.

总共才 16000 位，这种大小更合理

To pull this off, computers needed an extra piece of hardware that

为此，计算机需要额外硬件

could read characters out of RAM, and convert them into raster graphics to be drawn onto the screen.

来从内存读取字符，转换成光栅图形 \N 这样才能显示到屏幕上

This was called a character generator, and they were basically the first graphics cards.

这个硬件叫 "字符生成器"，基本算是第一代显卡

Inside, they had a little piece of Read Only Memory, a ROM,

它内部有一小块只读存储器，简称 ROM

that stored graphics for each character, called a dot matrix pattern.

存着每个字符的图形，叫"点阵图案"

If the graphics card saw the 8-bit code for the letter "K",

如果图形卡看到一个 8 位二进制，发现是字母 K

then it would raster scan the 2D pattern for the letter K onto the screen, in the appropriate position.

那么会把字母 K 的点阵图案 \N光栅扫描显示到屏幕的适当位置

To do this, the character generator had special access to a portion of a computer's memory

为了显示，"字符生成器" 会访问内存中一块特殊区域 \N 这块区域专为图形保留，叫 屏幕缓冲区

reserved for graphics, a region called the screen buffer.

为了显示，"字符生成器" 会访问内存中一块特殊区域 \N 这块区域专为图形保留，叫 屏幕缓冲区

Computer programs wishing to render text to the screen

程序想显示文字时，修改这块区域里的值就行

simply manipulated the values stored in this region,

程序想显示文字时，修改这块区域里的值就行

just as they could with any other data in RAM.

程序想显示文字时，修改这块区域里的值就行

This scheme required much less memory,

这个方案用的内存少得多 \N 但也意味着 只能画字符到屏幕上

but it also meant the only thing you could draw was text.

这个方案用的内存少得多 \N 但也意味着 只能画字符到屏幕上

Even still, people got pretty inventive with ASCII art!

即使有这样限制 \N 人们用 ASCII 艺术发挥了很多创意！

People also tried to make rudimentary, pseudo-graphical interfaces out of this basic set of characters

也有人用字符模仿图形界面

using things like underscores and plus signs to create boxes, lines and other primitive shapes.

用下划线和加号来画盒子，线，和其他简单形状

But, the character set was really too small to do anything terribly sophisticated.

但字符集实在太小，做不了什么复杂的事

So, various extensions to ASCII were made that added new semigraphical characters,

因此对 ASCII 进行了各种扩展，加新字符

like IBM's CP437 character set, seen here, which was used in DOS.

比如上图的 IBM CP437 字符集，用于 DOS.

On some systems, the text color and background color could be defined with a few extra bits.

某些系统上 \N 可以用额外的 bit 定义字体颜色和背景颜色

That allowed glorious interfaces like this DOS example,

做出这样的 DOS 界面 \N 这界面只用了刚刚提到的字符集

which is built entirely out the character set you just saw.

做出这样的 DOS 界面 \N 这界面只用了刚刚提到的字符集

Character generators were a clever way to save memory.

字符生成器 是一种省内存的技巧 \N 但没办法绘制任意形状

But, they didn't provide any way to draw arbitrary shapes.

字符生成器 是一种省内存的技巧 \N 但没办法绘制任意形状

And that's important if you want to draw content like electrical circuits, architectural

绘制任意形状很重要 \N 因为电路设计，建筑平面图，地图，好多东西都不是文字！

plans, maps, and... well pretty much everything that isn't text!

绘制任意形状很重要 \N 因为电路设计，建筑平面图，地图，好多东西都不是文字！

To do this, without resorting to memory-gobbling pixels,

为了绘制任意形状，同时不吃掉所有内存

computer scientists used the vector mode available on CRTs.

计算机科学家用 CRT 上的"矢量模式"

The idea is pretty straightforward: all content to be drawn on screen is defined by a series of lines.

概念非常简单：所有东西都由线组成

There's no text.

没有文字这回事

If you need to draw text, you have to draw it out of lines.

如果要显示文字，就用线条画出来

Don't read between the lines here. There is only lines!

只有线条，没有别的

Got it? Alright, no more word play.

明白了吗？好，我们举个实例吧

I'm drawing the line here.

明白了吗？好，我们举个实例吧

Let's pretend this video is a cartesian plane, 200 units wide and 100 tall, with the

假设这个视频是一个 笛卡尔平面 \N 200个单位宽，100个单位高

origin that's the zero-zero point in the upper left corner.

原点 (0,0) 在左上角

We can draw a shape with the following vector commands,

我们可以画形状，用如下矢量命令

which we've borrowed from the Vectrex, an early vector display system.

这些命令来自 Vectrex，一个早期矢量显示系统

First, we reset, which clears the screen,

首先，reset ，这个命令会清空屏幕

moves the drawing point of the electron gun to zero-zero,

把电子枪的绘图点移动到坐标 (0,0)

and sets the brightness of lines to zero.

并把线的亮度设为 0

Then we move the drawing point down to 50 50,

MOVE\_TO 50 50 \N 把绘图点移动到坐标 (50,50)

and set the line intensity to 100%.

INTENSITY 100 \N 把强度设为 100

With the intensity up, now we move to 100, 50, then 60, 75 and then back to 50,50.

现在亮度提高了\N 移动到 (100,50) 然后 (60,75) 然后 (50,50)

The last thing to do is set our line intensity back to 0%.

最后把强度设回 0

Cool! We've got a triangle!

酷，我们画了一个三角形！

This sequence of commands would consume on the order of 160 bits, which is way more efficient

这些命令占 160 bit \N 比存一个庞大的像素矩阵更好

than keeping a huge matrix of pixel values!

这些命令占 160 bit \N 比存一个庞大的像素矩阵更好

Just like how characters were stored in memory and turned into graphics by a character generator,

就像之前的"字符生成器" \N把内存里的字符转成图形一样

these vector instructions were also stored in memory, and rendered to a screen using

这些矢量指令也存在内存中 \N 通过矢量图形卡画到屏幕上

a vector graphics card.

这些矢量指令也存在内存中 \N 通过矢量图形卡画到屏幕上

Hundreds of commands could be packed together, sequentially, in the screen buffer,

数百个命令可以按序存在屏幕缓冲区

and used to build up complex graphics. All made of lines!

画出复杂图形，全是线段组成的！

Because all these vectors are stored in memory, computer programs can update the values freely,

由于这些矢量都在内存中 \N 程序可以更新这些值

allowing for graphics that change over time -- Animation!

让图形随时间变化 - 动画！

One of the very earliest video games, Spacewar!,

最早的电子游戏之一, Spacewar!

was built on a PDP-1 in 1962 using vector graphics.

是 1962 年在 PDP-1 上用矢量图形制作的.

It's credited with inspiring many later games, like Asteroids,

它启发了许多后来的游戏，比如 爆破彗星(Asteroids)

and even the first commercial arcade video game: Computer Space.

甚至第一个商业街机游戏：太空大战

1962 was also a huge milestone because of Sketchpad,

1962 年是一个大里程碑 \N Sketchpad 诞生

an interactive graphical interface

一个交互式图形界面，用途是计算机辅助设计 (CAD)

that offered Computer-Aided Design -- called CAD Software today.

一个交互式图形界面，用途是计算机辅助设计 (CAD)

It's widely considered the earliest example of a complete graphical application.

它被广泛认为是第一个完整的图形程序

And its inventor, Ivan Sutherland, later won the Turing Award for this breakthrough.

发明人 伊万·萨瑟兰 后来因此获得图灵奖

To interact with graphics,

为了与图形界面交互 \N Sketchpad 用了当时发明不久的输入设备 光笔

Sketchpad used a recently invented input device called a light pen,

为了与图形界面交互 \N Sketchpad 用了当时发明不久的输入设备 光笔

which was a stylus tethered to a computer with a wire.

就是一个有线连着电脑的触控笔

By using a light sensor in the tip, the pen detected the refresh of the computer monitor.

笔尖用光线传感器，可以检测到显示器刷新

Using the timing of the refresh,

通过判断刷新时间，电脑可以知道笔的位置

the computer could actually figure out the pen's position on the screen!

通过判断刷新时间，电脑可以知道笔的位置

With this light pen, and various buttons on a gigantic computer,

有了光笔和各种按钮 \N 用户可以画线和其他简单形状

users could draw lines and other simple shapes.

有了光笔和各种按钮 \N 用户可以画线和其他简单形状

Sketchpad could do things like make lines perfectly parallel, the same length, straighten

Sketchpad 可以让线条完美平行，长度相同 \N 完美垂直90度，甚至动态缩放

corners into perfect 90 degree intersections, and even scale shapes up and down dynamically.

Sketchpad 可以让线条完美平行，长度相同 \N 完美垂直90度，甚至动态缩放

These things that were laborious on paper, a computer now did with a press of a button!

这些在纸上很费力，在计算机上非常简单！

Users were also able to save complex designs they created,

用户还可以保存设计结果，方便以后再次使用 \N 甚至和其他人分享

and then paste them into later designs, and even share with other people.

用户还可以保存设计结果，方便以后再次使用 \N 甚至和其他人分享

You could have whole libraries of shapes, like electronic components and pieces of furniture

你可以有一整个库 \N 里面有电子元件和家具之类的

that you could just plop in and manipulate in your creations.

可以直接拖进来用

This might all sound pretty routine from today's perspective.

从如今的角度来看 好像很普通

But in 1962, when computers were still cabinet-sized behemoths chugging through punch cards,

但在1962年 \N 计算机还是吃纸带的大怪兽，有柜子般大小

Sketchpad and light pens were equal parts eye opening and brain melting.

Sketchpad 和光笔让人大开眼界

They represented a key turning point in how computers could be used.

它们代表了人机交互方式的关键转折点

They were no longer just number crunching math machines that hummed along behind closed doors.

电脑不再是关在门后 负责算数的机器了

Now, they were potential assistants, interactively augmenting human tasks.

可以当助手 帮人类做事

The earliest computers and displays with true pixel graphics emerged in the late 1960s.

最早用真正像素的计算机和显示器\N 出现于 1960 年代末

Bits in memory directly "mapped" to pixels on the screen,

内存中的位(Bit) 对应屏幕上的像素

what are called bitmapped displays.

这叫 位图显示

With full pixel control, totally arbitrary graphics were possible.

现在我们可以绘制任意图形了

You can think of a screen's graphics as a huge matrix of pixel values .

你可以把图形想成一个巨大像素值矩阵

As before,

就像之前

computers reserve a special region of memory for pixel data, called the frame buffer.

计算机把像素数据存在内存中一个特殊区域 \N 叫"帧缓冲区"

In the early days, the computer's RAM was used,

早期时，这些数据存在内存里\N 后来存在高速视频内存里，简称 VRAM

but later systems used special high speed Video RAM, or VRAM,

早期时，这些数据存在内存里\N 后来存在高速视频内存里，简称 VRAM

which was located on the graphics card itself for high speed access.

VRAM 在显卡上，这样访问更快 \N 如今就是这样做的.

This is how it's done today.

VRAM 在显卡上，这样访问更快 \N 如今就是这样做的.

On an 8-bit grayscale screen, we can set values from 0 intensity, which is black,

在 8 位灰度屏幕上，我们可用的颜色范围是 0 强度（黑色）

to 255 intensity, which is white.

到 255 强度(白色)

Well actually, it might be green... or orange, as many early displays couldn't do white.

其实更像绿色或橙色 \N 因为许多早期显示器不能显示白色

Let's pretend this video is a really low resolution bitmapped screen,

我们假设 这个视频在低分辨率的位图屏幕上

with a resolution of 60 by 35 pixels.

分辨率 60x35像素

If we wanted to set the pixel at 10 10 to be white,

如果我们想把 (10,10) 的像素设为白色 \N 可以用这样的代码

we could do it with a piece of code like this.

如果我们想把 (10,10) 的像素设为白色 \N 可以用这样的代码

If we wanted to draw a line, let's say from 30, 0 to 30, 35, we can use a loop, like so.

如果想画一条线 假设从(30,0)到(30,35) \N 可以用这样一个循环

.And this changes a whole line of pixels to white.

把整列像素变成白色

If we want to draw something more complicated, let's say a rectangle,

如果想画更复杂的图形，比如矩形，那么需要四个值

we need to know four values.

如果想画更复杂的图形，比如矩形，那么需要四个值

The X and Y coordinate of its starting corner, and its width and height.

1. 起始点X坐标 \N2. 起始点Y坐标 \N 3. 宽度 \N 4. 高度

So far, we've drawn everything in white, so let's specify this rectangle to be grey.

目前只试了白色，这次画矩形试下灰色

Grey is halfway between 0 and 255, so that's a color value of 127.

灰色介于0到255中间 \N 所以我们用 127 (255/2=127.5)

Then, with two loops C one nested in the other,

然后用两个循环，一个套另一个

so that the inner loop runs once for every iteration of the outer loop,

这样外部每跑一次，内部会循环多次 \N 可以画一个矩形

we can draw a rectangle.

这样外部每跑一次，内部会循环多次 \N 可以画一个矩形

When the computer executes our code as part of its draw routine, it colors in all the

计算机绘图时会用指定的颜色 127

pixels we specified.

计算机绘图时会用指定的颜色 127

Let's wrap this up into a "draw rectangle function", like this:

我们来包装成 "画矩形函数"，就像这样：

Now, to draw a second rectangle on the other side of the screen, maybe in black this time,

假设要在屏幕的另一边 画第二个矩形 \N 这次可能是黑色矩形

we can just call our rectangle drawing function. Voila!!

可以直接调用 "画矩形函数", 超棒！

Just like the other graphics schemes we've discussed,

就像之前说的其他方案

programs can manipulate pixel data in the frame buffer, creating interactive graphics.

程序可以操纵"帧缓冲区"中的像素数据，实现交互式图形

Pong time!

乒乓球时间！

Of course, programmers aren't wasting time writing drawing functions from scratch.

当然，程序员不会浪费时间从零写绘图函数 \N 而是用预先写好的函数来做，画直线，曲线，图形，文字等

They use graphics libraries with ready-to-go functions

当然，程序员不会浪费时间从零写绘图函数 \N 而是用预先写好的函数来做，画直线，曲线，图形，文字等

for drawing lines, curves, shapes, text, and other cool stuff.

当然，程序员不会浪费时间从零写绘图函数 \N 而是用预先写好的函数来做，画直线，曲线，图形，文字等

Just a new level of abstraction!

一层新抽象！

The flexibility of bitmapped graphics opened up a whole new world of possibilities for

位图的灵活性，为交互式开启了全新可能 \N 但它的高昂成本持续了十几年

interactive computing, but it remained expensive for decades.

位图的灵活性，为交互式开启了全新可能 \N 但它的高昂成本持续了十几年

As I mentioned last episode, by as late as 1971,

上集提到，1971 年 \N 整个美国也只有大约 7 万个电传打字机和 7 万个终端

it was estimated there were around 70,000 electro-mechanical teletype machines

上集提到，1971 年 \N 整个美国也只有大约 7 万个电传打字机和 7 万个终端

and 70,000 terminals in use, in the United States.

上集提到，1971 年 \N 整个美国也只有大约 7 万个电传打字机和 7 万个终端

Amazingly, there were only around 1,000 computers in the US that had interactive graphical screens.

令人惊讶的是 \N 只有大约 1000 台电脑有交互式图形屏幕

That's not a lot!

这可不多！

But the stage was set helped along by pioneering efforts like Sketchpad and Space Wars

Sketchpad 和 太空大战 这样的先驱 \N 推动了图形界面发展

for computer displays to become ubiquitous,

帮助普及了计算机显示器 \N 由此，图形界面的曙光初现

and with them, the dawn of graphical user interfaces,

帮助普及了计算机显示器 \N 由此，图形界面的曙光初现

which we'll cover in a few episodes!

接下来讲图形界面

I'll see you next week.

下周见

Hi, I'm Carrie Anne and welcome to Crash Course Computer Science.

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne \N 欢迎收看计算机科学速成课！

Early in this series we covered computing history

之前介绍了计算机历史 \N 从人类文明的曙光开始 (第1集）

From roughly the dawn of civilization,

之前介绍了计算机历史 \N 从人类文明的曙光开始 (第1集）

up to the birth of electronic general purpose computers in the mid 1940s.

一直到 1940 年代中期 电子计算机诞生

A lot of the material we've discussed over the past 23 episodes

过去 23 集里讲的很多东西

like programming languages and compilers

比如编程语言和编译器，算法和集成电路

Algorithms and integrated circuits

比如编程语言和编译器，算法和集成电路

Floppy disks and operating systems, telly types and screens

软盘和操作系统，电报机和屏幕

all emerged over roughly a 30-year period,

全都是1940~1970年代，大概这30年间里出现的

From the mid 1940s up to the mid 1970s

全都是1940~1970年代，大概这30年间里出现的

This is the era of computing before companies like Apple and Microsoft existed

那时苹果和微软还不存在，也没有推特，谷歌或者 Uber.

and long before anyone Tweeted, Googled or Uber-d.

那时苹果和微软还不存在，也没有推特，谷歌或者 Uber.

It was a formative period setting the stage for personal computers,

还没到个人电脑时代

worldwide web, self-driving cars, virtual reality, and many other topics

而万维网，无人驾驶汽车，虚拟现实等主题，\N 这个系列的后半部分会讲

we'll get to in the second half of this series.

而万维网，无人驾驶汽车，虚拟现实等主题，\N 这个系列的后半部分会讲

Today we're going to step back from circuits and algorithms

今天, 我们不管电路和算法 \N 来聊聊这个影响力巨大的时代

and review this influential period.

今天, 我们不管电路和算法 \N 来聊聊这个影响力巨大的时代

We'll pay special attention to the historical backdrop of the cold war,

我们会把重点放在 \N 冷战，太空竞赛，全球化，消费主义的兴起.

The space race and the rise of globalization and consumerism.

我们会把重点放在 \N 冷战，太空竞赛，全球化，消费主义的兴起.

Pretty much immediately after World War II concluded in 1945,

1945年二战结束后不久

there was tension between the world's two new superpowers

两个超级大国的关系越发紧张，美国和苏联开始了冷战

the United States and the USSR

两个超级大国的关系越发紧张，美国和苏联开始了冷战

The Cold War had begun and with it,

因此政府往科学和工程学 投入大量资金

massive government spending on science and engineering.

因此政府往科学和工程学 投入大量资金

Computing which had already demonstrated its value

计算机在战时已经证明了自身的价值 \N 比如曼哈顿计划 和 破解纳粹通讯加密

in wartime efforts like the Manhattan Project

计算机在战时已经证明了自身的价值 \N 比如曼哈顿计划 和 破解纳粹通讯加密

and code breaking Nazi communications,

计算机在战时已经证明了自身的价值 \N 比如曼哈顿计划 和 破解纳粹通讯加密

was lavished with government funding.

所以政府大量投入资源 \N 各种雄心勃勃的项目得以进行

They enabled huge ambitious computing projects to be undertaken,

所以政府大量投入资源 \N 各种雄心勃勃的项目得以进行

like ENIAC, EDVAC, Atlas and Whirlwind all mentioned in previous episodes.

比如之前提过的 ENIAC, EDVAC, Atlas, Whirlwind

This spurred rapid advances that simply weren't possible in the commercial sector alone,

这种高速发展，如果仅靠商业运作是根本无法做到的

where projects were generally expected to recoup development costs through sales.

要依靠销售收回开发成本.

This began to change in the early 1950s,

1950年代，事情开始发生变化,

especially with Eckert and Buckley's Univac 1,

特别是 Univac 1，它是第一台取得商业成功的电脑

the first commercially successful computer.

特别是 Univac 1，它是第一台取得商业成功的电脑

Unlike ENIAC or Atlas , this wasn't just one single computer. It was a model of computers.

不像 ENIAC 或 Atlas \N Univanc 1 不是一台机器，而是一个型号

in total more than 40 were built.

一共造了40多台

Most of these Univacs went to government offices or large companies.

大部分 Univac 去了政府或大公司

Which was part of the growing military industrial complex in the United States,

成为美国日益增长的军事工业综合体的一部分

with pockets deep enough to afford the cutting edge.

因为政府有钱承担这些尖端科技.

Famously, a Univac 1 built for the U.S atomic energy commission

一个著名的例子是，\N 一台给 美国原子能委员会 生产的 Univac 1

was used by CBS to predict the results of the 1952 U.S. presidential election.

被 CBS 用来预测 1952 年美国总统大选的结果

With just 1% of the vote the computer correctly predicted

仅用1％的选票，Univac 1 正确预测了结果. \N 艾森豪威尔 获得压倒性胜利，而专家预测 史蒂文森 会赢

Eisenhower landslide, while pundits favored Stevenson.

仅用1％的选票，Univac 1 正确预测了结果. \N 艾森豪威尔 获得压倒性胜利，而专家预测 史蒂文森 会赢

It was a media event that helped propel computing to the forefront of the public's imagination

这次事件把计算机推到了公众面前

Computing was unlike machines of the past,

计算机和以前的机器不一样

which generally augmented human physical abilities.

以前的机器 增强人类的物理能力

Trucks allowed us to carry more, automatic looms whoa faster,

比如卡车能带更多东西，自动织布机更快

Machine tools were more precise and so on

机床更精确 等等. 这些东西代表了工业革命.

for a bunch of contraptions that typify the industrial revolution.

机床更精确 等等. 这些东西代表了工业革命.

But computers on the other hand could augment human intellect.

而计算机增强的是人类智力

This potential wasn't lost on Vannevar Bush,

范内瓦·布什 看到了这种潜力 \N 他在1945年发表了一篇文章

who in 1945 published an article on a

范内瓦·布什 看到了这种潜力 \N 他在1945年发表了一篇文章

hypothetical computing device he envisioned called the Memex.

描述了一种假想计算设备叫 Memex

This was a device in which an individual stores all his books,

可以用这个设备 \N 存自己所有的书, 其他资料 以及和别人沟通

records and communications and which is mechanized,

而且数据是按照格式存储, \N 所以可以快速查询，有很大灵活性.

so it may be consulted with exceeding speed and flexibility

而且数据是按照格式存储, \N 所以可以快速查询，有很大灵活性.

It is an enlarged intimate supplement to his memory.

可以辅助我们的记忆

He also predicted that wholly new forms of encyclopedia will appear,

他还预测会出现 新的百科全书形式

ready-made, with a mesh of associative trails running through them.

信息之间相互链接

Sound familiar?

听起来是不是很熟悉？（维基百科）

Memex directly inspired several subsequent game-changing systems,

Memex 启发了 之后几个重要里程碑

like Ivan Sutherland Sketchpad, which we discussed last episode,

比如上集 伊万·萨瑟兰 的 Sketchpad(画板)

and Dough Engelbart's online system, which we will cover soon.

以及后面很快会讲到 \N Dough Engelbart 的 oN-LINE 系统（第26集）

Vannevar Bush was the head of the U.S. office of Scientific Research and Development,

范内瓦·布什 \N 做过"美国科学研究与开发办公室"的头头

which was responsible for funding and coordinating scientific research during World War 2.

这个部门负责在二战期间 \N 资助和安排科学研究

With the Cold War brewing, Bush lobbied for a creation of a peace time equivalent,

冷战时 范内瓦·布什 到处游说，\N 想建立一个职责类似，但是在和平时期运作的部门

the National Science Foundation, formed in 1950.

因此 国家科学基金会 于1950年成立

To this day the NSF provides federal funding to support scientific research in the United States.

至今，国家科学基金会 \N 依然负责给科学研究 提供政府资金

And it is a major reason the U.S. has continued to be a leader in the technology sector.

美国的科技领先全球，主要原因之一就是这个机构.

It was also in the 1950s that consumers started to buy transistor powered gadgets,

1950年代，消费者开始买晶体管设备

notable among them was the transistor radio,

其中值得注意的是 收音机

which was small, durable and battery-powered.

它又小又耐用，用电池就够了，而且便携

And it was portable,

它又小又耐用，用电池就够了，而且便携

unlike the vacuum tube based radio sets from the 1940s and before.

不像 1940 年代之前的收音机，用的是真空管.

It was a runaway success, the Furby or iPhone of its day.

收音机非常成功，卖的像"菲比精灵"和 iPhone 一样畅销.

The Japanese government looking for industrial opportunities,

日本政府也在寻求工业机会，想振兴战后经济，

to bolster their post-war economy, soon got in on the action.

日本政府也在寻求工业机会，想振兴战后经济，

Licensing the rights to Transistors from Bell Labs in 1952.

他们很快动手从贝尔实验室 取得晶体管的授权

Helping launch the Japanese semiconductor and electronics industry.

帮助振兴日本的半导体和电子行业

In 1955, the first Sony product was released:

1955年，索尼的第一款产品面世

The TR-55 Transistor Radio. Concentrating on quality and price,

TR-55 晶体管收音机. 他们把重心放在质量和价格.

Japanese companies captured half of the U.S. Market for portable radios in just five years.

因此日本公司在短短5年内，就占有了美国便携式收音机市场的一半.

This planted the first seeds of a major industrial rivalry in the decades to come.

这为日本成为美国的强大工业对手，埋下伏笔

In 1953, there were only around 100 computers on the entire planet

1953年，整个地球大概有100台计算机

and at this point, the USSR was only a few years behind the West in computing technology,

苏联这时的计算机科技 只比西方落后几年

completing their first programmable electronic computer in 1950.

苏联在1950年 \N完成了第一个可编程电子计算机

But the Soviets were way ahead in the burgeoning space race.

但苏联在太空竞赛远远领先

Let's go to the thought-bubble.

我们进入思想泡泡

The Soviets launched the world's first satellite into orbit, Sputnik one,

苏联在1957年 \N 把第一个卫星送上轨道，史波尼克1号

in 1957, and a few years later in 1961.

不久，在1961年

Soviet Cosmonaut, Yuri Gagarin became the first human in space.

苏联宇航员 尤里·加加林 第一个进入太空

This didn't sit well with the American public

美国民众对此不满

and prompted President Kennedy, a month after Gagarin's mission,

使得肯尼迪总统 \N 在加加林太空任务一个月后

to encourage the nation to land a man on the moon within the decade. And it was expensive!

提出要登陆月球. 登月很贵的!

NASA's budget grew almost tenfold,

NASA 的预算增长了几乎十倍,

peaking in 1966 at roughly 4.5 percent of the U.S. Federal budget

在 1966 年达到顶峰，占了政府预算的4.5%

Today, It's around half a percent

如今, NASA 的预算只占 0.5%

NASA used this funding to tackle a huge array of enormous challenges

NASA 用这笔钱资助各种科学研究

this culminated in the Apollo program

阿波罗计划花的钱最多，雇了40万人左右

Which is peak employed roughly 400,000 people

阿波罗计划花的钱最多，雇了40万人左右

further supported by over 20,000 universities and companies

而且有2万多家大学和公司参与.

one of these huge challenges was navigating in space

其中一个挑战是 怎样在太空中导航

NASA needed a computer to process complex trajectories

NASA 需要电脑计算复杂的轨道来引导太空船

and issue guidance commands to the spacecraft

NASA 需要电脑计算复杂的轨道来引导太空船

For this, they built the Apollo guidance computer,

因此，他们造了 "阿波罗导航计算机"

There were three significant requirements

有3个重要要求

First, the computer had to be fast, no surprise there.

1. 计算机要快, 这在意料之中.

Second, it had to be small & lightweight

2. 计算机要又小又轻.

there's not a lot of room in a spacecraft

太空船里的空间不多

and every ounce is precious when you're flying a quarter million miles to the moon

而且要飞去月球，能轻一点是一点

And finally it had to be really really ridiculously reliable

3. 要超级可靠

This is super important in a spacecraft

这对太空船非常重要，因为太空中有很多震动，辐射，极端温度变化

Where there's lots of vibration radiation and temperature change

这对太空船非常重要，因为太空中有很多震动，辐射，极端温度变化

And there's no running to Best Buy, if something breaks.

如果东西坏掉了，可没办法去"百思买"买新的

the technology of the era of vacuum tubes and discrete transistors

那时的主流科技 \N 真空管和晶体管 无法胜任这些要求

Just weren't up to the task

那个时代的主流科技，真空管和离散晶体管，无法胜任这些要求.

so NASA turned to a brand-new technology, integrated circuits.

所以 NASA 用全新科技：集成电路

Which we discussed a few episodes ago

我们几集前聊过

The Apollo guidance computer was the first computer to use them, a huge paradigm shift

阿波罗导航计算机 首先使用了集成电路

NASA was also the only place that could afford them

NASA 是唯一负担得起集成电路的组织

Initially each chip cost around $50

最初，一个芯片差不多50美金

And the guidance computer needed thousands of them.

导航计算机需要上千个芯片

But by paying that price, the Americans were able to beat the soviets to the moon

但美国也因此成功登月，打败苏联

Thanks, thought-bubble

谢了 思想泡泡

Although the Apollo Guidance computer is credited

虽然人们经常把集成电路的发展 \N 归功于阿波罗导航计算机

with spurring the development and adoption of integrated circuits

虽然人们经常把集成电路的发展 \N 归功于阿波罗导航计算机

It was a low volume, product there are only 17 Apollo missions after all.

但它们的产量很低，一共只有 17 次阿波罗任务

it was actually military applications

实际上是军事 大大推进了集成电路发展

Especially the minuteman and polaris nuclear missile systems

特别是洲际导弹和核弹，使集成电路大规模生产

That allowed integrated circuits to become a mass-produced item

特别是洲际导弹和核弹，使集成电路大规模生产

This rapid Advancement was further accelerated by the U.S.

美国建造强大计算机时，也进一步推进了集成电路

Building and buying huge powerful computers

美国建造强大计算机时，也进一步推进了集成电路

Often called supercomputers, because they were frequently

一般叫"超级计算机"，\N 因为它们经常比全球最快电脑还快10倍以上

10 times faster than any other computer on the planet, upon their release.

一般叫"超级计算机"，\N 因为它们经常比全球最快电脑还快10倍以上

but these machines built by companies like CDC, Cray and IBM were also

但 CDC，Cray，IBM 制造的计算机非常昂贵

super in cost, and pretty much only governments could afford to buy them.

几乎只有政府负担得起

in the U.S. these machines went to government Agencies like the NSA.

这些计算机用于政府机构，比如美国国家安全局

and government research labs like Lawrence Livermore and Los Alamos National laboratories

以及实验室比如 \N 劳伦斯·利弗莫尔 实验室 \N 洛斯·阿拉莫斯 国家实验室

Initially the U.S. semiconductor industry boomed

最初，美国的半导体行业 \N 靠高利润政府合同起步

buoyed by High profit government contracts

最初，美国的半导体行业 \N 靠高利润政府合同起步

However this meant that most U.S.companies overlooked

因此忽略了消费者市场，因为利润小

the consumer market where profit margins were small

因此忽略了消费者市场，因为利润小

the Japanese Semiconductor industry came to dominate this niche

因此日本半导体行业在1950和1960年代 \N 靠低利润率占领了消费者市场

by having to operate with lean profit margins in the 1950s and 60s

因此日本半导体行业在1950和1960年代 \N 靠低利润率占领了消费者市场

the Japanese had invested heavily in manufacturing capacity

日本人投入大量资金，大量制造以达到规模经济

to achieve economies of scale

日本人投入大量资金，大量制造以达到规模经济

in research to improve quality and Yields and in automation to keep manufacturing costs low.

同时研究技术，提高质量和产量 \N 以及用自动化来降低成本

in the 1970s with the Space Race and cold war subsiding

1970年代，太空竞赛和冷战逐渐消退 \N 高利润的政府合同变少

previously juicy defense contracts began to dry up.

1970年代，太空竞赛和冷战逐渐消退 \N 高利润的政府合同变少

and American semi-conductor and electronics companies found it harder to compete.

美国的半导体和电子设备公司发现更难竞争了

it didn't help the many computing components had been commoditized

虽然很多计算机组件商品化了，但并没有什么帮助

DRAM was DRAM

DRAM 就是 DRAM

So why buy expensive Intel memory when you could buy the same chip for less from Hitachi?

能从日立买便宜的，干嘛要从英特尔买贵的？

Throughout the 1970s U.S. companies began to downsize,

1970年代 美国公司开始缩小，合并，或直接倒闭

consolidate or outright fail.

1970年代 美国公司开始缩小，合并，或直接倒闭

Intel had to lay off a third of its workforce in 1974

1974年 英特尔不得不裁员三分之一

and even the storied Fairchild semiconductor

知名的仙童半导体也在 1979 年濒临倒闭 \N 被其他公司收购了

was acquired in 1979 after near bankruptcy

知名的仙童半导体也在 1979 年濒临倒闭 \N 被其他公司收购了

to survive many of these companies began to outsource their manufacturing in a bid to reduce costs.

为了生存，很多公司把生产外包出去，降低成本

Intel withdrew from its main product category, Memory IC

英特尔不再把精力放在 内存集成电路 \N 而是把精力放在处理器

and decided to refocus on processes.

英特尔不再把精力放在 内存集成电路 \N 而是把精力放在处理器

Which ultimately saved the company.

这个决定最后挽救了公司

This low and U.S.

美国公司的无力 \N 导致 夏普 和 卡西欧 这样的日本公司

electronics industry allowed Japanese companies like Sharp and Casio

美国公司的无力 \N 导致 夏普 和 卡西欧 这样的日本公司

to dominate the breakout computing product of the 1970s.

占领了1970年代的主流产品

Handheld electronic calculators.

手持计算器

by using integrated circuits, these could be made small and cheap.

因为集成电路，计算机又小又便宜

They replaced expensive desktop adding machines you find in offices.

取代了办公室里昂贵的桌面计算器

For most people it was the first time they didn't have to do math on paper, or use a slide rule

对大多数人 \N 这是他们第一次不必用纸笔和计算尺来做计算

They were an instant hit, selling by the millions.

手持计算机因此大卖

This further drove down the cost of integrated circuits

进一步降低了集成电路的成本

and led to the development and widespread use of micro processors.

使得微处理器被广泛使用

like the Intel 4004 we've discussed previously

比如之前讨论过的 Intel 4004

This chip was built by Intel in 1971

Intel 在1971年 \N 应日本计算器公司 Busicom 的要求做了这个芯片

at the request of Japanese calculator company Busicom.

Intel 在1971年 \N 应日本计算器公司 Busicom 的要求做了这个芯片

Soon, Japanese electronics were everywhere.

很快，日本电子产品到处都是

from televisions of VCRs to digital wristwatches and Walkmans

从电视到手表到随身听

the availability of inexpensive microprocessor

而廉价的微处理器，也催生了全新的产品，比如街机游戏

Spawned in entirely new products like video arcades,

而廉价的微处理器，也催生了全新的产品，比如街机游戏

the world got Pong in 1972 and Breakout in 1976.

1972年诞生了Pong，1976年诞生了打砖块

as cost continued to plummet

因为成本不断下降

soon it became possible for regular people to afford computing devices

很快，普通人也买得起计算机了

during this time we see the emergence of the first successful home computers

这段期间，第一批家用电脑开始出现，比如1975年的 Altair 8800

like the 1975 Altair 8800

这段期间，第一批家用电脑开始出现，比如1975年的 Altair 8800

and also the first home gaming consoles

以及第一款家用游戏机，比如1977年的Atari 2600

like the Atari 2600 in 1977,

以及第一款家用游戏机，比如1977年的Atari 2600

Home, now I repeat that, Home.

家用！我再说一遍 家用！

That seems like a small thing today.

如今没什么大不了的.

But this was the dawn of a whole new era in computing.

但那时是计算机的全新时代

in just three decades, computers have evolved from

在短短三十年内，计算机从大到人类可以在 CPU 里走来走去

machines where you could literally walk inside of the CPU.

在短短三十年内，计算机从大到人类可以在 CPU 里走来走去

assuming you had government clearance

当然，你要有政府许可你这样做.

to the point where a child could play with a handheld toy

发展到小到小孩都能拿住的手持玩具，而且微处理器还快得多.

Containing a microprocessor many times faster,

发展到小到小孩都能拿住的手持玩具，而且微处理器还快得多.

Critically, this dramatic evolution would have been but without two powerful forces at play

这种巨大变化是由两种力量推动的：政府和消费者

Governments and Consumers.

这种巨大变化是由两种力量推动的：政府和消费者

Government funding like the United States provided during the cold war

政府资金，比如冷战期间美国投入的钱

enabled early adoption of many nascent computing technologies

推动了计算机的早期发展

This funding helped flow entire Industries relate into computing long enough

并且让计算机行业活得足够久，使得技术成熟到可以商用

for the technology to mature and become commercially feasible.

并且让计算机行业活得足够久，使得技术成熟到可以商用

Then businesses and ultimately consumers, provided the demand to take it mainstream.

然后是公司，最后是消费者，把计算机变成了主流

The cold war may be over, but this relationship continues today

冷战虽然结束了，但这种关系今天仍在继续

Governments are still funding science research.

政府依然在资助科学研究

intelligence agencies are still buying supercomputers.

情报机构依然在超级计算机

humans are still being launched into space.

人类仍然被发射到太空里

And you're still buying TV, Xbox, Playstation, Laptop and Smartphone

而你依然在买电视，Xbox，Playstation，笔记本电脑和手机

and for these reasons,

因此，计算机会继续飞速发展

Computing continues to advance a lightning pace.

因此，计算机会继续飞速发展

I'll see you next week

我们下周见

Hi, I'm Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne \N 欢迎收看计算机科学速成课！

As we discussed last week, the idea of having a computer all to yourself a personal

上周说过"个人计算机"的概念 \N 在计算机发展的头 30 年难以想象

computer was elusive for the first three decades of electronic computing.

上周说过"个人计算机"的概念 \N 在计算机发展的头 30 年难以想象

It was just way too expensive for a computer to be owned and used by one single person.

如果只让一个人用，成本实在太高

But, by the early 1970s, all the required components had fallen into place to build

但到 70 年代初，各种组件的成本都下降了 \N 可以做出低成本 同时性能足够强大的计算机

a low cost, but still usefully powerful computer.

但到 70 年代初，各种组件的成本都下降了 \N 可以做出低成本 同时性能足够强大的计算机

Not a toy, but a tool.

不是玩具级计算机，是真正能用的计算机

Most influential in this transition was the advent of single-chip CPUs,

这个转变中 \N 最有影响力的是 单芯片 CPU 的出现

which were surprisingly powerful, yet small and inexpensive.

强大 + 体积小 + 便宜

Advances in integrated circuits also offered low-cost solid-state memory,

集成电路的进步，也提供了低成本固态存储器

both for computer RAM and ROM.

可以用于计算机的 RAM 和 ROM

Suddenly it was possible to have an entire computer on one circuit board,

忽然间 \N 把整台计算机做到一张电路板上成为可能

dramatically reducing manufacturing costs.

大大地降低了制造成本

Additionally, there was cheap and reliable computer storage,

而且，那时有便宜可靠的储存介质 \N 比如磁带和软盘

like magnetic tape cassettes and floppy disks.

而且，那时有便宜可靠的储存介质 \N 比如磁带和软盘

And finally, the last ingredient was low cost displays, often just repurposed televisions.

最后是 低成本的显示器 \N 通常是电视机稍作改装而成

If you blended these four ingredients together in the 1970s, you got,

如果在 1970 年代 \N 将这四种原料混在一起

what was called a microcomputer,

就得到了"微型计算机"

because these things were so tiny compared to "normal" computers of that era, the

因为和那个时代的"普通"计算机相比 \N 这些计算机很小

types you'd see in business or universities.

"普通"计算机就是公司或大学里的那种

But more important than their size was their cost.

但比大小更重要的是成本

These were, for the first time, sufficiently cheap.

这是有史以来第一次，计算机的价格足够低

It was practical to buy one and only have one person ever use it.

"一个人专用"的想法变得可行

No time sharing,

不用划分时间和别人公用计算机

no multi-user logins, just a single owner and user.

没有多用户登录，计算机只属于一个人，只有一个用户

The personal computer era had arrived.

个人计算机时代到来

Computer cost and performance eventually reached the point

计算机成本下降+性能提升, 让个人计算机成为可能

where personal computing became viable.

计算机成本下降+性能提升, 让个人计算机成为可能

But, it's hard to define exactly when that happened.

但这个时间点很难准确定义, 并没有一个具体时间点

There's no one point in time.

但这个时间点很难准确定义, 并没有一个具体时间点

And as such, there are many contenders for the title of "first" personal computer,

因此"第一台个人计算机"这个名号，有很多竞争者

like the Kenback-1 and MCM/70.

比如 Kenback-1 和 MCM/70

Less disputed, however, is the first commercially successful personal computer: The Altair 8800.

不过第一台取得商业成功的个人计算机 \N 争议较小：Altair 8800

This machine debuted on the cover of Popular Electronics in 1975,

首次亮相在 1975 年《Popular Electronics》封面

and was sold as a $439 kit that you built yourself.

售价 $439 美元，需要自己组装

Inflation adjusted, that's about $2,000 today,

计算通货膨胀后，相当如今的 2000 美元左右

which isn't chump change, but extremely cheap for a computer in 1975.

不算小钱，但比起 1975 年的其它计算机，算是非常便宜了

Tens of thousands of kits were sold to computer hobbyists,

各种需要自己组装的组件包 \N卖给了计算机爱好者

and because of its popularity, there were soon all sorts of nifty add-ons available...

因为买的人多，很快相关产品出现了

things like extra memory, a paper tape reader and even a teletype interface.

比如内存，纸带读取器，甚至电传接口

This allowed you, for example, to load a longer, more complicated program from punch tape,

让你可以从纸带上读取更长更复杂的程序

and then interact with it using a teletype terminal.

然后用电传终端交互

However, these programs still had to be written in machine code,

但程序还是要用 机器码 写

which was really low level and nasty, even for hardcore computer enthusiasts.

写起来很麻烦，即使计算机爱好者也讨厌写

This problem didn't escape a young Bill Gates and Paul Allen,

这没有吓跑年轻的比尔·盖茨和保罗·艾伦

who were 19 and 22 respectively.

他们当时是19岁和22岁

They contacted MITS, the company making the Altair 8800,

他们联系了制造 Altair 8800 的 MITS 公司

suggesting the computer would be more attractive to hobbyists

建议说，如果能运行 BASIC 程序 \N 会对爱好者更有吸引力

if it could run programs written in BASIC,

建议说，如果能运行 BASIC 程序 \N 会对爱好者更有吸引力

a popular and simple programming language.

BASIC 是一门更受欢迎更简单的编程语言

To do this, they needed to write a program that converted BASIC instructions

为此，他们需要一个程序 \N 把 BASIC 代码转成可执行机器码

into native machine code, what's called an interpreter.

这叫 解释器 (interpreter)

This is very similar to a compiler,

"解释器"和"编译器"类似

but happens as the programs runs instead of beforehand.

区别是"解释器"运行时转换\N而"编译器"提前转换

Let's go to the thought bubble!

让我们进入思想泡泡！

MITS was interested,

MITS 表示感兴趣

and agreed to meet Bill and Paul for a demonstration.

同意与 Bill 和 Paul 见个面，让他们演示一下

Problem is, they hadn't written the interpreter yet.

问题是，他们还没写好解释器

So, they hacked it together in just a few weeks

所以他们花了几个星期赶工 \N 而且还不是在 Altair 8800 上写的

without even an Altair 8800 to develop on,

所以他们花了几个星期赶工 \N 而且还不是在 Altair 8800 上写的

finishing the final piece of code on the plane.

最后在飞机上完成了代码

The first time they knew their code worked was at MITS headquarters

他们在墨西哥 阿尔伯克基（城市） \N 的 MITS 总部做演示时，才知道代码可以成功运行

in Albuquerque, New Mexico, for the demo.

他们在墨西哥 阿尔伯克基（城市） \N 的 MITS 总部做演示时，才知道代码可以成功运行

Fortunately, it went well and MITS agreed to distribute their software.

幸运的是进展顺利 \N MITS 同意在计算机上搭载他们的软件

Altair BASIC became the newly formed Microsoft's first product.

Altair BASIC 成了微软的第一个产品

Although computer hobbyists existed prior to 1975,

虽然1975年之前就有计算机爱好者

the Altair 8800 really jump-started the movement.

但 Altair 8800 大量催生了更多计算机爱好者

Enthusiast groups formed, sharing knowledge and software and passion about computing.

爱好者们组成各种小组 \N 分享知识，软件，以及对计算机的热爱

Most legendary among these is the Homebrew Computer Club,

最具传奇色彩的小组是"家酿计算机俱乐部"

which met for the first time in March 1975

第一次小组聚会在1975年3月

to see a review unit of the Altair 8800, one of the first to ship to California.

看一台第一批运来加州的 Altair 8800

At that first meeting was 24-year-old Steve Wozniak, who was so inspired by

第一次聚会上，24岁的 Steve Wozniak \N 被 Altair 8800 大大激励

the Altair 8800 that he set out to design his own computer.

开始想设计自己的计算机

In May 1976, he demonstrated his prototype to the Club

1976年5月，他向小组展示了原型机

and shared the schematics with interested members.

并且把电路图分享给感兴趣的其他会员

Unusual for the time, it was designed to connect to a TV and offered a text interface

他的设计不同寻常 \N 要连到电视显示，并提供文本界面

a first for a low-cost computer.

在低成本计算机上还是第一次见

Interest was high, and shortly after fellow club member and college friend Steve Jobs

同是俱乐部成员和大学朋友的 史蒂夫·乔布斯

suggested that instead of just sharing the designs for free,

建议说与其免费分享设计，不如直接出售装好的主板

they should just sell an assembled motherboard.

建议说与其免费分享设计，不如直接出售装好的主板

However, you still had to add your own keyboard, power supply, and enclosure.

但用户依然需要自己加键盘，电源和机箱

It went on sale in July 1976 with a price tag of $666.66.

1976年7月开始发售，价格$666.66美元

It was called the Apple-I, and it was Apple Computer's first product.

它叫 Apple-I ，苹果计算机公司的第一个产品

Thanks thought bubble!

谢了 思想泡泡

Like the Altair 8800, the Apple-I was sold as a kit.

就像 Altair 8800 一样，Apple-I 也是作为套件出售

It appealed to hobbyists,

Apple-I 吸引了业余爱好者 \N 不介意机器买回来自己组装

who didn't mind tinkering and soldering,

Apple-I 吸引了业余爱好者 \N 不介意机器买回来自己组装

but consumers and businesses weren't interested.

但个人消费者和公司对 Apple-I 不感兴趣

This changed in 1977,

这在 1977 年发生变化 \N 市场上有了三款开箱即用的计算机

with the release of three game-changing computers, that could be used right out of the box.

这在 1977 年发生变化 \N 市场上有了三款开箱即用的计算机

First was the Apple II,

第一款是 Apple-II

Apple's earliest product that sold as a complete system

苹果公司第一个提供全套设备的产品

that was professionally designed and manufactured.

设计和制造工艺都是专业的

It also offered rudimentary color graphics and sound output,

它还提供了简单彩色图形和声音输出

amazing features for a low cost machine.

这些功能对低成本机器非常了不起

The Apple II series of computers sold by the millions and quickly

Apple-II 卖了上百万套

propelled Apple to the forefront of the personal computing industry.

把苹果公司推到了个人计算机行业的前沿

The second computer was the TRS-80 Model I,

第二款是"TRS-80 1型"

made by the Tandy Corporation

由 Tandy 公司生产

and sold by RadioShack hence the "TRS"

由 Radioshack 销售，所以叫 TRS

Although less advanced than the Apple II,

虽然不如 Apple-II 先进 \N 但因为价格只有一半，所以卖得很火爆

it was half the cost and sold like hot cakes.

虽然不如 Apple-II 先进 \N 但因为价格只有一半，所以卖得很火爆

Finally, there was the Commodore PET 2001,

最后一款是 Commodore PET 2001

with a unique all-in-one design

有一体化设计

that combined computer, monitor, keyboard and tape drive into one device,

集成了计算机，显示器，键盘和磁带驱动器

aimed to appeal to consumers.

目标是吸引普通消费者

It started to blur the line between computer and appliance.

计算机和家用电器之间的界限开始变得模糊

These three computers became known as the 1977 Trinity.

这3台计算机被称为1977年的"三位一体"

They all came bundled with BASIC interpreters,

它们都自带了 BASIC 解释器

allowing non-computer-wizards to create programs.

让不那么精通计算机的人也能用 BASIC 写程序

The consumer software industry also took off,

针对消费者的软件行业 开始腾飞

offering games and productivity tools for personal computers,

市场上出现了各种 \N 针对个人计算机的游戏和生产力工具

like calculators and word processors.

比如计算器和文字处理器

The killer app of the era was 1979's VisiCalc,

最火的是 1979 年的 VisiCalc

the first spreadsheet program

第一个电子表格程序

which was infinitely better than paper

比纸好无数倍

and the forbearer of programs like Microsoft Excel and Google Sheets.

是微软 Excel 和 Google Sheets 的老祖先

But perhaps the biggest legacy of these computers was their marketing,

但这些计算机带来的最大影响 \N 也许是他们的营销策略

they were the first to be targeted at households, and not just businesses and hobbyists.

它们针对普通消费者 \N 而不是企业和爱好者

And for the first time in a substantial way,

这是第一次大规模地

computers started to appear in homes, and also small businesses and schools.

计算机出现在家庭，小公司，以及学校中

This caught the attention of the biggest computer company on the planet, IBM, who had seen its

这引起了全球最大计算机公司 IBM 的注意

share of the overall computer market shrink from 60% in 1970 to around 30% by 1980.

其市场份额从1970年的60％ \N 在1980年降到了30％左右

This was mainly because IBM had ignored the microcomputer market,

因为IBM忽略了增长的"微型计算机"市场

which was growing at about 40% annually.

这个市场每年增长约40％

As microcomputers evolved into personal computers, IBM knew it needed to get in on the action.

随着微型计算机演变成个人计算机 \N IBM 知道他们需要采取行动

But to do this, it would have to radically rethink its computer strategy and design.

但要做到这一点 \N 公司要从根本上重新思考战略和设计

In 1980, IBM's least-expensive computer, the 5120, cost roughly ten thousand dollars,

1980年 IBM 最便宜的计算机 \N "5120"的价格大概是一万美元

which was never going to compete with the likes of the Apple II.

永远也没法和 Apple-II 这样的计算机竞争

This meant starting from scratch.

意味着要从头开始

A crack team of twelve engineers, later nicknamed the dirty dozen,

一个由十二名工程师组成的精干团队（后来叫"肮脏十二人"）

were sent off to offices in Boca Raton, Florida,

被派往佛罗里达州的 \N 博卡拉顿（Boca Raton）办公室

to be left alone and put their talents to work.

让他们独立工作

Shielded from IBM internal politics, they were able to design a machine as they desired.

不受 IBM 内部的政治斗争干扰 \N 他们想怎么设计怎么设计

Instead of using IBM proprietary CPUs, they chose Intel chips.

没用 IBM 的 CPU，选了 Intel 的芯片

Instead of using IBM's prefered operating system, CP/M,

也没用 IBM 的首选操作系统 CP/M

they licenced Microsoft's Disk Operating System: DOS

而是用了微软的 DOS

and so on, from the screen to the printer.

依此类推，从屏幕到打印机都这样自由选择

For the first time, IBM divisions had to compete with outside firms

IBM 第一次不得不与外部公司竞争

to build hardware and software for the new computer.

来给新计算机做硬件和软件

This radical break from the company tradition of in-house development kept costs low

这和 IBM 的传统做法不同：自己做硬件来节省成本

and brought partner firms into the fold.

然后和其它公司合作

After just a year of development,

经过短短一年

the IBM Personal Computer, or IBM PC was released.

IBM 个人计算机发布了，简称 IBM PC

It was an immediate success,

产品立马取得了成功

especially with businesses that had long trusted the IBM brand.

长期信任 IBM 品牌的企业买了很多

But, most influential to its ultimate success was that the computer featured an open architecture,

但最有影响力的是 \N 它使用 "开放式架构"

with good documentation and expansion slots,

有良好的文档和扩展槽

allowing third parties to create new hardware and peripherals for the platform.

使得第三方可以做硬件/外设

That included things like graphics cards, sounds cards, external hard drives, joysticks,

包括显卡，声卡，外置硬盘，游戏控制杆 \N 以及无数其它组件

and countless other add-ons.

包括显卡，声卡，外置硬盘，游戏控制杆 \N 以及无数其它组件

This spurred innovation, and also competition, resulting in a huge ecosystem of products.

这刺激了创新，激发了竞争，产生了巨大的生态系统

This open architecture became known as "IBM Compatible".

这个开放架构叫 IBM Compatible"（IBM 兼容）

If you bought an "IBM Compatible" computer, it meant you

意味着如果买了"IBM兼容"的计算机

could use that huge ecosystem of software and hardware.

你可以用庞大生态系统中的其它软硬件

Being an open architecture also meant that competitor companies could follow the standard

开放架构也意味着 竞争对手公司可以遵循这个标准

and create their own IBM Compatible computers.

做出自己的"IBM 兼容"计算机

Soon, Compaq and Dell were selling their own PC clones...

很快，康柏和戴尔也开始卖 PC

And Microsoft was happy to license MS-DOS to them,

微软很乐意把 MS-DOS 授权给他们

quickly making it the most popular PC operating system.

使 DOS 迅速成为最受欢迎的 PC 操作系统

IBM alone sold two million PCs in the first three years, overtaking Apple.

仅在前三年 \N IBM就卖出了200万台 PC ，超过了苹果

With a large user base, software and hardware developers concentrated

有了庞大用户群，软件和硬件开发人员 \N 把精力放在"IBM 兼容"平台，因为潜在用户更多

their efforts on IBM Compatible platforms, there were just more users to sell to.

有了庞大用户群，软件和硬件开发人员 \N 把精力放在"IBM 兼容"平台，因为潜在用户更多

Then, people wishing to buy a computer bought the one with the

同时，想买计算机的人 \N 也会看哪种计算机的软硬件选择更多

most software and hardware available, and this effect snowballed.

就像雪球效应一样

Companies producing non-IBM-compatible computers, often with superior specs,

而那些生产非"IBM兼容"计算机的公司 （一般性能更好）

failed.

都失败了

Only Apple kept significant market share without IBM compatibility.

只有苹果公司在没有"IBM兼容"的情况下 \N 保持了足够市场份额

Apple ultimately chose to take the opposite approach a "closed architecture" proprietary

苹果公司最终选了相反的方式："封闭架构"

designs that typically prevent people from adding new hardware to their computers.

即自己设计一切，用户一般无法加新硬件到计算机中

This meant that Apple made its own computers, with its own operating system, and often its

意味着苹果公司要做自己的计算机，自己的操作系统

own peripherals, like displays, keyboards, and printers.

还有自己的外围设备，如显示器，键盘和打印机

By controlling the full stack, from hardware to software,

通过控制整个范围，从硬件到软件

Apple was able to control the user experience and improve reliability.

苹果能控制用户体验并提高可靠性

These competing business strategies were the genesis of the "Mac" versus "PC" division

不同的商业策略是"Mac vs PC 谁更好"这种争论的起源

that still exists today... which is a misnomer, because they're both personal computers!

这些争论如今还存在 \N 不过"Mac vs PC"用词不对，因为它们都是个人计算机！

But whatever.

但是随便啦

To survive the onslaught of low-cost PCs,

为了在低成本个人计算机的竞争冲击下生存下来

Apple needed to up its game,

苹果需要提高自身水平 \N 提供比 PC 和 DOS 更好的用户体验

and offer a user experience that PCs and DOS couldn't.

苹果需要提高自身水平 \N 提供比 PC 和 DOS 更好的用户体验

Their answer was the Macintosh, released in 1984.

他们的答案是 Macintosh，于 1984 年发布

This ground breaking, reasonably-low-cost, all-in-one computer

一台突破性 价格适中的一体式计算机 \N 用的不是命令行界面，而是图形界面

booted not a command-line text-interface, but rather a graphical user interface,

一台突破性 价格适中的一体式计算机 \N 用的不是命令行界面，而是图形界面

our topic for next week. See you then.

我们下周讨论图形界面. 到时见

Hi, I'm Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨 我是 Carrie Anne 欢迎收看计算机科学速成课

We ended last episode with the 1984 release of Apple's Macintosh personal computer.

我们上集最后 \N 谈了苹果在1984年发布的 Macintosh

It was the first computer a regular person could buy with a graphical user interface

这是普通人可以买到的 \N 第一台带图形用户界面的计算机

and a mouse to interact with it.

还带一个鼠标

This was a radical evolution from the command line interfaces

那时的计算机全是命令行 \N 图形界面是个革命性进展

found on all other personal computers of the era.

那时的计算机全是命令行 \N 图形界面是个革命性进展

Instead of having to remember...

不必记住或猜正确的命令

or guess... the right commands to type in,

我们不必记住或猜测正确的命令

a graphical user interface shows you what functions are possible.

图形界面直接显示了，你可以做什么

You just have to look around the screen for what you want to do.

只要在屏幕上找选项就行了

It's a "point and click" interface.

这是一个"选择并点击"的界面

All of a sudden, computers were much more intuitive.

突然间计算机更直观了

Anybody, not just hobbyists or computer scientists,

不只是爱好者或科学家能用计算机 \N 任何人都可以用计算机解决问题

could figure things out all by themselves.

不只是爱好者或科学家能用计算机 \N 任何人都可以用计算机解决问题

The Macintosh is credited with taking Graphical User Interfaces, or GUIs, mainstream,

人们认为是 Macintosh \N 把图形用户界面（GUI）变成主流

but in reality they were the result of many decades of research.

但实际上图形界面是数十年研究的成果

In previous episodes, we discussed some early interactive graphical applications,

前几集，我们讨论了早期的交互式图形程序

like Sketchpad and Spacewar!, both made in 1962.

比如 Sketchpad 和太空战争 \N 都是1962年制作的

But these were one-off programs,

但都是一次性项目，不是整合良好的体验

and not whole integrated computing experiences.

但都是一次性项目，不是整合良好的体验

Arguably, the true forefather of modern GUIs was Douglas Engelbart.

现代图形界面的先驱 \N 可以说是 道格拉斯·恩格尔巴特

Let's go to the thought bubble!

让我们进入思想泡泡！

During World War 2, while Engelbart was stationed in the Philippines as a radar operator,

二战期间 \N 恩格尔巴特 驻扎在菲律宾做雷达操作员

he read Vannevar Bush's article on the Memex.

他读了 万尼瓦尔·布什 的 Memex 文章

These ideas inspired him,

这些文章启发了他

and when his Navy service ended,

当他海军服役结束时

he returned to school, completing a Ph.D. in 1955 at U.C. Berkeley.

他回到学校 \N 1955年在 UCB 取得博士学位

Heavily involved in the emerging computing scene,

他沉溺于新兴的计算机领域

he collected his thoughts in a seminal 1962 report,

他在1962年一份开创性报告中 \N 汇集了各种想法

titled: "Augmenting Human Intellect".

报告名为："增强人类智力"

Engelbart believed that the complexity of the problems facing mankind

恩格尔巴特认为，人类面临的问题 \N 比解决问题的能力增长得更快

was growing faster than our ability to solve them.

恩格尔巴特认为，人类面临的问题 \N 比解决问题的能力增长得更快

Therefore, finding ways to augment our intellect

因此，找到增强智力的方法 \N 似乎是必要且值得一做的目标

would seem to be both a necessary and a desirable goal.

因此，找到增强智力的方法 \N 似乎是必要且值得一做的目标

He saw that computers could be useful beyond just automation,

他构想计算机不仅做自动化工作

and be essential interactive tools for future knowledge workers to tackle complex problems.

也可以成为未来知识型员工 \N 应对复杂问题的工具

Further inspired by Ivan Sutherland's recently demonstrated Sketchpad,

伊凡·苏泽兰 的"几何画板" \N 进一步启发了 恩格尔巴特

Engelbart set out to make his vision a reality, recruiting a team to build the oN-Line System.

他决定动手把愿景变为现实 \N 开始招募团队来做 oN-Line System

He recognized that a keyboard alone was insufficient

他意识到如果只有键盘 \N 对他想搭建的程序来说是不够的

for the type of applications he was hoping to enable.

他意识到如果只有键盘 \N 对他想搭建的程序来说是不够的

In his words:

用他的话说：

"We envisioned problem-solvers using computer-aided working stations to augment their efforts.

"我们设想人们用计算机辅助工作站来增强工作

They required the ability to interact with information displays

用户需要和屏幕上的信息互动

using some sort of device to move [a cursor] around the screen."

用某种设备在屏幕上移动[光标]"

And in 1964, working with colleague Bill English,

1964年，和同事比尔·英格利希的共同努力下

he created the very first computer mouse.

他创造了第一个计算机鼠标，尾部有一根线

The wire came from the bottom of the device

他创造了第一个计算机鼠标，尾部有一根线

and looked very much like a rodent and the nickname stuck.

看起来很像老鼠 \N 因此"鼠标"这个名字沿用了下来

Thanks thought bubble!

谢了思想泡泡！

In 1968, Engelbart demonstrated his whole system at the Fall Joint Computer Conference,

1968年 恩格尔巴特 \N 在"秋季计算机联合会议"展示了他的系统

in what's often referred to as "the mother of all demos".

这次演示 被视为如今所有演示的祖先

The demo was 90 minutes long and demonstrated many features of modern computing:

演示有90分钟 \N 展现了现代计算机的许多功能：

bitmapped graphics,

包括 位图图像

video conferencing,

视频会议

word processing,

文字处理

and collaborative real-time editing of documents.

和 实时协作编辑文件

There were also precursors to modern GUIs,

还有现代图形界面的原型 比如鼠标和多窗口

like the mouse and multiple windows

还有现代图形界面的原型 比如鼠标和多窗口

- although they couldn't overlap.

- 不过窗口不能重叠

It was way ahead of its time,

远远先于那个时代

and like many products with that label, it ultimately failed,

就像其它"跨时代"的产品一样，它最终失败了

at least commercially.

至少商业上是这样

But its influence on computer researchers of the day was huge.

但它对当时的计算机研究者影响巨大

Engelbart was recognized for this watershed moment in computing with a Turing Award in 1997.

恩格尔巴特 因此获得1997年图灵奖

Federal funding started to reduce in the early 1970s,

政府资金在 1970 年代初开始减少

which we discussed two episodes ago.

我们在两集前说过\N（第24集：冷战和消费主义）

At that point, many of Engelbart's team, including Bill English,

那时，恩格尔巴特团队里的许多人，包括比尔·英格利希

left and went to Xerox's newly formed Palo Alto Research Center,

去了施乐公司新成立的"帕洛阿尔托研究中心"

more commonly known as Xerox PARC.

更为人熟知的名字是 Xerox PARC

It was here that the first true GUI computer was developed:

他们在这里开发了第一台带真正 GUI 的计算机：

the Xerox Alto, finished in 1973.

施乐奥托 于1973年完成

For the computer to be easy to use,

为了让计算机易于使用，需要的不只是花哨的图形

it needed more than just fancy graphics.

为了让计算机易于使用，需要的不只是花哨的图形

It needed to be built around a concept that people were already familiar with,

还要借助一些人们已经熟悉的概念

so they could immediately recognize how to use the interface with little or no training.

让人们不用培训 就能很快明白如何使用

Xerox's answer was to treat the 2D screen like the top of a desk or desktop.

施乐的答案是将2D屏幕当作"桌面"

Just like how you can have many papers laid out on a desk,

就像桌面上放很多文件一样

a user could have several computer programs open at once.

用户可以打开多个程序 \N 每个程序都在一个框里，叫"窗口"

Each was contained in their own frame,

用户可以打开多个程序 \N 每个程序都在一个框里，叫"窗口"

which offered a view onto the application called a window.

用户可以打开多个程序 \N 每个程序都在一个框里，叫"窗口"

Also like papers on a desk,

就像桌上的文件一样

these windows could overlap, blocking the items behind them.

窗口可以重叠，挡住后面的东西

And there were desk accessories, like a calculator and clock,

还有桌面配件，比如计算器和时钟

that the user could place on the screen and move around.

用户可以把配件在屏幕上四处移动

It wasn't an exact copy of a desktop though.

它不是现实桌面的完美复制，而是用桌面这种隐喻

Instead, it was a metaphor of a desktop.

它不是现实桌面的完美复制，而是用桌面这种隐喻

For this reason, surprisingly, it's called the Desktop Metaphor.

因此叫"桌面隐喻"

There are many ways to design an interface like this,

有很多方法来设计界面 \N 但 Alto 团队用窗口，图标，菜单和指针来做

but the Alto team did it with windows, icons, menus, and a pointer

有很多方法来设计界面 \N 但 Alto 团队用窗口，图标，菜单和指针来做

- what's called a WIMP interface.

- 因此叫 WIMP 界面

It's what most desktop GUIs use today.

如今大部分图形界面都用这个

It also offered a basic set of widgets,

它还提供了一套基本部件

reusable graphical building blocks, things like buttons, checkboxes, sliders, and tabs

可复用的基本元素 \N 比如按钮，打勾框，滑动条和标签页

which were also drawn from real world objects to make them familiar.

这些也来自现实世界，让人们有熟悉感

GUI applications are constructed from these widgets,

GUI 程序就是这些小组件组成的

so let's try coding a simple example using this new programming paradigm.

让我们试着写一个简单例子

First, we have to tell the operating system that we need a new window to be created for our app.

首先，我们必须告诉操作系统 \N 为程序创建一个窗口

We do this through a GUI API.

我们通过 GUI API 实现 \N 需要指定窗口的名字和大小

We need to specify the name of the window and also its size.

我们通过 GUI API 实现 \N 需要指定窗口的名字和大小

Let's say 500 by 500 pixels.

假设是 500×500 像素

Now, let's add some widgets - a text box and a button.

现在再加一些小组件，一个文本框和一个按钮

These require a few parameters to create.

创建它们需要一些参数

First, we need to specify what window they should appear in,

首先要指定出现在哪个窗口 \N 因为程序可以有多个窗口

because apps can have multiple windows.

首先要指定出现在哪个窗口 \N 因为程序可以有多个窗口

We also need to specify the default text, the X and Y location in the window, and a width and height.

还要指定默认文字\N 窗口中的 X,Y 位置 以及宽度和高度

Ok, so now we've got something that looks like a GUI app,

好，现在我们有个 \N 看起来像 GUI 程序的东西

but has no functionality.

但它还没有功能

If you click the "roll" button, nothing happens.

如果点 Roll 按钮，什么也不会发生

In previous examples we've discussed,

在之前的例子中，代码是从上到下执行的

the code pretty much executes from top to bottom.

在之前的例子中，代码是从上到下执行的

GUIs, on the other hand, use what's called event-driven programming;

但 GUI 是 "事件驱动编程"

code can fire at any time, and in different orders, in response to events.

代码可以在任意时间执行 以响应事件

In this case, it's user driven events,

这里是用户触发事件 \N 比如点击按钮，选一个菜单项，或滚动窗口

like clicking on a button, selecting a menu item, or scrolling a window.

这里是用户触发事件 \N 比如点击按钮，选一个菜单项，或滚动窗口

Or if a cat runs across your keyboard,

或一只猫踩过键盘

it's a bunch of events all at once!

就会一次触发好多事件！

Let's say that when the user clicks the "roll" button,

假设当用户点 Roll 按钮

we want to randomly generate a number between 1 and 20,

我们产生1到20之间的随机数

and then show that value in our text box.

然后在文本框中，显示这个数字

We can write a function that does just that.

我们可以写一个函数来做

We can even get a little fancy and say if we get the number 20,

我们还可以让它变有趣些，假设随机数是 20 \N 就把背景颜色变成血红色！

set the background color of the window to blood red!

我们还可以让它变有趣些，假设随机数是 20 \N 就把背景颜色变成血红色！

The last thing we need to do is hook this code up

最后，把代码与"事件"相连 \N 每次点按钮时 都触发代码

so that it's triggered each time our button is clicked.

最后，把代码与"事件"相连 \N 每次点按钮时 都触发代码

To do this, we need to specify that our function "handles" this event for our button,

那么，要设置事件触发时 \N 由哪个函数来处理

by adding a line to our initialize function.

我们可以在初始化函数中，加一行代码来实现

The type of event, in this case, is a click event,

我们要处理的，是"点击"事件 \N 然后函数会处理这个事件

and our function is the event handler for that event.

我们要处理的，是"点击"事件 \N 然后函数会处理这个事件

Now we're done.

现在完成了

We can click that button all day long,

可以点按钮点上一整天 \N 每次都会执行 rollD20 函数

and each time, our "roll D20" function gets dispatched and executed.

可以点按钮点上一整天 \N 每次都会执行 rollD20 函数

This is exactly what's happening behind the scenes

这就是程序背后的原理

when you press the little bold button in a text editor, or select shutdown from a dropdown menu

在编辑器里点 粗体 或菜单里选 关机 \N 一个处理该事件的函数会触发

- a function linked to that event is firing.

在编辑器里点 粗体 或菜单里选 关机 \N 一个处理该事件的函数会触发

Hope I don't roll a 20.

希望不会随机到 20

Ahhhh!

啊！！！

Ok, back to the Xerox Alto!

好，现在回到施乐奥托！

Roughly 2000 Altos were made, and used at Xerox and given to University labs.

大约制作了2000台奥托\N 有的在施乐公司内部用，有的送给大学实验室

They were never sold commercially.

从来没有商业出售过

Instead, the PARC team kept refining the hardware and software,

然而，PARC 团队不断完善硬件和软件

culminating in the Xerox Star system, released in 1981.

最终于1981年发布了 施乐之星系统

The Xerox Star extended the desktop metaphor.

施乐之星扩展了"桌面隐喻"

Now, files looked like pieces of paper,

现在文件看起来就像一张纸 \N 还可以存在文件夹里

and they could be stored in little folders,

现在文件看起来就像一张纸 \N 还可以存在文件夹里

all of which could sit on your desktop, or be put away into digital filing cabinets.

这些都可以放桌面上，或数字文件柜里

It's a metaphor that sits ontop of the underlying file system.

这样来隐喻底层的文件系统

From a user's perspective, this is a new level of abstraction!

从用户角度来看，是一层新抽象！

Xerox, being in the printing machine business, also advanced text and graphics creation tools.

施乐卖的是印刷机 \N 但在文本和图形制作工具领域也有领先

For example, they introduced the terms: cut, copy and paste.

例如，他们首先使用了\N "剪切""复制""粘贴"这样的术语

This metaphor was drawn

这个比喻来自编辑打字机文件

from how people dealt with making edits in documents written on typewriters.

这个比喻来自编辑打字机文件

You'd literally cut text out with scissors, and then paste it, with glue,

真的是剪刀"剪切" \N 然后胶水"粘贴" 到另一个文件

into the spot you wanted in another document.

真的是剪刀"剪切" \N 然后胶水"粘贴" 到另一个文件

Then you'd photocopy the page to flatten it back down into a single layer,

然后再复印一次，新文件就是一层了

making the change invisible.

看不出编辑的痕迹

Thank goodness for computers!

感谢计算机的出现！

This manual process was moot with the advent of word processing software,

文字处理软件出现后 \N 这种手工做法就消失了

which existed on platforms like the Apple II and Commodore PET.

Apple II 和 Commodore PET 上有文字处理软件

But Xerox went way beyond the competition

但施乐在这点上走的更远

with the idea that whatever you made on the computer

无论你在计算机上做什么 \N 文件打印出来应该长得一样

should look exactly like the real world version, if you printed it out.

无论你在计算机上做什么 \N 文件打印出来应该长得一样

They dubbed this What-You-See-Is-What-You-Get or WYSIWYG.

他们叫这个"所见即所得"

Unfortunately, like Engelbart's oN-Line System,

不幸的是，就像恩格尔巴特的 oN-Line System

the Xerox Star was ahead of its time.

施乐之星也领先于那个时代，销售量不高

Sales were sluggish

施乐之星也领先于那个时代，销售量不高

because it had a price tag equivalent to nearly $200,000 today for an office setup.

因为在办公室里配一个，相当如今20万美元

It also didn't help that the IBM PC launched that same year,

IBM 同年推出了 IBM PC

followed by a tsunami of cheap "IBM Compatible" PC Clones.

之后便宜的"IBM兼容"计算机席卷市场

But the great ideas that PARC researchers had been cultivating

但 PARC 研究人员花了十几年做的这些 \N 没有被浪费

and building for almost a decade didn't go to waste.

但 PARC 研究人员花了十几年做的这些 \N 没有被浪费

In December of 1979, a year and a half before the Xerox Star shipped,

1979年12月，施乐之星出货前一年半

a guy you may have heard of visited: Steve Jobs.

有个人去施乐公司参观 \N 你可能听说过这个人：史蒂夫·乔布斯

There's a lot of lore surrounding this visit,

这次参观有很多传闻

with many suggesting that Steve Jobs and Apple stole Xerox's ideas.

许多人认为\N 乔布斯和苹果偷走了施乐的创意

But that simply isn't true.

但那不是事实

In fact, Xerox approached Apple, hoping to partner with them.

事实上是施乐公司主动找苹果，希望合作

Ultimately, Xerox was able to buy a million dollar stake in Apple

最终施乐还买了苹果的一百万美元股份

before its highly anticipated I.P.O.

在苹果备受瞩目的 首次公开募股(IPO) 前买的

-but it came with an extra provision:

但一个额外条款是：

"disclose everything cool going on at Xerox PARC".

"公布一切施乐研究中心正在进行的酷工作"

Steve knew they had some of the greatest minds in computing,

史蒂夫知道他们很厉害

but he wasn't prepared for what he saw.

但他完全没预想到这些

There was a demonstration of Xerox's graphical user interface,

其中有个演示是

running on a crisp, bitmapped display,

一个清晰的位图显示器上，运行着施乐公司的图形界面 \N 操作全靠鼠标直观进行

all driven with intuitive mouse input.

一个清晰的位图显示器上，运行着施乐公司的图形界面 \N 操作全靠鼠标直观进行

Steve later said, "It was like a veil being lifted from my eyes.

史蒂夫后来说："就像拨开了眼前的一层迷纱

I could see the future of what computing was destined to be."

我可以看到计算机产业的未来"

Steve returned to Apple with his engineering entourage,

史蒂夫和随行的工程师回到苹果公司，开始开发新功能

and they got to work inventing new features,

史蒂夫和随行的工程师回到苹果公司，开始开发新功能

like the menu bar and a trash can to store files to be deleted;

比如菜单栏和垃圾桶，垃圾桶存删除文件

it would even bulge when full - again with the metaphors.

满了甚至会膨胀 - 再次使用了隐喻

Apple's first product with a graphical user interface, and mouse,

苹果第一款有图形界面和鼠标的产品

was the Apple Lisa, released in 1983.

是 1983 年发行的 Apple Lisa

It was a super advanced machine, with a super advanced price

一台超级先进的机器，标了"超级先进"的价格

- almost 25 thousand dollars today.

- 差不多是如今的 25000 美元

That was significantly cheaper than the Xerox Star,

虽然比施乐之星便宜不少

but it turned out to be an equal flop in the market.

但在市场上同样失败

Luckily, Apple had another project up its sleeve:

幸运的是，苹果还有另一个项目：

The Macintosh, released a year later, in 1984.

Macintosh，于 1984 年发布

It had a price of around 6,000 dollars today - a quarter of the Lisa's cost.

价格大约是如今的6000美元 - Lisa 的四分之一

And it hit the mark, selling 70,000 units in the first 100 days.

它成功了，开售100天就卖了7万台

But after the initial craze, sales started to falter,

但在最初的狂潮后，销售额开始波动

and Apple was selling more of its Apple II computers than Macs.

苹果公司卖的 Apple II 比 Mac 多

A big problem was that no one was making software for this new machine

一个大问题是：没人给这台新机器做软件

with it's new radical interface.

一个大问题是：没人给这台新机器做软件

And it got worse. The competition caught up fast.

之后情况变得更糟，竞争对手赶上来了

Soon, other personal computers had primitive,

不久，其它价格只有 Mac 几分之一的个人计算机 \N 有了原始但可用的图形界面

but usable graphical user interfaces on computers a fraction of the cost.

不久，其它价格只有 Mac 几分之一的个人计算机 \N 有了原始但可用的图形界面

Consumers ate it up, and so did PC software developers.

消费者认可它们， PC 软件开发者也认可

With Apple's finances looking increasingly dire,

随着苹果的财务状况日益严峻 \N 以及和苹果新 CEO 约翰·斯卡利 的关系日益紧张

and tensions growing with Apple's new CEO, John Sculley,

随着苹果的财务状况日益严峻 \N 以及和苹果新 CEO 约翰·斯卡利 的关系日益紧张

Steve Jobs was ousted.

史蒂夫乔布斯被赶出了苹果公司

A few months later, Microsoft released Windows 1.0.

几个月后，微软发布了 Windows 1.0

It may not have been as pretty as Mac OS,

它也许不如 Mac OS 漂亮

but it was the first salvo in what would become a bitter rivalry

但让微软在市场中站稳脚跟 \N 奠定了统治地位

and near dominance of the industry by Microsoft.

但让微软在市场中站稳脚跟 \N 奠定了统治地位

Within ten years, Microsoft Windows was running on almost 95% of personal computers.

十年内，95％的个人计算机上都有微软的 Windows

Initially, fans of Mac OS could rightly claim superior graphics and ease-of-use.

最初，Mac OS 的爱好者还可以说\N Mac 有卓越的图形界面和易用性

Those early versions of Windows were all built on top of DOS,

Windows 早期版本都是基于 DOS \N 而 DOS 设计时 没想过运行图形界面

which was never designed to run GUIs.

Windows 早期版本都是基于 DOS \N 而 DOS 设计时 没想过运行图形界面

But, after Windows 3.1,

但 Windows 3.1 之后

Microsoft began to develop a new consumer-oriented OS

微软开始开发新的，\N 面向消费者的 GUI 操作系统

with upgraded GUI called Windows 95.

叫 Windows 95

This was a significant rewrite that offered much more than just polished graphics.

这是一个意义非凡的版本 \N 不仅提供精美的界面

It also had advanced features Mac OS didn't have,

还有 Mac OS 没有的高级功能

like program multitasking and protected memory.

比如"多任务"和"受保护内存"

Windows 95 introduced many GUI elements still seen in Windows versions today,

Windows 95 引入了许多 \N 如今依然见得到的 GUI 元素

like the Start menu, taskbar, and Windows Explorer file manager.

比如开始菜单，任务栏和 Windows 文件管理器

Microsoft wasn't infallible though.

不过微软也失败过

Looking to make the desktop metaphor even easier and friendlier,

为了让桌面更简单友好 \N 微软开发了一个产品叫 Microsoft Bob

it worked on a product called Microsoft Bob,

为了让桌面更简单友好 \N 微软开发了一个产品叫 Microsoft Bob

and it took the idea of using metaphors to an extreme.

将比喻用到极致

Now you had a whole virtual room on your screen,

现在屏幕上有了一个虚拟房间

with applications embodied as objects that you could put on tables and shelves.

程序是物品，可以放在桌子和书架上

It even came with a crackling fireplace and a virtual dog to offer assistance.

甚至还有噼啪作响的壁炉 \N 和提供帮助的虚拟狗狗

And you see those doors on the sides?

你看到那边的门没？

Yep, those went to different rooms in your computer

是的，那些门通往不同房间 \N 房间里有不同程序

where different applications were available.

是的，那些门通往不同房间 \N 房间里有不同程序

As you might have guessed,

你可能猜到了，它没有获得成功

it was not a success.

你可能猜到了，它没有获得成功

This is a great example of how the user interfaces we enjoy today

这是一个好例子，说明如今的用户界面

are the product of what's essentially natural selection.

是自然选择后的结果

Whether you're running Windows, Mac, Linux, or some other desktop GUI,

无论你用的是 \N Windows，Mac，Linux 或其他 GUI

it's almost certainly an evolved version of the WIMP paradigm first introduced on the Xerox Alto.

几乎都是施乐奥托 WIMP 的变化版

Along the way, a lot of bad ideas were tried, and failed.

一路上，人们试了各种做法并失败了

Everything had to be invented, tested, refined, adopted or dropped.

一切都必须发明，测试，改进，适应或抛弃

Today, GUIs are everywhere and while they're good,

如今，图形界面无处不在 \N 使用体验一般只是可以接受，而不是非常好

they are not always great.

如今，图形界面无处不在 \N 使用体验一般只是可以接受，而不是非常好

No doubt you've experienced design-related frustrations

你肯定体验过差劲的设计

after downloading an application, used someone else's phone,

比如下载了很烂的 App，用过别人糟糕的手机

or visited a website. And for this reason,

或者看到过很差的网站，因此

computer scientists and interface designers continue to work hard

计算机科学家和界面设计师 \N 会继续努力工作

to craft computing experiences that are both easier and more powerful.

做出更好更强大的界面

Ultimately, working towards Engelbart's vision of augmenting human intellect.

朝着恩格尔巴特"增强人类智能"的愿景努力

I'll see you next week.

我们下周见

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

嗨，我是 Carrie Anne \N 欢迎收看计算机科学速成课！

Over the past five episodes,

在过去五集

we've worked up from text-based teletype interfaces to pixelated bitmapped graphics.

我们从基于电传打字机的命令行界面 \N 讲到图形怎么显示到屏幕上

Then, last episode,we covered Graphical User Interfaces and all

再到上集的 图形用户界面（GUI）

their "Ooey Gooey" richness.

以及图形界面的美味

All of these examples have been 2D. But of course "we are living in a 3D world

之前的例子都是2D, 但我们生活的世界是3D的

and I'm a 3 dimensional girl!

我也是个三维 girl~

So today, we're going to talk about some fundamental methods in 3D computer graphics

所以今天，我们讲3D图形的基础知识

and how you render them onto a 2D screen.

以及如何渲染 3D 图形到 2D 屏幕上

As we discussed in episode 24 we can write functions that draw a line between any two points like A and B.

24集中说过 \N 可以写一个函数，从A到B画一条线

By manipulating the X and Y coordinates of points A and B, we can manipulate the line.

通过控制 A 和 B 的(X,Y)坐标，可以控制一条线

In 3D graphics, points have not just two coordinates, but three -- X, Y and Z.

在3D图像中, 点的坐标不再是两点, 而是三点, X,Y,Z

Or "zee" but I'm going to say "zed".

或读"Zee"，但我之后会读成"Zed"

Of course, we don't have X/Y/Z coordinates on a 2D computer screen

当然，2D的电脑屏幕上\N不可能有 XYZ 立体坐标轴

so graphics algorithms are responsible for "flattening" 3D coordinates onto a 2D plane.

所以有图形算法 \N 负责把3D坐标"拍平"显示到2D屏幕上

This process is known as 3D Projection.

这叫"3D投影"

Once all of the points have been converted from 3D to 2D

所有的点都从3D转成2D后

we can use the regular 2D line drawing function to connect the dots… literally.

就可以用画2D线段的函数 来连接这些点

This is called Wireframe Rendering.

这叫 "线框渲染"

Imagine building a cube out of chopsticks, and shining a flashlight on it.

想象用筷子做一个立方体，然后用手电筒照它

The shadow it casts onto your wall - its projection - is flat.

墙上的影子就是投射，是平的

If you rotate the cube around

如果旋转立方体

you can see it's a 3D object, even though it's a flat projection.

投影看起来会像 3D 物体，尽管是投影面是平的

This transformation from 3D to 2D is exactly what your computer is doing

电脑也是这样3D转2D

just with a lot more math… and less chopsticks.

只不过用大量数学，而不是筷子

There are several types of 3D Projection.

3D投影有好几种

What you're seeing right now is an Orthographic Projection

你现在看到的，叫 正交投影

where, for example, the parallel sides in the cube appear as parallel in the projection.

立方体的各个边，在投影中互相平行

In the real 3D world through, parallel lines converge as they get further from the viewer

在真实3D世界中,平行线段会在远处收敛于一点

like a road going to the horizon.

就像远处的马路汇聚到一点

This type of 3D projection is called Perspective Projection .

这叫 透视投射

It's the same process, just with different math.

过程是类似的，只是数学稍有不同

Sometimes you want perspective and sometimes you don't --

有时你想要透视投影，有时不想

the choice is up to the developer.

具体取决于开发人员

Simple shapes, like cubes, are easily defined by straight lines.

如果想画立方体这种简单图形，直线就够了

But for more complex shapes, triangles are better

但更复杂的图形，三角形更好

-- what are called polygons in 3D graphics.

在3D图形学中 \N 我们叫三角形"多边形"(Polygons)

Look at this beautiful teapot made out of polygons.

看看这个多边形组成的 漂亮茶壶

A collection of polygons like this is a mesh

一堆多边形的集合叫 网格

The denser the mesh, the smoother the curves and the finer the details.

网格越密，表面越光滑，细节越多

But, that also increases the polygon count, which means more work for the computer

但意味着更多计算量

Game designers have to carefully balance model fidelity vs. polygon count,

游戏设计者要平衡角色的真实度 \N 和多边形数量

because if the count goes too high

如果数量太多 \N 帧率会下降到肉眼可感知，用户会觉得卡

the framerate of an animation drops below what users perceive as smooth.

如果数量太多 \N 帧率会下降到肉眼可感知，用户会觉得卡

For this reason, there are algorithms for simplifying meshes.

因此有算法用来简化网格

The reason triangles are used,

之所以三角形更常用 \N 而不是用正方形，或其它更复杂的图形

and not squares, or polygons, or some other more complex shape

之所以三角形更常用 \N 而不是用正方形，或其它更复杂的图形

is simplicity:

是因为三角形的简单性

three points in space unambiguously define a plane.

空间中 三点定义一个平面

If you give me three points in a 3D space, I can draw a plane through it

如果给3个3D点，我能画出一个平面

there is only one.. single.. answer.

而且只有这一个答案

This isn't guaranteed to be true for shapes with four or more points.

4个或多于4个点就不一定了

Also two points aren't enough to define a plane, only a line,

而2个点不够定义平面，只能定义线段

so three is the perfect and minimal number. Triangles for the win!

所以3是最完美的数字，三角形万岁

Wireframe rendering is cool and all - sorta retro - but of course 3D graphics can also be filled.

线框渲染 虽然很酷，但3D图像需要填充

The classic algorithm for doing this is called Scanline Rendering,

填充图形的经典算法叫 扫描线渲染 (Scanline Rendering) \N 于1967年诞生在犹他州大学

first developed in 1967 at the University of Utah

填充图形的经典算法叫 扫描线渲染 (Scanline Rendering) \N 于1967年诞生在犹他州大学

For a simple example, let's consider just one polygon.

为了例子简单，我们只看一个多边形

Our job here is to figure out how this polygon translates to filled pixels on a computer screen

我们要思考 \N 这个多边形如何转成一块填满像素的区域

so let's first overlay a grid of pixels to fill

我们先铺一层像素网格

The scanline algorithm starts by reading the three points that make up the polygon

扫描线算法 先读多边形的3个点

and finding the lowest and highest Y values. It will only consider rows between these two points.

找最大和最小的Y值，只在这两点间工作

Then, the algorithm works down one row at a time.

然后算法从上往下，一次处理一行

In each row, it calculates where a line running through

计算每一行和多边形相交的2个点

the center of a row - intersects with the side of the polygon.

计算每一行和多边形相交的2个点

Because polygons are triangles, if you intersect one line, you have to intersect with another.

因为是三角形,如果相交一条边, 必然相交另一条

It's guaranteed!

因为是三角形,如果相交一条边, 必然相交另一条

The job of the scanline algorithm is to fill in the pixels between the two intersections.

扫描线算法 会填满2个相交点之间的像素

Let's see how this works.

来看个具体例子

On the first row we look at we intersect here and here.

第一行 相交于这里和这里

The algorithm then colors in all pixels between those two intersections.

算法把两点间填满颜色

And this just continues, row by row, which is why it's called Scan... Line... Rendering.

然后下一行，再下一行，所以叫 扫描..线..渲染

When we hit the bottom of the polygon, we're done.

扫到底部就完成了

The rate at which a computer fills in polygons is called the fillrate.

填充的速度叫 fillrate（填充速率）

Admittedly, this is a pretty ugly filled polygon. It has what are known as "Jaggies" rough edges.

当然 这样的三角形比较丑，边缘满是锯齿

This effect is less pronounced when using smaller pixels.

当像素较小时 就不那么明显

But nonetheless, you see these in games all the time, especially on lower powered platforms.

但尽管如此 \N 你肯定在游戏里见过这种效果，特别是低配电脑

One method to soften this effect is Antialiasing.

一种减轻锯齿的方法叫\N 抗锯齿(Antialiasing)

Instead of filling pixels in a polygon with the same color,

与其每个像素都涂成一样的颜色

we can adjust the color based on how much the polygon cuts through each pixel

可以判断多边形切过像素的程度，来调整颜色

If a pixel is entirely inside of a polygon,it gets fully colored.

如果像素在多边形内部，就直接涂颜色

But if the polygon only grazes a pixel, it'll get a lighter shade.

如果多边形划过像素，颜色就浅一些

This feathering of the edges is much more pleasant to the eyes.

这种边缘羽化的效果 看着更舒服些

Antialiasing is used all over the place, including in 2D graphics, like fonts and icons.

抗锯齿 被广泛使用，比如字体和图标

If you lean in real close to your monitor..

如果你把脸贴近屏幕

Closer, Closer.

近点..再近点

You'll see all the fonts in your browser are Antialiased. So smooth!

你能看到浏览器里字体是抗锯齿的，超平滑

In a 3D scene, there are polygons that are part objects in the back, near the front,and just about everywhere.

在3D场景中，多边形到处都是

Only some are visible,

但只有一部分能看见

because some objects are hidden behind other objects in the scene

因为其它的被挡住了

-- what's called occlusion .

这叫 遮挡

The most straightforward way to handle this is to use a sort algorithm,

最直接的处理办法是用排序算法

and arrange all the polygons in the scene from farthest to nearest, then render them in that order.

从远到近排列，然后从远到近渲染

This is called the Painter's Algorithm , because painters also have to start with the background

这叫 画家算法 因为画家也是先画背景

and then increasingly work up to foreground elements.

然后再画更近的东西

Consider this example scene with three overlapping polygons.

看这个例子，有3个重叠的多边形

To make things easier to follow, we're going to color the polygons differently.

为了简单，我们画成不同颜色

Also for simplicity, we'll assume these polygons are all parallel to the screen

同时，假设3个多边形都和屏幕平行

but in a real program, like a game,

但在实际应用中, 比如游戏里\N多边形可能是倾斜的

the polygons can be tilted in 3D space.

但在实际应用中, 比如游戏里\N多边形可能是倾斜的

Our three polygons, A B and C… are at distance 20, 12 and 14.

3个多边形A,B,C，距离20,12,14

The first thing the Painter's Algorithm does is sort all the polygons, from farthest to nearest.

画家算法的第一件事，是从远到近排序

Now that they're in order, we can use scanline rendering to fill each polygon, one at a time.

现在有序了，我们可以用 扫描线算法 填充多边形，一次填一个

We start with Polygon A, the farthest one away.

我们从最远的A开始

Then we repeat the process for the next farthest polygon, in this case, C.

然后重复这个过程，填充第二远的C

And then we repeat this again, for Polygon B.

然后是 B

Now we're all done, and you can see the ordering is correct. The polygons that are closer, are in front!

现在完成了，可以看到顺序是对的，近的多边形在前面！

An alternative method for handling occlusion is called Z-Buffering .

还有一种方法叫 深度缓冲

It achieves the same output as before, but with a different algorithm.

它和之前的算法做的事情一样，但方法不同

Let's go back to our previous example, before it was sorted.

我们回到之前的例子，回到排序前的状态

That's because this algorithm doesn't need to sort any polygons, which makes it faster.

因为这个算法不用排序，所以速度更快

In short, Z-buffering keeps track of the closest distance

简而言之，Z-buffering 算法会记录

to a polygon for every pixel in the scene.

场景中每个像素和摄像机的距离

It does this by maintaining a Z-Buffer, which is just a matrix of values that sits in memory.

在内存里存一个数字矩阵

At first, every pixel is initialized to infinity.

首先，每个像素的距离被初始化为"无限大"

Then Z-buffering starts with the first polygon in its list. In this case, that's A.

然后 Z-buffering 从列表里第一个多边形开始处理，也就是A

It follows the same logic as the scanline algorithm, but instead of coloring in pixels,

它和扫描线算法逻辑相同，但不是给像素填充颜色

it checks the distance of the polygon versus what's recorded in its Z-Buffer.

而是把多边形的距离\N和 Z-Buffer 里的距离进行对比

It records the lower of the two values.

它总是记录更低的值

For our Polygon A, with a distance of 20, it wins against infinity every time.

A距离20，20小于"无限大"，所以缓冲区记录20

When it's done with Polygon A, it moves on to the next polygon in its list, and the same thing happens.

算完A之后算下一个，以此类推

Now, because we didn't sort the polygons,

因为没对多边形排序

it's not always the case that later polygons overwrite high values.

所以后处理的多边形并不总会覆盖前面的

In the case of Polygon C,

对于多边形C

only some of the values in the Z-buffer get new minimum distances.

缓冲区里只有一部分值会被多边形C的距离值覆盖

This completed Z-buffer is used in conjunction with a fancier version of scanline rendering

Z缓冲区完成后，会和"扫描线"算法的改进高级版配合使用

that not only tests for line intersection,

不仅可以勘测到线的交叉点

but also does a look up to see if that pixel will even be visible in the final scene.

还可以知道某像素是否在最终场景中可见

If it's not, the algorithm skips it and moves on.

如果不可见，扫描线算法会跳过那个部分

An interesting problem arises when two polygons have the same distance,

当两个多边形距离相同时 \N 会出现一个有趣问题

like if Polygon A and B are both at a distance of 20. Which one do you draw on top?

比如多边形 A 和 B 距离都是 20, 哪个画上面?

Polygons are constantly being shuffled around in memory and changing their access order.

多边形会在内存中移来移去，访问顺序会不断变化

Plus, rounding errors are inherent in floating point computations.

另外,计算浮点数有舍入误差

So, which one gets drawn on top is often unpredictable.

所以哪一个画在上面, 往往是不可预测的

The result is a flickering effect called Z-Fighting, which if you've played 3D games no doubt encountered.

导致出现 Z-fighting 效果 \N 如果你玩过3D游戏，肯定见过

Speaking of glitches, another common optimization in 3D graphics is called Back-Face Culling.

说起 故障，3D游戏中有个优化叫 背面剔除

If you think about it, a triangle has two sides, a front and a back.

你想想,三角形有两面,正面和背面

With something like the head of an avatar, or the ground in a game,

游戏角色的头部或地面，只能看到朝外的一面

you should only ever see one side -- the side facing outwards.

游戏角色的头部或地面，只能看到朝外的一面

So to save processing time, the back-side of polygons are often ignored in the rendering pipeline

所以为了节省处理时间，会忽略多边形背面

which cuts the number of polygon faces to consider in half.

减了一半多边形面数

This is great, except when there's a bug that lets you get inside of those objects,and look outwards.

这很好,但有个bug是 如果进入模型内部往外看

Then the avatar head or ground becomes invisible.

头部和地面会消失

Moving on. We need to talk about lighting -- also known as shading

继续，我们讲灯光，也叫 明暗处理

because if it's a 3D scene, the lighting should vary over the surface of objects.

因为3D场景中, 物体表面应该有明暗变化

Let's go back to our teapot mesh.

我们回到之前的茶壶网格

With scanline rendering coloring in all the polygons, our teapot looks like this.

用"扫描线"算法渲染所有多边形后，茶壶看起来像这样

Not very 3D.

没什么 3D 感

So, let's add some lighting to enhance the realism!

我们来加点灯光，提高真实感

As an example, we'll pick 3 polygons from different parts of our teapot.

为了举例，我们从茶壶上挑3个不同位置的多边形

Unlike our previous examples, we're now going to consider how these polygons are oriented in 3D space

和之前的例子不同，这次要考虑这些多边形面对的方向

they're no longer parallel to the screen, but rather tilted in different 3D directions.

它们不平行于屏幕，而是面对不同方向

The direction they face is called the Surface Normal ,

他们面对的方向叫 " 表面法线 "

and we can visualize that direction with a little 3D arrow that's perpendicular to the polygon's surface.

我们可以用一个垂直于表面的小箭头\N来显示这个方向

Now let's add a light source.

现在加个光源

Each polygon is going to be illuminated a different amount. Some will appear brighter

每个多边形被照亮的程度不同 有的更亮

because their angle causes more light to be reflected towards the viewer.

因为面对的角度\N导致更多光线反射到观察者

For example, the bottom-most polygon is tilted downwards,

举个例子，底部的多边形向下倾斜

away from the light source, which means it's going to be dark.

远离光源，所以更暗一些

In a similar way, the rightmost polygon is slightly facing away from the light,

类似的，最右的多边形更背对光源

so it will be partially illuminated.

所以只有部分照亮

And finally, there's the upper-left polygon.

最后是左上角的多边形

Its angle means that it will reflect light from the light source towards our view.

因为它面对的角度 意味着会把光线反射到我们这里

So, it'll appear bright.

所以会显得更亮

If we do this for every polygon, our teapot looks like this which is much more realistic!

如果对每个多边形执行同样的步骤，看上去会更真实!

This approach is called Flat Shading, and it's the most basic lighting algorithm.

这叫 平面着色，是最基本的照明算法

Unfortunately, it also makes all those polygon boundaries really noticeable

不幸的是，这使多边形的边界非常明显，看起来不光滑

and the mesh doesn't look smooth.

不幸的是，这使多边形的边界非常明显，看起来不光滑

For this reason, more advanced lighting algorithms were developed,

因此开发了更多算法

such as Gouraud Shading and Phong Shading .

比如 高洛德着色 和 冯氏着色

Instead of coloring in polygons using just one colour,

不只用一种颜色给整个多边形上色

they vary the colour across the surface in clever ways,

而是以巧妙的方式改变颜色

which results in much nicer output.

得到更好的效果

We also need to talk about textures ,

我们还要说下" 纹理 "

which in graphics refers to the look of a surface,rather than its feel.

纹理在图形学中指外观，而不是手感

Like with lighting, there are many algorithms with all sorts of fancy effects.

就像照明算法一样，\N 纹理也有多种算法，来做各种花哨效果

The simplest is texture mapping .

最简单的是 纹理映射

To visualize this process,let's go back to our single polygon.

为了理解纹理映射，回到单个多边形

When we're filling this in, using scanline rendering,

用"扫描线算法"填充时

we can look up what color to use at every pixel according to a texture image saved in memory.

可以看看内存内的纹理图像 决定像素用什么颜色

To do this, we need a mapping between the polygon's coordinates and the texture's coordinates.

为了做到这点，\N 需要把多边形坐标和纹理坐标对应起来

Let's jump to the first pixel that scanline rendering needs to fill in.

我们来看看"扫描线算法"要填充的第一个像素

The texturing algorithm will consult the texture in memory,

纹理算法会查询纹理

take the average color from the corresponding region, and fill the polygon accordingly.

从相应区域取平均颜色，并填充多边形

This process repeats for all pixels in the polygon, and that's how we get textures.

重复这个过程，就可以获得纹理

If you combine all the techniques we've talked about this episode, you get a wonderfully funky little teapot.

如果结合这集提到的所有技巧 \N 会得到一个精美的小茶壶

And this teapot can sit in an even bigger scene, comprised of millions of polygons.

这个茶壶可以放进更大的场景里 \N 场景由上百万个多边形组成

Rendering a scene like this takes a fair amount of computation.

渲染这样的场景需要大量计算

But importantly, it's the same type of calculations being performed

但重要的是，再大的场景，过程都是一样的 \N 一遍又一遍，处理所有多边形

over and over and over again for many millions of polygons –

但重要的是，再大的场景，过程都是一样的 \N 一遍又一遍，处理所有多边形

scanline filling, antialiasing, lighting, and texturing.

扫描线填充, 抗锯齿, 光照, 纹理化

However there are a couple of ways to make this much faster!

然而，有几种方法可以加速渲染

First off, we can speed things up by having special hardware

首先，我们可以为这种特定运算 \N 做专门的硬件来加快速度，让运算快如闪电

with extra bells and whistles just for these specific types of computations, making them lightning fast.

首先，我们可以为这种特定运算 \N 做专门的硬件来加快速度，让运算快如闪电

And secondly,we can divide up a 3D scene into many smaller parts,

其次，我们可以把3D场景分解成多个小部分

and then render all the pieces in parallel,rather than sequentially.

然后并行渲染，而不是按顺序渲染

CPU's aren't designed for this, so they aren't particularly fast.

CPU不是为此设计的，因此图形运算不快

So, computer engineers created special processors just for graphics

所以，计算机工程师为图形做了专门的处理器

– a GPU, or Graphics Processing Unit.

叫 GPU "图形处理单元"

These can be found on graphics cards inside of your computer, along with RAM reserved for graphics.

GPU 在显卡上，周围有专用的 RAM

This is where all the meshes and textures live,

所有网格和纹理都在里面

allowing them to be accessed super fast by many different cores of the GPU all at once.

让 GPU 的多个核心可以高速访问

A modern graphics card, like a GeForce GTX 1080 TI,

现代显卡，如 GeForce GTX 1080 TI

contains 3584 processing cores, offering massive parallelization.

有3584个处理核心，提供大规模并行处理

It can process hundreds of millions of polygons every second!

每秒处理上亿个多边形！

Ok, that concludes our whistle stop tour of 3D graphics.

好了，本集对3D图形的介绍到此结束

Next week, we switch topics entirely.

下周我们聊全新的主题

I'll ping you then.

我到时会 ping 你~

Hi, I’m Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

The internet is amazings

互联网太棒啦

In just a few keystrokes, we can stream videos on Youtube -- Hello!

键盘敲几下就能在 Youtube 直播--哈喽！

- read articles on Wikipedia,

在维基百科上阅读文章

order supplies on amazon, video chat with friends, and tweet about the weather.

在亚马逊买东西 和朋友视频 发一条天气推特

Without a doubt, the ability for computers, and their users, to send and receive information

毫无疑问，\N 用户在全球网络中发送和接收信息的能力

over a global telecommunications network forever changed the world.

永远改变了这个世界

150 years ago, sending a letter from London to California would have taken two to three

150年前 发一封信件从伦敦到加州 要花2~3周

weeks, and that’s if you paid for express mail.

而且还是特快邮件

Today, that email takes a fraction of a second.

如今 电子邮件只要几分之一秒.

This million fold improvement in latency, that’s the time it takes for a message to

"时延"改善了上百万倍 \N (时延指传播一条信息所需的时间)

transfer, juiced up the global economy helping the modern world to

振兴了全球经济 \N 帮助现代世界在遍布全球的光纤中快速发展

move at the speed of light on fiber optic cables spanning the globe.

振兴了全球经济 \N 帮助现代世界在遍布全球的光纤中快速发展

You might think that computers and networks always went hand in hand, but actually most

你可能觉得计算机和网络密切相关，但事实上，

computers pre-1970 were humming away all alone.

1970年以前 大多数计算机是独立运行的

However, as big computers began popping up everywhere,

然而 因为大型计算机开始随处可见

and low cost machines started to show up on people’s desks,

廉价机器开始出现在书桌上

it became increasingly useful to share data and resources,

分享数据和资源渐渐变得有用起来

and the first networks of computers appeared.

首个计算机网络出现了

Today, we’re going to start a three-episode arc on how computer networks came into being

今天起，我们花3集视频讲网络是如何发展成现在的样子

and the fundamental principles and techniques that power them.

以及支撑它们的基础原理和技术

The first computer networks appeared in the 1950s and 60s.

第一个计算机网络出现在1950~1960年代

They were generally used within an organization – like a company or research lab

通常在公司或研究室内部使用，为了方便信息交换

to facilitate the exchange of information between different people and computers.

通常在公司或研究室内部使用，为了方便信息交换

This was faster and more reliable than the previous method of having someone walk a pile

比把纸卡或磁带送到另一栋楼里更快速可靠

of punch cards, or a reel of magnetic tape, to a computer on the other side of a building

比把纸卡或磁带送到另一栋楼里更快速可靠

‒ which was later dubbed a sneakernet.

这叫"球鞋网络"

A second benefit of networks was the ability to share physical resources.

第二个好处是能共享物理资源

For example, instead of each computer having its own printer,

举个例子，与其每台电脑配一台打印机

everyone could share one attached to the network.

大家可以共享一台联网的打印机

It was also common on early networks to have large, shared, storage drives,

早期网络也会共享存储空间

ones too expensive to have attached to every machine.

因为每台电脑都配存储器太贵了

These relatively small networks of close-by computers

计算机近距离构成的小型网络 \N 叫局域网， 简称LAN

are called Local Area Networks, or LANs.

计算机近距离构成的小型网络 \N 叫局域网， 简称LAN

A LAN could be as small as two machines in the same room,

局域网能小到是同一个房间里的两台机器

or as large as a university campus with thousands of computers.

或大到校园里的上千台机器

Although many LAN technologies were developed and deployed,

尽管开发和部署了很多不同 LAN 技术

the most famous and successful was Ethernet, developed in the

其中最著名和成功的是"以太网" , 开发于1970年代 \N 在施乐的"帕洛阿尔托研究中心"诞生, 今日仍被广泛使用

early 1970s at Xerox PARC, and still widely used today.

其中最著名和成功的是"以太网" , 开发于1970年代 \N 在施乐的"帕洛阿尔托研究中心"诞生, 今日仍被广泛使用

In its simplest form, a series of computers are connected to a single, common ethernet cable.

以太网的最简单形式是：一条以太网电线连接数台计算机

When a computer wants to transmit data to another computer,

当一台计算机要传数据给另一台计算机时

it writes the data, as an electrical signal, onto the cable.

它以电信号形式，将数据传入电缆

Of course, because the cable is shared, every computer plugged into the network sees the

当然 因为电缆是共享的 \N 连在同一个网络里的其他计算机也看得到数据

transmission, but doesn’t know if data is intended for them or another computer.

但不知道数据是给它们的，还是给其他计算机的

To solve this problem, Ethernet requires that each computer has a unique

为了解决这个问题 以太网需要每台计算机有唯一的 \N 媒体访问控制地址 简称 MAC地址

Media Access Control address, or MAC address.

为了解决这个问题 以太网需要每台计算机有唯一的 \N 媒体访问控制地址 简称 MAC地址

This unique address is put into a header that prefixes any data sent over the network.

这个唯一的地址放在头部，作为数据的前缀发送到网络中

So, computers simply listen to the ethernet cable,

所以，计算机只需要监听以太网电缆 \N 只有看到自己的 MAC 地址，才处理数据

and only process data when they see their address in the header.

所以，计算机只需要监听以太网电缆 \N 只有看到自己的 MAC 地址，才处理数据

This works really well; every computer made today comes with its own unique MAC address

这运作得很好 现在制造的每台计算机都自带唯一的MAC地址

for both Ethernet and WiFi.

用于以太网和无线网络

The general term for this approach is Carrier Sense Multiple Access, or CSMA for short.

多台电脑共享一个传输媒介，\N 这种方法叫 "载波侦听多路访问" 简称"CSMA"

The "carrier", in this case, is any shared transmission medium that carries data

载体(carrier)指运输数据的共享媒介

copper wire in the case of ethernet, and the air carrying radio waves for WiFi.

以太网的"载体"是铜线 \N WiFi 的"载体"是传播无线电波的空气

Many computers can simultaneously sense the carrier,

很多计算机同时侦听载体

hence the "Sense" and "Multiple Access",

所以叫"侦听"和"多路访问"

and the rate at which a carrier can transmit data is called its Bandwidth.

而载体传输数据的速度 叫"带宽"

Unfortunately, using a shared carrier has one big drawback.

不幸的是 使用共享载体有个很大的弊端

When network traffic is light, computers can simply wait for silence on the carrier,

当网络流量较小时 计算机可以等待载体清空

and then transmit their data.

然后传送数据

But, as network traffic increases, the probability that

但随着网络流量上升 两台计算机想同时写入数据的概率也会上升

two computers will attempt to write data at the same time also increases.

但随着网络流量上升 两台计算机想同时写入数据的概率也会上升

This is called a collision, and the data gets all garbled up,

这叫冲突 数据全都乱套了

like two people trying to talk on the phone at the same time.

就像两个人同时在电话里讲话

Fortunately, computers can detect these collisions by listening to the signal on the wire.

幸运的是 计算机能够通过监听电线中的信号检测这些冲突

The most obvious solution is for computers to stop transmitting,

最明显的解决办法是停止传输

wait for silence, then try again.

等待网络空闲, 然后再试一遍

Problem is, the other computer is going to try that too,

问题是 其他计算机也打算这样做

and other computers on the network that have been waiting for the

其他等着的计算机可能在任何停顿间隙闯入

carrier to go silent will try to jump in during any pause.

其他等着的计算机可能在任何停顿间隙闯入

This just leads to more and more collisions.

导致越来越多冲突

Soon, everyone is talking over one another and has a backlog of things they need to say,

很快，每个人都一个接一个地讲话 而且有一堆事要说

like breaking up with a boyfriend over a family holiday dinner.

就像在家庭聚餐中和男朋友分手一样

Terrible idea!

馊主意！

Ethernet had a surprisingly simple and effective fix.

以太网有个超简单有效的解决方法

When transmitting computers detect a collision,

当计算机检测到冲突 就会在重传之前等待一小段时间

they wait for a brief period before attempting to re-transmit.

当计算机检测到冲突 就会在重传之前等待一小段时间

As an example, let’s say 1 second.

因为要举例，假设是 1 秒好了

Of course, this doesn’t work if all the computers use the same wait duration

当然 如果所有计算机用同样的等待时间 是不行的

-- they’ll just collide again one second later.

它们会在一秒后再次冲突

So, a random period is added: one computer might wait 1.3 seconds,

所以加入一个随机时间 一台计算机可能等1.3秒

while another waits 1.5 seconds.

另一台计算机等待1.5秒

With any luck, the computer that waited 1.3 seconds will wake up,

要是运气好 等1.3秒的计算机会醒来

find the carrier to be silent, and start transmitting.

发现载体是空闲的 然后开始传输

When the 1.5 second computer wakes up a moment later, it’ll see the carrier is in use,

当1.5秒的计算机醒来后 会发现载体被占用 \N 会等待其他计算机完成

and will wait for the other computer to finish.

当1.5秒的计算机醒来后 会发现载体被占用 \N 会等待其他计算机完成

This definitely helps, but doesn’t totally solve the problem, so an extra trick is used.

这有用 但不能完全解决问题 所以要用另一个小技巧

As I just explained, if a computer detects a collision while transmitting,

正如我刚才说的 \N如果一台计算机在传输数据期间检测到冲突

it will wait 1 second, plus some random extra time.

会等一秒+随机时间

However, if it collides again, which suggests network congestion,

然而 如果再次发生冲突 表明有网络拥塞

instead of waiting another 1 second, this time it will wait 2 seconds.

这次不等1秒，而是等2秒

If it collides again, it’ll wait 4 seconds, and then 8, and then 16,

如果再次发生冲突 等4秒 然后8秒 16秒等等

and so on, until it’s successful.

直到成功传输

With computers backing off, the rate of collisions goes down,

因为计算机的退避 冲突次数降低了 \N 数据再次开始流动起来 网络变得顺畅

and data starts moving again, freeing up the network.

因为计算机的退避 冲突次数降低了 \N 数据再次开始流动起来 网络变得顺畅

Family dinner saved!

家庭晚餐有救啦！

This "backing off" behavior using an exponentially growing wait time is called

这种指数级增长等待时间的方法叫：

Exponential Backoff.

指数退避

Both Ethernet and WiFi use it, and so do many transmission protocols.

以太网和WiFi都用这种方法 很多其他传输协议也用

But even with clever tricks like Exponential Backoff,

但即便有了"指数退避"这种技巧

you could never have an entire university’s

想用一根网线链接整个大学的计算机还是不可能的

worth of computers on one shared ethernet cable.

想用一根网线链接整个大学的计算机还是不可能的

To reduce collisions and improve efficiency,

为了减少冲突+提升效率

we need to shrink the number of devices on any given shared carrier

我们需要减少同一载体中设备的数量 \N 载体和其中的设备总称 "冲突域"

-- what’s called the Collision Domain.

我们需要减少同一载体中设备的数量 \N 载体和其中的设备总称 "冲突域"

Let go back to our earlier Ethernet example, where we had six computers on one shared cable,

让我们回到之前以太网的例子 一根电缆连6台计算机

a.k.a. one collision domain.

也叫一个冲突域

To reduce the likelihood of collisions, we can break this network

为了减少冲突 我们可以用交换机把它拆成两个冲突域

into two collision domains by using a Network Switch.

为了减少冲突 我们可以用交换机把它拆成两个冲突域

It sits between our two smaller networks, and only passes data between them if necessary.

交换机位于两个更小的网络之间 \N 必要时才在两个网络间传数据

It does this by keeping a list of what MAC addresses are on what side of the network.

交换机会记录一个列表 \N 写着哪个 MAC 地址在哪边网络

So if A wants to transmit to C, the switch doesn’t forward the data to the other network

如果A想传数据给C \N 交换机不会把数据转发给另一边的网络

– there’s no need.

没必要

This means if E wants to transmit to F at the same time, the network is wide open, and

如果E想同一时间传数据给F，网络仍然是空的

two transmissions can happen at once.

两个传输可以同时发生

But, if F wants to send data to A, then the switch passes it through,

但如果F想发数据给A 数据会通过交换机

and the two networks are both briefly occupied.

两个网络都会被短暂占用

This is essentially how big computer networks are constructed,

大的计算机网络也是这样构建的

including the biggest one of all – The Internet –

包括最大的网络 - 互联网

which literally inter-connects a bunch of smaller networks,

也是多个连在一起的稍小一点网络

allowing inter-network communication.

使不同网络间可以传递信息

What’s interesting about these big networks,

这些大型网络有趣之处是

is that there’s often multiple paths to

从一个地点到另一个地点通常有多条路线

get data from one location to another.

从一个地点到另一个地点通常有多条路线

And this brings us to another fundamental networking topic, routing.

这就带出了另一个话题 路由

The simplest way to connect two distant computers, or networks,

连接两台相隔遥远的计算机或网路，最简单的办法 \N 是分配一条专用的通信线路

is by allocating a communication line for their exclusive use.

连接两台相隔遥远的计算机或网路，最简单的办法 \N 是分配一条专用的通信线路

This is how early telephone systems worked.

早期电话系统就是这样运作的

For example, there might be 5 telephone lines running between Indianapolis and Missoula.

假设"印第安纳波利斯"和"米苏拉"之间，有五条电话线

If John picked up the phone wanting to call Hank, in the 1910s,

如果在1910年代，John 想打电话给 Hank

John would tell a human operator where he wanted to call,

John要告诉操作员他想打到什么地方

and they’d physically connect John’s phone line into

然后工作人员手动将 John 的电话连到 \N 通往米苏拉的未使用线路

an unused line running to Missoula.

然后工作人员手动将 John 的电话连到 \N 通往米苏拉的未使用线路

For the length of the call, that line was occupied, and if all 5 lines were already

通话期间 这条线就被占用了 如果五条线都被占用了 \N John 要等待某条线空出来

in use, John would have to wait for one to become free.

通话期间 这条线就被占用了 如果五条线都被占用了 \N John 要等待某条线空出来

This approach is called Circuit Switching,

这叫 "电路交换"\N因为是把电路连接到正确目的地

because you’re literally switching whole

这叫 "电路交换"\N因为是把电路连接到正确目的地

circuits to route traffic to the correct destination.

这叫 "电路交换"\N因为是把电路连接到正确目的地

It works fine, but it’s relatively inflexible and expensive,

能用倒是能用 \N 但不灵活而且价格昂贵 因为总有闲置的线路

because there’s often unused capacity.

能用倒是能用 \N 但不灵活而且价格昂贵 因为总有闲置的线路

On the upside, once you have a line to yourself – or if you have the money to buy one for

好处是 如果有一条专属于自己的线路 \N 你可以最大限度地随意使用，无需共享

your private use – you can use it to its full capacity, without having to share.

好处是 如果有一条专属于自己的线路 \N 你可以最大限度地随意使用，无需共享

For this reason, the military, banks and other high importance operations

因此军队, 银行和其他一些机构

still buy dedicated circuits to connect their data centers.

依然会购买专用线路来连接数据中心

Another approach for getting data from one place to another is Message Switching,

传输数据的另一个方法是 "报文交换"

which is sort of like how the postal system works.

"报文交换" 就像邮政系统一样

Instead of dedicated route from A to B, messages are passed through several stops.

不像之前A和B有一条专有线路 \N 消息会经过好几个站点

So if John writes a letter to Hank,

如果 John 写一封信给 Hank

it might go from Indianapolis to Chicago, and then

信件可能从"印第安纳波利斯"到"芝加哥"

hop to Minneapolis, then Billings, and then finally make it to Missoula.

然后"明尼阿波利斯" 然后"比林斯" 最后到"米苏拉"

Each stop knows where to send it next

每个站点都知道下一站发哪里 \N 因为站点有表格，记录到各个目的地，信件该怎么传

because they keep a table of where to pass letters given a destination address.

每个站点都知道下一站发哪里 \N 因为站点有表格，记录到各个目的地，信件该怎么传

What’s neat about Message Switching is that it can use different routes,

报文交换的好处是 可以用不同路由 \N 使通信更可靠更能容错

making communication more reliable and fault-tolerant.

报文交换的好处是 可以用不同路由 \N 使通信更可靠更能容错

Sticking with our mail example,

回到邮件的例子

if there’s a blizzard in Minneapolis grinding things to a halt,

如果"明尼阿波利斯"有暴风雪中断了通信 \N "芝加哥"可以传给"奥马哈"

the Chicago mail hub can decide to route the letter through Omaha instead.

如果"明尼阿波利斯"有暴风雪中断了通信 \N "芝加哥"可以传给"奥马哈"

In our example, cities are acting like network routers.

在这个例子里，城市就像路由器一样

The number of hops a message takes along a route is called the hop count.

消息沿着路由跳转的次数 \N 叫"跳数"(hop count)

Keeping track of the hop count is useful because it can help identify routing problems.

记录跳数很有用，因为可以分辨出路由问题

For example, let’s say Chicago thinks the fastest route to Missoula is through Omaha,

举例，假设芝加哥认为 \N 去米苏拉的最快路线是 奥马哈

but Omaha thinks the fastest route is through Chicago.

但奥马哈认为 \N 去米苏拉的最快路线是 芝加哥

That's bad, because both cities are going to look at the destination address, Missoula,

这就糟糕了\N 因为2个城市看到目的地是米苏拉

and end up passing the message back and forth between them, endlessly.

结果报文会在2个城市之间\N不停传来传去

Not only is this wasting bandwidth, but it’s a routing error that needs to get fixed!

不仅浪费带宽 而且这个路由错误需要修复!

This kind of error can be detected because the hop count is

这种错误会被检测到，因为跳数记录在消息中 \N 而且传输时会更新跳数

stored with the message and updated along its journey.

这种错误会被检测到，因为跳数记录在消息中 \N 而且传输时会更新跳数

If you start seeing messages with high hop counts,

如果看到某条消息的跳数很高 \N 就知道路由肯定哪里错了

you can bet something has gone awry in the routing!

如果看到某条消息的跳数很高 \N 就知道路由肯定哪里错了

This threshold is the Hop Limit.

这叫"跳数限制"

A downside to Message Switching is that messages are sometimes big.

报文交换的缺点之一是有时候报文比较大

So, they can clog up the network, because the whole message has to be transmitted from

会堵塞网络 因为要把整个报文从一站传到下一站后 \N 才能继续传递其他报文

one stop to the next before continuing on its way.

会堵塞网络 因为要把整个报文从一站传到下一站后 \N 才能继续传递其他报文

While a big file is transferring, that whole link is tied up.

传输一个大文件时 整条路都阻塞了

Even if you have a tiny, one kilobyte email trying to get through,

即便你只有一个1KB的电子邮件要传输 \N 也只能等大文件传完，或是选另一条效率稍低的路线

it either has to wait for the big file transfer to finish or take a less efficient route.

即便你只有一个1KB的电子邮件要传输 \N 也只能等大文件传完，或是选另一条效率稍低的路线

That’s bad.

这就糟了

The solution is to chop up big transmissions into many small pieces, called packets.

解决方法是 将大报文分成很多小块，叫"数据包"

Just like with Message Switching, each packet contains a destination address on the network,

就像报文交换 每个数据包都有目标地址 \N 因此路由器知道发到哪里

so routers know where to forward them.

就像报文交换 每个数据包都有目标地址 \N 因此路由器知道发到哪里

This format is defined by the "Internet Protocol", or IP for short,

报文具体格式由"互联网协议"定义，简称 IP \N 这个标准创建于 1970 年代

a standard created in the 1970s.

报文具体格式由"互联网协议"定义，简称 IP \N 这个标准创建于 1970 年代

Every computer connected to a network gets an IP Address.

每台联网的计算机都需要一个IP地址

You’ve probably seen these as four, 8-bit numbers written with dots in between.

你可能见过，以点分隔的4组数字

For example,172.217.7.238 is an IP Address for one of Google’s servers.

例如 172.217.7.238 是 Google 其中一个服务器的IP地址

With millions of computers online, all exchanging data,

数百万台计算机在网络上不断交换数据 \N 瓶颈的出现和消失是毫秒级的

bottlenecks can appear and disappear in milliseconds.

数百万台计算机在网络上不断交换数据 \N 瓶颈的出现和消失是毫秒级的

Network routers are constantly trying to balance the load across whatever routes they know

路由器会平衡与其他路由器之间的负载 \N 以确保传输可以快速可靠，这叫"阻塞控制"

to ensure speedy and reliable delivery, which is called congestion control.

路由器会平衡与其他路由器之间的负载 \N 以确保传输可以快速可靠，这叫"阻塞控制"

Sometimes different packets from the same message take different routes through a network.

有时，同一个报文的多个数据包 \N 会经过不同线路

This opens the possibility of packets arriving at their destination out of order,

到达顺序可能会不一样，这对一些软件是个问题

which is a problem for some applications.

到达顺序可能会不一样，这对一些软件是个问题

Fortunately, there are protocols that run on top of IP,

幸运的是，在 IP 之上还有其他协议

like TCP/IP, that handle this issue.

比如 TCP/IP, 可以解决乱序问题

We’ll talk more about that next week.

我们下周会讲

Chopping up data into small packets,

将数据拆分成多个小数据包，然后通过灵活的路由传递

and passing these along flexible routes with spare capacity,

将数据拆分成多个小数据包，然后通过灵活的路由传递

is so efficient and fault-tolerant, it’s what the whole internet runs on today.

非常高效且可容错，如今互联网就是这么运行的

This routing approach is called Packet Switching.

这叫"分组交换"

It also has the nice property of being decentralized,

有个好处是 它是去中心化的

with no central authority or single point of failure.

没有中心权威机构 没有单点失败问题

In fact, the threat of nuclear attack is why

事实上 因为冷战期间有核攻击的威胁，所以创造了分组交换

packet switching was developed during the cold war!

事实上 因为冷战期间有核攻击的威胁，所以创造了分组交换

Today, routers all over the globe work cooperatively to find efficient routings,

如今，全球的路由器协同工作，找出最高效的线路

exchanging information with each other using special protocols,

用各种标准协议运输数据

like the Internet Control Message Protocol (ICMP)

比如 "因特网控制消息协议"(ICMP)

and the Border Gateway Protocol (BGP).

和 "边界网关协议"(BGP)

The world's first packet-switched network,

世界上第一个分组交换网络

and the ancestor to the modern internet, was the ARPANET,

以及现代互联网的祖先是 ARPANET

named after the US agency that funded it,

名字来源于赞助这个项目的机构，美国高级研究计划局

the Advanced Research Projects Agency.

名字来源于赞助这个项目的机构，美国高级研究计划局

Here’s what the entire ARPANET looked like in 1974.

这是 1974 年整个 ARPANET 的样子

Each smaller circle is a location,

每个小圆表示一个地点 \N 比如大学或实验室，那里运行着一个路由器

like a university or research lab, that operated a router.

每个小圆表示一个地点 \N 比如大学或实验室，那里运行着一个路由器

They also plugged in one or more computers

并且有一台或多台计算机

– you can see PDP-1’s, IBM System 360s,

能看到 "PDP-1" 和"IBM 360系统"

and even an ATLAS in London connected over a satellite link.

甚至还有一个伦敦的 ATLAS \N 是通过卫星连到网络里的

Obviously the internet has grown by leaps and bounds in the decades since.

显然 互联网在这几十年间发展迅速

Today, instead of a few dozen computers online, it’s estimated to be nearing 10 billion.

如今不再只有几十台计算机联网 \N 据估计 有接近100亿台联网设备

And it continues to grow rapidly,

而且互联网会继续快速发展

especially with the advent of wifi-connected refrigerators, thermostat

特别是如今各种智能设备层出不穷 \N 比如联网冰箱，恒温器

and other smart appliances, forming an "internet of things".

以及其他智能家电，它们组成了"物联网"

So that’s part one – an overview of computer networks.

第一部分到此结束 \N 我们对计算机网络进行了概览

Is it a series of tubes?

网络是一堆管子组成的吗？

Well, sort of.

额 算是吧

Next week we’ll tackle some higher-level transmission protocols,

下周我们会讨论一些高级传输协议

slowly working our way up to the World Wide Web.

然后讲万维网

I’ll see you then!

到时见啦

Hi, I’m Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne \N 欢迎收看计算机科学速成课！

As we talked about last episode, your computer is connected to a large, distributed network,

上集讲到，你的计算机和一个巨大的分布式网络连在一起

called The Internet.

这个网络叫互联网

I know this because you’re watching a YouTube video,

你现在就在网上看视频呀

which is being streamed over that very internet.

你现在就在网上看视频呀

It’s arranged as an ever-enlarging web of interconnected devices.

互联网由无数互联设备组成，而且日益增多

For your computer to get this video,

计算机为了获取这个视频 \N 首先要连到局域网，也叫 LAN

the first connection is to your local area network, or LAN,

计算机为了获取这个视频 \N 首先要连到局域网，也叫 LAN

which might be every device in your house that’s connected to your wifi router.

你家 WIFI 路由器连着的所有设备，组成了局域网.

This then connects to a Wide Area Network, or WAN,

局域网再连到广域网，广域网也叫 WAN

which is likely to be a router run by your Internet Service Provider, or ISP，

WAN 的路由器一般属于你的"互联网服务提供商"，简称 ISP

companies like Comcast, AT&T or Verizon.

比如 Comcast，AT&T 和 Verizon 这样的公司

At first, this will be a regional router, like one for your neighborhood,

广域网里，先连到一个区域性路由器，这路由器可能覆盖一个街区。

and then that router connects to an even bigger WAN,

然后连到一个更大的 WAN，可能覆盖整个城市

maybe one for your whole city or town.

然后连到一个更大的 WAN，可能覆盖整个城市

There might be a couple more hops, but ultimately you’ll connect to the backbone of the internet

可能再跳几次，但最终会到达互联网主干

made up of gigantic routers with super high-bandwidth connections running between them.

互联网主干由一群超大型、带宽超高路由器组成

To request this video file from YouTube,

为了从 YouTube 获得这个视频，

a packet had to work its way up to the backbone,

数据包（packet）要先到互联网主干

travel along that for a bit, and then work its way back down to a YouTube server that had the file.

沿着主干到达有对应视频文件的 YouTube 服务器

That might be four hops up, two hops across the backbone,

数据包从你的计算机跳到 Youtube 服务器，可能要跳个10次，

and four hops down, for a total of ten hops.

先跳4次到互联网主干，2次穿过主干，\N主干出来可能再跳4次，然后到 Youtube 服务器

If you’re running Windows, Mac OS or Linux, you can see the route data takes to different

如果你在用 Windows, Mac OS 或 Linux系统，可以用 traceroute 来看跳了几次

places on the internet by using the traceroute program on your computer.

如果你在用 Windows, Mac OS 或 Linux系统，可以用 traceroute 来看跳了几次

Instructions in the Doobly Doo.

更多详情看视频描述（YouTube原视频下）

For us here at the Chad & Stacey Emigholz Studio in Indianapolis,

我们在"印第安纳波利斯"的 Chad&Stacy Emigholz 工作室，\N 访问加州的 DFTBA 服务器，

the route to the DFTBA server in California goes through 11 stops.

经历了11次中转

We start at 192.168.0.1 -- that's the IP address for my computer on our LAN.

从 192.168.0.1 出发，这是我的电脑在 局域网（LAN）里的 IP 地址

Then there’s the wifi router here at the studio,

然后到工作室的 WIFI 路由器

then a series of regional routers, then we get onto the backbone,

然后穿过一个个地区路由器，到达主干.

and then we start working back down to the computer hosting "DFTBA.com”,

然后从主干出来，又跳了几次，到达"DFTBA.com”的服务器

which has the IP address 104.24.109.186.

IP 地址是 104.24.109.186.

But how does a packet actually get there?

但数据包\*到底\*是怎么过去的？

What happens if a packet gets lost along the way?

如果传输时数据包被弄丢了，会发生什么？

If I type "DFTBA.com” into my web browser, how does it know the server’s address?

如果在浏览器里输 "DFTBA.com"，浏览器怎么知道服务器的地址多少？

These are our topics for today!

我们今天会讨论这些话题.

As we discussed last episode, the internet is a huge distributed network

上集说过，互联网是一个巨型分布式网络 \N 会把数据拆成一个个数据包来传输

that sends data around as little packets.

上集说过，互联网是一个巨型分布式网络 \N 会把数据拆成一个个数据包来传输

If your data is big enough, like an email attachment,

如果要发的数据很大，比如邮件附件 \N 数据会被拆成多个小数据包

it might get broken up into many packets.

如果要发的数据很大，比如邮件附件 \N 数据会被拆成多个小数据包

For example, this video stream is arriving to your computer right now

举例，你现在看的这个视频 \N 就是一个个到达你电脑的数据包

as a series of packets, and not one gigantic file.

而不是一整个大文件发过来

Internet packets have to conform to a standard called the Internet Protocol, or IP.

数据包（packet）想在互联网上传输 \N 要符合"互联网协议"的标准，简称 IP

It’s a lot like sending physical mail through the postal system

就像邮寄手写信一样，邮寄是有标准的\N 每封信需要一个地址，而且地址必须是独特的

– every letter needs a unique and legible address written on it,

就像邮寄手写信一样，邮寄是有标准的\N 每封信需要一个地址，而且地址必须是独特的

and there are limits to the size and weight of packages.

并且大小和重量是有限制的

Violate this, and your letter won’t get through.

违反这些规定，信件就无法送达.

IP packets are very similar.

IP 数据包也是如此

However, IP is a very low level protocol

因为 IP 是一个非常底层的协议

– there isn’t much more than a destination address in a packet’s header

数据包的头部（或者说前面）只有目标地址

which is the metadata that’s stored in front of the data payload.

头部存 "关于数据的数据" \N 也叫 元数据(metadata)

This means that a packet can show up at a computer, but the computer may not know

这意味着当数据包到达对方电脑 \N 对方不知道把包交给哪个程序，是交给 Skype 还是使命召唤？

which application to give the data to; Skype or Call of Duty.

这意味着当数据包到达对方电脑 \N 对方不知道把包交给哪个程序，是交给 Skype 还是使命召唤？

For this reason, more advanced protocols were developed that sit on top of IP.

因此需要在 IP 之上，开发更高级的协议.

One of the simplest and most common is the User Datagram Protocol, or UDP.

这些协议里 \N 最简单最常见的叫"用户数据报协议"，简称 UDP

UDP has its own header, which sits inside the data payload.

UDP 也有头部，这个头部位于数据前面

Inside of the UDP header is some useful, extra information.

头部里包含有用的信息

One of them is a port number.

信息之一是端口号

Every program wanting to access the internet will

每个想访问网络的程序 \N 都要向操作系统申请一个端口号.

ask its host computer’s Operating System to be given a unique port.

每个想访问网络的程序 \N 都要向操作系统申请一个端口号.

Like Skype might ask for port number 3478.

比如 Skype 会申请端口 3478

When a packet arrives to the computer, the Operating System

当一个数据包到达时 \N 接收方的操作系统会读 UDP 头部，读里面的端口号

will look inside the UDP header and read the port number.

当一个数据包到达时 \N 接收方的操作系统会读 UDP 头部，读里面的端口号

Then, if it sees, for example, 3478, it will give the packet to Skype.

如果看到端口号是 3478，就把数据包交给 Skype

So to review, IP gets the packet to the right computer,

总结：\NIP 负责把数据包送到正确的计算机 \N UDP 负责把数据包送到正确的程序

but UDP gets the packet to the right program running on that computer.

总结：\NIP 负责把数据包送到正确的计算机 \N UDP 负责把数据包送到正确的程序

UDP headers also include something called a checksum,

UDP 头部里还有"校验和"，用于检查数据是否正确

which allows the data to be verified for correctness.

UDP 头部里还有"校验和"，用于检查数据是否正确

As the name suggests, it does this by checking the sum of the data.

正如"校验和"这个名字所暗示的 \N 检查方式是把数据求和来对比

Here’s a simplified version of how this works.

以下是个简单例子

Let's imagine the raw data in our UDP packet is

假设 UDP 数据包里 \N 原始数据是 89 111 33 32 58 41

89 111 33 32 58 and 41.

假设 UDP 数据包里 \N 原始数据是 89 111 33 32 58 41

Before the packet is sent, the transmitting computer calculates the checksum

在发送数据包前 \N 电脑会把所有数据加在一起，算出"校验和"

by adding all the data together: 89 plus 111 plus 33 and so on.

89+111+33+... 以此类推

In our example, this adds up to a checksum of 364.

得到 364，这就是"校验和".

In UDP, the checksum value is stored in 16 bits.

UDP 中，\N"校验和"以 16 位形式存储 (就是16个0或1)

If the sum exceeds the maximum possible value, the upper-most bits overflw,

如果算出来的和，超过了 16 位能表示的最大值 \N 高位数会被扔掉，保留低位

and only the lower bits are used.

如果算出来的和，超过了 16 位能表示的最大值 \N 高位数会被扔掉，保留低位

Now, when the receiving computer gets this packet,

当接收方电脑收到这个数据包

it repeats the process, adding up all the data.

它会重复这个步骤 \N 把所有数据加在一起，89+111+33... 以此类推

89 plus 111 plus 33 and so on.

它会重复这个步骤 \N 把所有数据加在一起，89+111+33... 以此类推

If that sum is the same as the checksum sent in the header, all is well.

如果结果和头部中的校验和一致 \N 代表一切正常

But, if the numbers don’t match, you know that the data got corrupted

如果不一致，数据肯定坏掉了

at some point in transit, maybe because of a power fluctuation or faulty cable.

也许传输时碰到了功率波动，或电缆出故障了

Unfortunately, UDP doesn’t offer any mechanisms to fix the data, or request a new copy

不幸的是，UDP 不提供数据修复或数据重发的机制

receiving programs are alerted to the corruption, but typically just discard the packet.

接收方知道数据损坏后，一般只是扔掉.

Also, UDP provides no mechanisms to know if packets are getting through

而且，UDP 无法得知数据包是否到达.

a sending computer shoots the UDP packet off,

发送方发了之后，无法知道数据包是否到达目的地

but has no confirmation it ever gets to its destination successfully.

发送方发了之后，无法知道数据包是否到达目的地

Both of these properties sound pretty catastrophic, but some applications are ok with this,

这些特性听起来很糟糕，但是有些程序不在意这些问题

because UDP is also really simple and fast.

因为 UDP 又简单又快.

Skype, for example, which uses UDP for video chat, can handle corrupt or missing packets.

拿 Skype 举例 \N 它用 UDP 来做视频通话，能处理坏数据或缺失数据

That’s why sometimes if you’re on a bad internet connection,

所以网速慢的时候 Skype 卡卡的 \N 因为只有一部分数据包到了你的电脑

Skype gets all glitchy – only some of the UDP packets are making it to your computer.

所以网速慢的时候 Skype 卡卡的 \N 因为只有一部分数据包到了你的电脑

But this approach doesn’t work for many other types of data transmission.

但对于其他一些数据，这个方法不适用.

Like, it doesn’t really work if you send an email, and it shows up with the middle missing.

比如发邮件，\N 邮件不能只有开头和结尾 没有中间.

The whole message really needs to get there correctly!

邮件要完整到达收件方

When it "absolutely, positively needs to get there”,

如果"所有数据必须到达" \N 就用"传输控制协议"，简称 TCP

programs use the Transmission Control Protocol, or TCP,

如果"所有数据必须到达" \N 就用"传输控制协议"，简称 TCP

which like UDP, rides inside the data payload of IP packets.

TCP 和 UDP 一样，头部也在存数据前面

For this reason, people refer to this combination of protocols as TCP/IP.

因此，人们叫这个组合 TCP/IP

Like UDP, the TCP header contains a destination port and checksum.

就像 UDP ，TCP 头部也有"端口号"和"校验和"

But, it also contains fancier features, and we’ll focus on the key ones.

但 TCP 有更高级的功能，我们这里只介绍重要的几个

First off, TCP packets are given sequential numbers.

1. TCP 数据包有序号

So packet 15 is followed by packet 16, which is followed by 17, and so on...

15号之后是16号，16号之后是17号，以此类推 \N 发上百万个数据包也是有可能的.

for potentially millions of packets sent during that session.

15号之后是16号，16号之后是17号，以此类推 \N 发上百万个数据包也是有可能的.

These sequence numbers allow a receiving computer to put the packets into the correct order,

序号使接收方可以把数据包排成正确顺序，即使到达时间不同.

even if they arrive at different times across the network.

序号使接收方可以把数据包排成正确顺序，即使到达时间不同.

So if an email comes in all scrambled, the TCP implementation in your computer’s operating

哪怕到达顺序是乱的，TCP 协议也能把顺序排对

system will piece it all together correctly.

哪怕到达顺序是乱的，TCP 协议也能把顺序排对

Second, TCP requires that once a computer has correctly received a packet

2. TCP 要求接收方的电脑收到数据包 \N 并且"校验和"检查无误后（数据没有损坏）\N 给发送方发一个确认码，代表收到了

– and the data passes the checksum – that it send back an acknowledgement,

2. TCP 要求接收方的电脑收到数据包 \N 并且"校验和"检查无误后（数据没有损坏）\N 给发送方发一个确认码，代表收到了

or "ACK” as the cool kids say, to the sending computer.

"确认码" 简称 ACK \N 得知上一个数据包成功抵达后，发送方会发下一个数据包

Knowing the packet made it successfully, the sender can now transmit the next packet.

"确认码" 简称 ACK \N 得知上一个数据包成功抵达后，发送方会发下一个数据包

But this time, let’s say, it waits, and doesn’t get an acknowledgement packet back.

假设这次发出去之后，没收到确认码 \N 那么肯定哪里错了

Something must be wrong. If enough time elapses,

如果过了一定时间还没收到确认码 \N 发送方会再发一次

the sender will go ahead and just retransmit the same packet.

如果过了一定时间还没收到确认码 \N 发送方会再发一次

It’s worth noting here that the original packet might have actually gotten there,

注意 数据包可能的确到了

but the acknowledgment is just really delayed.

只是确认码延误了很久，或传输中丢失了

Or perhaps it was the acknowledgment that was lost.

只是确认码延误了很久，或传输中丢失了

Either way, it doesn’t matter, because the receiver has those sequence numbers,

但这不碍事 因为收件方有序列号

and if a duplicate packet arrives, it can be discarded.

如果收到重复的数据包就删掉

Also, TCP isn’t limited to a back and forth conversation – it can send many packets,

还有，TCP 不是只能一个包一个包发

and have many outstanding ACKs, which increases bandwidth significantly, since you aren’t

可以同时发多个数据包，收多个确认码 \N 这大大增加了效率，不用浪费事件等确认码

wasting time waiting for acknowledgment packets to return.

可以同时发多个数据包，收多个确认码 \N 这大大增加了效率，不用浪费事件等确认码

Interestingly, the success rate of ACKs, and also the round trip time

有趣的是，确认码的成功率和来回时间 \N 可以推测网络的拥堵程度

between sending and acknowledging, can be used to infer network congestion.

有趣的是，确认码的成功率和来回时间 \N 可以推测网络的拥堵程度

TCP uses this information to adjust how aggressively it sends packets –

TCP 用这个信息，调整同时发包数量，解决拥堵问题

a mechanism for congestion control.

TCP 用这个信息，调整同时发包数量，解决拥堵问题

So, basically, TCP can handle out-of-order packet delivery, dropped packets

简单说，TCP 可以处理乱序和丢失数据包，丢了就重发.

– including retransmission – and even throttle its transmission rate according to available bandwidth.

还可以根据拥挤情况自动调整传输率

Pretty awesome!

相当厉害！

You might wonder why anyone would use UDP when TCP has all those nifty features.

你可能会奇怪，既然 TCP 那么厉害，还有人用 UDP 吗？

The single biggest downside are all those acknowledgment packets

TCP 最大的缺点是 \N 那些"确认码"数据包把数量翻了一倍

– it doubles the number of messages on the network,

TCP 最大的缺点是 \N 那些"确认码"数据包把数量翻了一倍

and yet, you're not transmitting any more data.

但并没有传输更多信息

That overhead, including associated delays, is sometimes not worth the improved robustness,

有时候这种代价是不值得的 \N 特别是对时间要求很高的程序，比如在线射击游戏

especially for time-critical applications, like Multiplayer First Person Shooters.

有时候这种代价是不值得的 \N 特别是对时间要求很高的程序，比如在线射击游戏

And if it’s you getting lag-fragged you’ll definitely agree!

如果你玩游戏很卡，你也会觉得这样不值!

When your computer wants to make a connection to a website, you need two things

当计算机访问一个网站时 \N 需要两个东西：1.IP地址 2.端口号

- an IP address and a port.

当计算机访问一个网站时 \N 需要两个东西：1.IP地址 2.端口号

Like port 80, at 172.217.7.238.

例如 172.217.7.238 的 80 端口 \N 这是谷歌的 IP 地址和端口号

This example is the IP address and port for the Google web server.

例如 172.217.7.238 的 80 端口 \N 这是谷歌的 IP 地址和端口号

In fact, you can enter this into your browser’s address bar, like so,

事实上，你可以输到浏览器里，然后你会进入谷歌首页

and you’ll end up on the google homepage.

事实上，你可以输到浏览器里，然后你会进入谷歌首页

This gets you to the right destination,

有了这两个东西就能访问正确的网站 \N 但记一长串数字很讨厌

but remembering that long string of digits would be really annoying.

有了这两个东西就能访问正确的网站 \N 但记一长串数字很讨厌

It’s much easier to remember: google.com.

google.com 比一长串数字好记

So the internet has a special service that maps these domain names to addresses.

所以互联网有个特殊服务 \N 负责把域名和 IP 地址一一对应

It’s like the phone book for the internet.

就像专为互联网的电话簿 \N 它叫"域名系统"，简称 DNS

And it’s called the Domain Name System, or DNS for short.

就像专为互联网的电话簿 \N 它叫"域名系统"，简称 DNS

You can probably guess how it works.

它的运作原理你可能猜到了

When you type something like "youtube.com” into your web browser,

在浏览器里输 youtube.com \N 浏览器会去问 DNS 服务器，它的 IP 地址是多少

it goes and asks a DNS server – usually one provided by your ISP – to lookup the address.

一般 DNS 服务器 \N 是互联网供应商提供的

DNS consults its huge registry, and replies with the address... if one exists.

DNS 会查表，如果域名存在，就返回对应 IP 地址.

In fact, if you try mashing your keyboard, adding ".com”, and then hit enter in your

如果你乱敲键盘加个.com 然后按回车

browser, you’ll likely be presented with an error that says DNS failed.

你很可能会看到 DNS 错误

That’s because that site doesn’t exist, so DNS couldn’t give your browser an address.

因为那个网站不存在，所以 DNS 无法返回给你一个地址

But, if DNS returns a valid address, which it should for "YouTube.com”, then your

如果你输的是有效地址，比如 youtube.com \N DNS 按理会返回一个地址

browser shoots off a request over TCP for the website’s data.

然后浏览器会给这个 IP 地址 \N 发 TCP 请求

There’s over 300 million registered domain names, so to make out DNS Lookup a little

如今有三千万个注册域名，所以为了更好管理

more manageable, it’s not stored as one gigantically long list,

DNS 不是存成一个超长超长的列表，而是存成树状结构

but rather in a tree data structure.

DNS 不是存成一个超长超长的列表，而是存成树状结构

What are called Top Level Domains, or TLDs, are at the very top.

顶级域名（简称 TLD）在最顶部，比如 .com 和 .gov

These are huge categories like .com and .gov.

顶级域名（简称 TLD）在最顶部，比如 .com 和 .gov

Then, there are lower level domains that sit below that, called second level domains; Examples

下一层是二级域名，比如 .com 下面有 \N google.com 和 dftba.com

under .com include google.com and dftba.com.

下一层是二级域名，比如 .com 下面有 \N google.com 和 dftba.com

Then, there are even lower level domains, called subdomains,

再下一层叫子域名，\N 比如 images.google.com, store.dftba.com

like images.google.com, store.dftba.com.

再下一层叫子域名，\N 比如 images.google.com, store.dftba.com

And this tree is absolutely HUGE!

这个树超！级！大！

Like I said, more than 300 million domain names, and that's just second level domain

我前面说的"三千万个域名"只是二级域名 \N 不是所有子域名

names, not all the sub domains.

我前面说的"三千万个域名"只是二级域名 \N 不是所有子域名

For this reason, this data is distributed across many DNS servers,

因此，这些数据散布在很多 DNS 服务器上

which are authorities for different parts of the tree.

不同服务器负责树的不同部分

Okay, I know you’ve been waiting for it...

好了 我知道你肯定在等这个梗：

We’ve reached a new level of abstraction!

我们到了一层新抽象！

Over the past two episodes, we’ve worked up from electrical signals on wires,

过去两集里 \N 我们讲了线路里的电信号，以及无线网络里的无线信号

or radio signals transmitted through the air in the case of wireless networks.

过去两集里 \N 我们讲了线路里的电信号，以及无线网络里的无线信号

This is called the Physical Layer.

这些叫"物理层"

MAC addresses, collision detection,

而"数据链路层"负责操控"物理层"，\N 数据链路层有：媒体访问控制地址（MAC），碰撞检测，

exponential backoff and similar low level protocols that

而"数据链路层"负责操控"物理层"，\N 数据链路层有：媒体访问控制地址（MAC），碰撞检测，

mediate access to the physical layer are part of the Data Link Layer.

指数退避，以及其他一些底层协议

Above this is the Network Layer,

再上一层是"网络层"

which is where all the switching and routing technologies that we discussed operate.

负责各种报文交换和路由

And today, we mostly covered the Transport layer, protocols like UDP and TCP,

而今天，我们讲了"传输层"里一大部分， 比如 UDP 和 TCP 这些协议,

which are responsible for point to point data transfer between computers,

负责在计算机之间进行点到点的传输

and also things like error detection and recovery when possible.

而且还会检测和修复错误

We’ve also grazed the Session Layer –

我们还讲了一点点"会话层"

where protocols like TCP and UDP are used to open a connection,

"会话层"会使用 TCP 和 UDP 来创建连接，传递信息，然后关掉连接

pass information back and forth, and then close the connection when finished

"会话层"会使用 TCP 和 UDP 来创建连接，传递信息，然后关掉连接

– what’s called a session.

这一整套叫"会话"

This is exactly what happens when you, for example, do a DNS Lookup, or request a webpage.

查询 DNS 或看网页时，就会发生这一套流程

These are the bottom five layers of the Open System Interconnection (OSI) model,

这是 开放式系统互联通信参考模型(OSI) 的底下5层

a conceptual framework for compartmentalizing all these different network processes.

这个概念性框架 把网络通信划分成多层

Each level has different things to worry about and solve,

每一层处理各自的问题

and it would be impossible to build one huge networking implementation.

如果不分层 \N 直接从上到下捏在一起实现网络通信，是完全不可能的

As we’ve talked about all series, abstraction allows computer scientists and engineers to

抽象使得科学家和工程师能分工同时改进多个层 \N 不被整体复杂度难倒.

be improving all these different levels of the stack simultaneously,

抽象使得科学家和工程师能分工同时改进多个层 \N 不被整体复杂度难倒.

without being overwhelmed by the full complexity.

抽象使得科学家和工程师能分工同时改进多个层 \N 不被整体复杂度难倒.

And amazingly, we’re not quite done yet

而且惊人的是！我们还没讲完呢！

The OSI model has two more layers, the Presentation Layer and the Application Layer,

OSI 模型还有两层，"表示层"和"应用程序层"

which include things like web browsers, Skype,

其中有浏览器，Skype，HTML解码，在线看电影等

HTML decoding, streaming movies and more.

其中有浏览器，Skype，HTML解码，在线看电影等

Which we’ll talk about next week. See you then.

我们下周说，到时见

Hi, I’m Carrie Anne, and welcome to Crash Course Computer Science.

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne \N 欢迎收看计算机科学速成课！

Over the past two episodes, we’ve delved into the wires, signals, switches, packets,

前两集我们深入讨论了电线 信号 交换机 数据包 \N 路由器以及协议，它们共同组成了互联网.

routers and protocols that make up the internet.

前两集我们深入讨论了电线 信号 交换机 数据包 \N 路由器以及协议，它们共同组成了互联网.

Today we’re going to move up yet another level of abstraction

今天我们向上再抽象一层，来讨论万维网

and talk about the World Wide Web.

今天我们向上再抽象一层，来讨论万维网

This is not the same thing as the Internet,

万维网(World Wide Web) \N 和互联网(Internet)不是一回事

even though people often use the two terms interchangeably.

尽管人们经常混用这两个词

The World Wide Web runs on top of the internet,

万维网在互联网之上运行

in the same way that Skype, Minecraft or Instagram do.

互联网之上还有 Skype, Minecraft 和 Instagram

The Internet is the underlying plumbing that conveys the data for all these different applications.

互联网是传递数据的管道，各种程序都会用，

And The World Wide Web is the biggest of them all

其中传输最多数据的程序是万维网

– a huge distributed application running on millions of servers worldwide,

分布在全球数百万个服务器上

accessed using a special program called a web browser.

可以用"浏览器"来访问万维网

We’re going to learn about that, and much more, in today’s episode.

这集我们会深入讲解万维网

The fundamental building block of the World Wide Web – or web for short

万维网的最基本单位，是单个页面

– is a single page.

万维网的最基本单位，是单个页面

This is a document, containing content, which can include links to other pages.

页面有内容，也有去往其他页面的链接 \N 这些链接叫"超链接"

These are called hyperlinks.

页面有内容，也有去往其他页面的链接 \N 这些链接叫"超链接"

You all know what these look like: text or images that you can click,

你们都见过：可以点击的文字或图片，把你送往另一个页面

and they jump you to another page.

你们都见过：可以点击的文字或图片，把你送往另一个页面

These hyperlinks form a huge web of interconnected information,

这些超链接形成巨大的互联网络

which is where the whole thing gets its name.

这就是"万维网"名字的由来

This seems like such an obvious idea.

现在说起来觉得很简单，但在超链接做出来之前

But before hyperlinks were implemented,

现在说起来觉得很简单，但在超链接做出来之前

every time you wanted to switch to another piece of information on a computer,

计算机上每次想看另一个信息时

you had to rummage through the file system to find it, or type it into a search box.

你需要在文件系统中找到它 \N 或是把地址输入搜索框

With hyperlinks, you can easily flow from one related topic to another.

有了超链接，你可以在相关主题间轻松切换

The value of hyperlinked information was conceptualized by Vannevar Bush way back in 1945.

超链接的价值早在 1945 年 \N 就被 Vannevar Bush 意识到了

He published an article describing a hypothetical machine called a Memex,

在第 24 集中我们说过，他发过一篇文章 \N 描述一个假想的机器 Memex

which we discussed in Episode 24.

在第 24 集中我们说过，他发过一篇文章 \N 描述一个假想的机器 Memex

Bush described it as "associative indexing... whereby any item may be caused

Bush的形容是"关联式索引.. 选一个物品会引起

at will to select another immediately and automatically."

另一个物品被立即选中"

He elaborated: "The process of tying two things together is the important thing...

他解释道："将两样东西联系在一起的过程十分重要

thereafter, at any time, when one of those items is in view,

在任何时候，当其中一件东西进入视线

the other [item] can be instantly recalled merely by tapping a button."

只需点一下按钮，立马就能回忆起另一件"

In 1945, computers didn’t even have screens, so this idea was way ahead of its time!

1945年的时候计算机连显示屏都没有，\N 所以这个想法非常超前！

Text containing hyperlinks is so powerful,

因为文字超链接是如此强大

it got an equally awesome name: hypertext!

它得到了一个同样厉害的名字："超文本"！

Web pages are the most common type of hypertext document today.

如今超文本最常指向的，是另一个网页

They’re retrieved and rendered by web browsers

然后网页由浏览器渲染，我们待会会讲

which we'll get to in a few minutes.

然后网页由浏览器渲染，我们待会会讲

In order for pages to link to one another, each hypertext page needs a unique address.

为了使网页能相互连接，每个网页需要一个唯一的地址

On the web, this is specified by a Uniform Resource Locator, or URL for short.

这个地址叫 "统一资源定位器"，简称 URL

An example web page URL is thecrashcourse.com/courses.

一个网页URL的例子是 "thecrashcourse.com/courses"

Like we discussed last episode, when you request a site,

就像上集讨论的，当你访问一个网站时

the first thing your computer does is a DNS lookup.

计算机首先会做"DNS查找"

This takes a domain name as input – like "thecrashcourse.com"

"DNS查找"的输入是一个域名 \N 比如 thecrashcourse.com

and replies back with the corresponding computer’s IP address.

DNS 会输出对应的IP地址

Now, armed with the IP address of the computer you want,

现在有了IP地址 \N 你的浏览器会打开一个 TCP 连接到这个 IP 地址

your web browser opens a TCP connection to a computer

现在有了IP地址 \N 你的浏览器会打开一个 TCP 连接到这个 IP 地址

that’s running a special piece of software called a web server.

这个地址运行着"网络服务器"

The standard port number for web servers is port 80.

网络服务器的标准端口是 80 端口

At this point, all your computer has done is connect to

这时，你的计算机连到了 \N thecrashcourse.com 的服务器

the web server at the address thecrashcourse.com

这时，你的计算机连到了 \N thecrashcourse.com 的服务器

The next step is to ask that web server for the "courses" hypertext page.

下一步是向服务器请求"courses"这个页面

To do this, it uses the aptly named Hypertext Transfer Protocol, or HTTP.

这里会用"超文本传输协议"(HTTP)

The very first documented version of this spec, HTTP 0.9, created in 1991,

HTTP的第一个标准，HTTP 0.9，创建于1991年

only had one command – "GET".

只有一个指令，"GET" 指令

Fortunately, that’s pretty much all you need.

幸运的是，对当时来说也够用

Because we’re trying to get the "courses" page,

因为我们想要的是"courses"页面

we send the server the following command– GET /courses.

我们向服务器发送指令:"GET /courses"

This command is sent as raw ASCII text to the web server,

该指令以"ASCII编码"发送到服务器

which then replies back with the web page hypertext we requested.

服务器会返回该地址对应的网页 \N 然后浏览器会渲染到屏幕上

This is interpreted by your computer's web browser and rendered to your screen.

服务器会返回该地址对应的网页 \N 然后浏览器会渲染到屏幕上

If the user follows a link to another page, the computer just issues another GET request.

如果用户点了另一个链接，计算机会重新发一个GET请求

And this goes on and on as you surf around the website.

你浏览网站时，这个步骤会不断重复

In later versions, HTTP added status codes,

在之后的版本，HTTP添加了状态码

which prefixed any hypertext that was sent following a GET request.

状态码放在请求前面

For example, status code 200 means OK – I’ve got the page and here it is!

举例，状态码 200 代表 "网页找到了,给你"

Status codes in the four hundreds are for client errors.

状态码400~499代表客户端出错

Like, if a user asks the web server for a page that doesn’t exist,

比如网页不存在，就是可怕的404错误

that’s the dreaded 404 error!

比如网页不存在，就是可怕的404错误

Web page hypertext is stored and sent as plain old text,

"超文本"的存储和发送都是以普通文本形式

for example, encoded in ASCII or UTF-16, which we talked about in Episodes 4 and 20.

举个例子，编码可能是 ASCII 或 UTF-16 \N 我们在第4集和第20集讨论过

Because plain text files don’t have a way to specify what’s a link and what’s not,

因为如果只有纯文本 \N 无法表明什么是链接，什么不是链接

it was necessary to develop a way to "mark up" a text file with hypertext elements.

所以有必要开发一种标记方法

For this, the Hypertext Markup Language was developed.

因此开发了 超文本标记语言（HTML)

The very first version of HTML version 0.8, created in 1990,

HTML 第一版的版本号是 0.8，创建于 1990 年

provided 18 HTML commands to markup pages.

有18种HTML指令

That’s it!

仅此而已

Let’s build a webpage with these!

我们来做一个网页吧！

First, let’s give our web page a big heading.

首先，给网页一个大标题

To do this, we type in the letters "h1", which indicates the start of a first level

我们输 h1 代表一级标题，然后用<>括起来

heading, and we surround that in angle brackets.

我们输 h1 代表一级标题，然后用<>括起来

This is one example of an HTML tag.

这就是一个HTML标签

Then, we enter whatever heading text we want.

然后输入想要的标题

We don’t want the whole page to be a heading.

我们不想一整页都是标题 \N 所以加 </h1> 作为结束标签

So, we need to "close" the "h1" tag like so, with a little slash in the front.

我们不想一整页都是标题 \N 所以加 </h1> 作为结束标签

Now lets add some content.

现在来加点内容

Visitors may not know what Klingons are, so let’s make that word a hyperlink to the

读者可能不知道"克林贡"是什么，所以我们给这个词

Klingon Language Institute for more information.

加一个超链接到"克林贡语言研究院"

We do this with an "A" tag, inside of which we include an attribute

我们用 <a> 标签来做，它有一个 href 属性

that specifies a hyperlink reference.

说明链接指向哪里，当点击链接时就会进入那个网页

That’s the page to jump to if the link is clicked.

说明链接指向哪里，当点击链接时就会进入那个网页

And finally, we need to close the A tag.

最后用 </a> 关闭标签

Now lets add a second level heading, which uses an "h2" tag.

接下来用 <h2> 标签做二级标题

HTML also provides tags to create lists.

HTML也有做列表的标签

We start this by adding the tag for an ordered list.

我们先写<ol> \N 代表 有序列表（ordered list）

Then we can add as many items as we want,

然后想加几个列表项目 就加几个 \N 用 <li> 包起来就行

surrounded in "<li>" tags, which stands for list item.

然后想加几个列表项目 就加几个 \N 用 <li> 包起来就行

People may not know what a bat'leth is, so let’s make that a hyperlink too.

读者可能不知道Bat'leth是什么，那么也加上超链接

Lastly, for good form, we need to close the ordered list tag.

最后，为了保持良好格式，用</ol>代表列表结束

And we’re done – that’s a very simple web page!

这就完成了 - 一个很简单的网页！

If you save this text into notepad or textedit, and name it something like "test.html",

如果把这些文字存入记事本或文本编辑器，\N 然后文件取名"test.html"

you should be able to open it by dragging it into your computer’s web browser.

就可以拖入浏览器打开

Of course, today’s web pages are a tad more sophisticated.

当然，如今的网页更复杂一些

The newest version of HTML, version 5, has over a hundred different tags –

最新版的 HTML，HTML5，有100多种标签

for things like images, tables, forms and buttons.

图片标签，表格标签，表单标签，按钮标签，等等

And there are other technologies we’re not going to discuss, like Cascading Style Sheets

还有其他相关技术就不说了\N 比如 层叠样式表 (CSS)

or CSS and JavaScript, which can be embedded into HTML pages and do even fancier things.

和 JavaScript，这俩可以加进网页，做一些更厉害的事

That brings us back to web browsers.

让我们回到浏览器

This is the application on your computer that lets you talk with all these web servers.

网页浏览器可以和网页服务器沟通

Browsers not only request pages and media,

浏览器不仅获取网页和媒体，获取后还负责显示.

but also render the content that’s being returned.

浏览器不仅获取网页和媒体，获取后还负责显示.

The first web browser, and web server,

第一个浏览器和服务器

was written by (now Sir) Tim Berners-Lee over the course of two months in 1990.

是 Tim Berners-Lee 在 1990 年写的，一共花了2个月

At the time, he was working at CERN in Switzerland.

那时候他在瑞士的"欧洲核子研究所"工作

To pull this feat off, he simultaneously created several of the fundamental web standards

为了做出来，他同时建立了几个最基本的网络标准

we discussed today: URL, HTML and HTTP.

URL, HTML 和 HTTP.

Not bad for two months work!

两个月能做这些很不错啊！

Although to be fair, he’d been researching hypertext systems for over a decade.

不过公平点说，他研究超文本系统已经有十几年了

After initially circulating his software amongst colleagues at CERN,

和同事在 CERN 内部使用一阵子后

it was released to the public in 1991.

在 1991 年发布了出去

The World Wide Web was born.

万维网就此诞生

Importantly, the web was an open standard,

重要的是，万维网有开放标准

making it possible for anyone to develop new web servers and browsers.

大家都可以开发新服务器和新浏览器

This allowed a team at the University of Illinois at Urbana-Champaign to

因此"伊利诺伊大学香槟分校"的一个小组

create the Mosaic web browser in 1993.

在 1993 年做了 Mosaic 浏览器

It was the first browser that allowed graphics to be embedded alongside text;

第一个可以在文字旁边显示图片的浏览器

previous browsers displayed graphics in separate windows.

之前浏览器要单开一个新窗口显示图片

It also introduced new features like bookmarks, and had a friendly GUI interface,

还引进了书签等新功能，界面友好，使它很受欢迎

which made it popular.

还引进了书签等新功能，界面友好，使它很受欢迎

Even though it looks pretty crusty, it’s recognizable as the web we know today!

尽管看上去硬邦邦的，但和如今的浏览器长的差不多

By the end of the 1990s, there were many web browsers in use,

1990年代末有许多浏览器面世

like Netscape Navigator, Internet Explorer, Opera, OmniWeb and Mozilla.

Netscape Navigator, Internet Explorer \N Opera, OmniWeb, Mozilla

Many web servers were also developed,

也有很多服务器面世

like Apache and Microsoft’s Internet Information Services (IIS).

比如 Apache 和 微软互联网信息服务(IIS)

New websites popped up daily, and web mainstays

每天都有新网站冒出来，如今的网络巨头

like Amazon and eBay were founded in the mid-1990s.

比如亚马逊和 eBay，创始于 1990 年代中期

It was a golden era!

那是个黄金时代！

The web was flourishing and people increasingly needed ways to find things.

随着万维网日益繁荣，人们越来越需要搜索

If you knew the web address of where you wanted to go –

如果你知道网站地址 \N 比如 ebay.com，直接输入浏览器就行

like ebay.com – you could just type it into the browser.

如果你知道网站地址 \N 比如 ebay.com，直接输入浏览器就行

But what if you didn’t know where to go?

如果不知道地址呢？

Like, you only knew that you wanted pictures of cute cats.

比如想找可爱猫咪的图片

Right now!

现在就要！

Where do you go?

去哪里找呢？

At first, people maintained web pages

起初人们会维护一个目录，链接到其他网站

which served as directories hyperlinking to other websites.

起初人们会维护一个目录，链接到其他网站

"Most famous among these was "Jerry and David's guide to the World Wide Web",

其中最有名的叫"Jerry和David的万维网指南"

renamed Yahoo in 1994.

1994年改名为Yahoo

As the web grew, these human-edited directories started to get unwieldy,

随着网络越来越大，人工编辑的目录变得不便利

and so search engines were developed.

所以开发了搜索引擎

Let’s go to the thought bubble!

让我们进入思想泡泡!

The earliest web search engine that operated like the ones we use today, was JumpStation,

长的最像现代搜索引擎的最早搜素引擎，叫JumpStation

created by Jonathon Fletcher in 1993 at the University of Stirling.

由Jonathon Fletcher于1993年在斯特林大学创建

This consisted of three pieces of software that worked together.

它有 3 个部分

The first was a web crawler, software that followed all the links it could find on the web;

第一个是爬虫，一个跟着链接到处跑的软件

anytime it followed a link to a page that had new links,

每当看到新链接，就加进自己的列表里

it would add those to its list.

每当看到新链接，就加进自己的列表里

The second component was an ever enlarging index,

第二个部分是不断扩张的索引

recording what text terms appeared on what pages the crawler had visited.

记录访问过的网页上，出现过哪些词

The final piece was a search algorithm that consulted the index;

最后一个部分，是查询索引的搜索算法

for example, if I typed the word "cat" into JumpStation,

举个例子，如果我在 JumpStation 输入"猫"

every webpage where the word "cat" appeared would come up in a list.

每个有"猫"这个词的网页都会出现

Early search engines used very simple metrics to rank order their search results, most often

早期搜索引擎的排名方式 非常简单

just the number of times a search term appeared on a page.

取决于 搜索词在页面上的出现次数

This worked okay, until people started gaming the system,

刚开始还行，直到有人开始钻空子

like by writing "cat" hundreds of times on their web pages just to steer traffic their way.

比如在网页上写几百个"猫"，把人们吸引过来

Google’s rise to fame was in large part

谷歌成名的一个很大原因是 \N 创造了一个聪明的算法来规避这个问题

due to a clever algorithm that sidestepped this issue.

谷歌成名的一个很大原因是 \N 创造了一个聪明的算法来规避这个问题

Instead of trusting the content on a web page,

与其信任网页上的内容 \N 搜索引擎会看其他网站 有没有链接到这个网站

they looked at how other websites linked to that page.

与其信任网页上的内容 \N 搜索引擎会看其他网站 有没有链接到这个网站

If it was a spam page with the word cat over and over again, no site would link to it.

如果只是写满"猫"的垃圾网站，没有网站会指向它

But if the webpage was an authority on cats, then other sites would likely link to it.

如果有关于猫的有用内容，有网站会指向它

So the number of what are called "backlinks", especially from reputable sites,

所以这些"反向链接"的数量，特别是有信誉的网站

was often a good sign of quality.

代表了网站质量

This started as a research project called BackRub at Stanford University in 1996, before

Google 一开始时是 1996 年斯坦福大学 \N 一个叫 BackRub 的研究项目

being spun out, two years later, into the Google we know today.

两年后分离出来，演变成如今的谷歌

Thanks thought bubble!

谢谢思想泡泡！

Finally, I want to take a second to talk about a term you’ve probably heard a lot recently,

最后 我想讲一个词，你最近可能经常听到

"Net Neutrality".

网络中立性

Now that you’ve built an understanding of packets, internet routing, and the World Wide

现在你对数据包，路由和万维网，有了个大体概念

Web, you know enough to understand the essence, at least the technical essence, of this big debate.

足够你理解这个争论的核心点，至少从技术角度

In short, network neutrality is the principle that

简单说"网络中立性"是

all packets on the internet should be treated equally.

应该平等对待所有数据包

It doesn’t matter if the packets are my email or you streaming this video,

不论这个数据包是我的邮件，或者是你在看视频

they should all chug along at the same speed and priority.

速度和优先级应该是一样的

But many companies would prefer that their data arrive to you preferentially.

但很多公司会乐意让它们的数据优先到达

Take for example, Comcast, a large ISP that also owns many TV channels,

拿 Comcast 举例，它们不但是大型互联网服务提供商\N 而且拥有多家电视频道

like NBC and The Weather Channel, which are streamed online.

比如 NBC 和 The Weather Channel，可以在线看.

Not to pick on Comcast, but in the absence of Net Neutrality rules,

我不是特意找Comcast麻烦 \N 但要是没有网络中立性

they could for example say that they want their content to be delivered silky smooth, with high priority…

Comcast 可以让自己的内容优先到达 \N 节流其他线上视频

But other streaming videos are going to get throttled,

Comcast 可以让自己的内容优先到达 \N 节流其他线上视频

that is, intentionally given less bandwidth and lower priority.

节流(Throttled) 意思是故意给更少带宽和更低优先级

Again I just want to reiterate here this is just conjecture.

再次重申，这只是举例，不是说 Comcast 很坏

At a high level, Net Neutrality advocates argue that giving internet providers this

支持网络中立性的人说 \N 没有中立性后，服务商可以推出提速的"高级套餐"

ability to essentially set up tolls on the internet – to provide premium packet delivery

支持网络中立性的人说 \N 没有中立性后，服务商可以推出提速的"高级套餐"

plants the seeds for an exploitative business model.

给剥削性商业模式埋下种子

ISPs could be gatekeepers to content, with strong incentives to not play nice with competitors.

互联网服务供应商成为信息的"守门人"，\N 它们有着强烈的动机去碾压对手

Also, if big companies like Netflix and Google can pay to get special treatment,

另外，Netflix和Google这样的大公司可以花钱买特权

small companies, like start-ups, will be at a disadvantage, stifling innovation.

而小公司，比如刚成立的创业公司，\N 会处于劣势，阻止了创新

On the other hand, there are good technical reasons why you might

另一方面，从技术原因看

want different types of data to flow at different speeds.

也许你会希望不同数据传输速度不同

That skype call needs high priority,

你希望Skype的优先级更高，邮件晚几秒没关系

but it’s not a big deal if an email comes in a few seconds late.

你希望Skype的优先级更高，邮件晚几秒没关系

Net-neutrality opponents also argue that market forces and competition would discourage bad

而反对"网络中立性"的人认为，市场竞争会阻碍不良行为

behavior, because customers would leave ISPs that are throttling sites they like.

如果供应商把客户喜欢的网站降速 \N 客户会离开供应商

This debate will rage on for a while yet, and as we always encourage on Crash Course,

这场争辩还会持续很久，\N 就像我们在 Crash Course 其他系列中说过

you should go out and learn more

你应该自己主动了解更多信息

because the implications of Net Neutrality are complex and wide-reaching.

因为"网络中立性"的影响十分复杂而且广泛

I’ll see you next week.

我们下周再见

Hi, I’m Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne \N 欢迎收看计算机科学速成课！

Over the last three episodes, we’ve talked about how computers have become interconnected,

过去3集 我们讲了计算机如何互连

allowing us to communicate near-instantly across the globe.

让我们能瞬时跨全球沟通

But, not everyone who uses these networks is going to play by the rules,

但不是每个使用网络的人都会规规矩矩 不损害他人利益

or have our best interests at heart.

但不是每个使用网络的人都会规规矩矩 不损害他人利益

Just as how we have physical security like locks, fences

就像现实世界中我们用锁和栅栏保证物理安全

and police officers to minimize crime in the real world,

有警察减少犯罪

we need cybersecurity to minimize crime and harm in the virtual world.

我们需要网络安全减少虚拟世界中的犯罪

Computers don’t have ethics.

计算机没有道德观念

Give them a formally specified problem and

只要给计算机写清具体问题 它们很乐意地闪电般算出答案

they’ll happily pump out an answer at lightning speed.

只要给计算机写清具体问题 它们很乐意地闪电般算出答案

Running code that takes down a hospital’s computer systems

破坏医院计算机系统的代码 和 保持病人心跳的代码 \N 对计算机来说没有区别

is no different to a computer than code that keeps a patient's heart beating.

破坏医院计算机系统的代码 和 保持病人心跳的代码 \N 对计算机来说没有区别

Like the Force, computers can be pulled to the light side or the dark side.

就像"原力"一样 \N 计算机可以被拉到"光明面"或"黑暗面"

Cybersecurity is like the Jedi Order, trying to bring peace and justice to the cyber-verse.

网络安全就像 绝地武士团 \N 给网络世界带来和平与正义

The scope of cybersecurity evolves as fast as the capabilities of computing,

计算机安全的范围，和计算能力的发展速度一样快

but we can think of it as a set of techniques to protect the secrecy,

我们可以把计算机安全，看成是保护系统和数据的：

integrity and availability of computer systems and data against threats.

保密性，完整性和可用性

Let’s unpack those three goals:

我们逐个细说：

Secrecy, or confidentiality, means that only authorized people

"保密性"是只有有权限的人 \N 才能读取计算机系统和数据

should be able to access or read specific computer systems and data.

"保密性"是只有有权限的人 \N 才能读取计算机系统和数据

Data breaches, where hackers reveal people’s credit card information,

黑客泄露别人的信用卡信息，就是攻击保密性.

is an attack on secrecy.

黑客泄露别人的信用卡信息，就是攻击保密性.

Integrity means that only authorized people

"完整性"是只有有权限的人 \N 才能使用和修改系统和数据

should have the ability to use or modify systems and data.

"完整性"是只有有权限的人 \N 才能使用和修改系统和数据

Hackers who learn your password and send e-mails masquerading as you, is an integrity attack.

黑客知道你的邮箱密码，假冒你发邮件\N 就是攻击"完整性"

And availability means that authorized people should

"可用性"是有权限的人 \N 应该随时可以访问系统和数据

always have access to their systems and data.

"可用性"是有权限的人 \N 应该随时可以访问系统和数据

Think of Denial of Service Attacks, where hackers overload a website

拒绝服务攻击(DDOS) 就是黑客

with fake requests to make it slow or unreachable for others.

发大量的假请求到服务器，让网站很慢或者挂掉

That’s attacking the service’s availability.

这就是攻击"可用性"

To achieve these three general goals, security experts start with

为了实现这三个目标，安全专家会从 \N 抽象层面想象"敌人"可能是谁，这叫"威胁模型分析"

a specification of who your "enemy" is, at an abstract level, called a threat model.

为了实现这三个目标，安全专家会从 \N 抽象层面想象"敌人"可能是谁，这叫"威胁模型分析"

This profiles attackers: their capabilities, goals, and probable means of attack

模型会对攻击者有个大致描述：\N 能力如何，目标可能是什么，可能用什么手段

– what’s called, awesomely enough, an attack vector.

攻击手段又叫"攻击矢量"

Threat models let you prepare against specific threats, rather than

"威胁模型分析"让你能为特定情境做准备

being overwhelmed by all the ways hackers could get to your systems and data.

不被可能的攻击手段数量所淹没 \N 因为手段实在有太多种了

And there are many, many ways.

不被可能的攻击手段数量所淹没 \N 因为手段实在有太多种了

Let’s say you want to "secure" physical access to your laptop.

假设你想确保笔记本计算机的"物理安全" \N 你的威胁模型是"好管闲事的室友"

Your threat model is a nosy roommate.

假设你想确保笔记本计算机的"物理安全" \N 你的威胁模型是"好管闲事的室友"

To preserve the secrecy, integrity and availability of your laptop,

为了保证保密性，完整性和可用性 \N 你可以藏在脏兮兮的洗衣篮里

you could keep it hidden in your dirty laundry hamper.

为了保证保密性，完整性和可用性 \N 你可以藏在脏兮兮的洗衣篮里

But, if your threat model is a mischievous younger sibling

但如果威胁模型是调皮的兄弟姐妹，知道你喜欢藏哪里

who knows your hiding spots,

但如果威胁模型是调皮的兄弟姐妹，知道你喜欢藏哪里

then you’ll need to do more: maybe lock it in a safe.

那么你需要更多保护：比如锁在保险箱里

In other words, how a system is secured depends heavily on who it’s being secured against.

换句话说，要怎么保护，具体看对抗谁

Of course, threat models are typically a bit more formally defined than just "nosy roommate".

当然，威胁模型通常比"好管闲事的室友"更正式一些

Often you’ll see threat models specified in terms of technical capabilities.

通常威胁模型分析里 会以能力水平区分

For example, "someone who has physical access to your laptop along with unlimited time".

比如"某人可以物理接触到笔记本计算机，而且时间无限"

With a given threat model, security architects need to come up

在给定的威胁模型下，安全架构师要

with a solution that keeps a system secure –

提供解决方案，保持系统安全

as long as certain assumptions are met,

只要某些假设不被推翻

like no one reveals their password to the attacker.

比如没人会告诉攻击者密码

There are many methods for protecting computer systems, networks and data.

保护计算机系统，网络和数据的方法有很多

A lot of security boils down to two questions:

很多安全问题可以总结成2个问题：

who are you, and what should you have access to?

你是谁？你能访问什么？

Clearly, access should be given to the right people, but refused to the wrong people.

权限应该给合适的人，拒绝错误的人

Like, bank employees should be able to open ATMs to restock them, but not me…

比如银行员工可以打开取款机来补充现金。\N 但我不应该有权限打开

because I’d take it all... all of it!

因为我会把钱拿走 全拿走！

That ceramic cat collection doesn’t buy itself!

陶瓷猫收藏品可不会从天上掉下来哟！

So, to differentiate between right and wrong people, we use authentication

所以，为了区分谁是谁，\N 我们用 "身份认证"(authentication)

- the process by which a computer understands who it’s interacting with.

- 让计算机得知使用者是谁

Generally, there are three types, each with their own pros and cons:

身份认证有三种，各有利弊：

What you know.

你知道什么

What you have.

你有什么

And what you are.

你是什么

What you know authentication is based on knowledge of a secret that

"你知道什么" 是基于某个秘密

should be known only by the real user and the computer,

只有用户和计算机知道

for example, a username and password.

比如 用户名和密码

This is the most widely used today because it’s the easiest to implement.

这是如今使用最广泛的，因为最容易实现

But, it can be compromised if hackers guess or otherwise come to know your secret.

但如果黑客通过猜测或其他方式，知道你的密码，就惨了

Some passwords are easy for humans to figure out, like 12356 or qwerty.

有些密码很容易猜中，比如12356或qwerty

But, there are also ones that are easy for computers.

但有些密码对计算机很容易

Consider the PIN: 2580.

比如PIN码：2580

This seems pretty difficult to guess – and it is – for a human.

看起来很难猜中 - 起码对人类来说是这样

But there are only ten thousand possible combinations of 4-digit PINs.

但4位数字，只有一万种可能

A computer can try entering 0000, then try 0001, and then 0002,

一台计算机可以尝试0000，然后0001，然后0002，

all the way up to 9999... in a fraction of a second.

然后到9999，不到一秒内试完

This is called a brute force attack, because it just tries everything.

这叫"暴力攻击"，因为只是试遍一切可能

There’s nothing clever to the algorithm.

这种算法没什么聪明的地方

Some computer systems lock you out, or have you wait a little, after say three wrong attempts.

如果你错误尝试3次，\N 有些系统会阻止你继续尝试，或让你等一会儿

That’s a common and reasonable strategy,

这个策略普遍而且合理

and it does make it harder for less sophisticated attackers.

对于一般的攻击者确实很难

But think about what happens if hackers have already taken over

但假设黑客控制了

tens of thousands of computers, forming a botnet.

数以万计的计算机，形成一个僵尸网络

Using all these computers, the same pin – 2580 –

用这么多计算机尝试密码 2580

can be tried on many tens of thousands of bank accounts simultaneously.

同时尝试很多银行账户

Even with just a single attempt per account, they’ll very likely

即使每个账户只试一次，也很可能

get into one or more that just happen to use that PIN.

碰到某个账户刚好用这个 PIN

In fact, we’ve probably guessed the pin of someone watching this video!

事实上，看视频的某人可能刚好用这个 PIN

Increasing the length of PINs and passwords can help,

增加密码长度有帮助

but even 8 digit PINs are pretty easily cracked.

但即使8位数字的PIN码也很容易破解

This is why so many websites now require you to use a mix of upper and lowercase letters,

这就是为什么现在很多网站 要求大写+小写字母

special symbols, and so on – it explodes the number of possible password combinations.

还有特殊符号等，大大增加可能的密码

An 8-digit numerical PIN only has a hundred million combinations

8位数字的PIN只有一亿种组合

– computers eat that for breakfast!

对计算机轻而易举

But an 8-character password with all those funky things mixed in

但包含各种字符的8位长度密码

has more than 600 trillion combinations.

有超过600万亿种组合

Of course, these passwords are hard for us mere humans to remember,

当然，这些密码会难以记住，

so a better approach is for websites to let us pick something more memorable,

所以更好的方法是 选一些更好记的东西

like three words joined together:

比如三个单词连在一起：

"green brothers rock" or "pizza tasty yum".

"格林兄弟好厉害"或"披萨尝起来好好吃"

English has around 100,000 words in use,

英文大约有10万个单词

so putting three together would give you roughly

所以三个单词连一起大概有

1 quadrillion possible passwords. Good luck trying to guess that!

1亿亿种可能，想猜中的话，祝你好运！

I should also note here that using non-dictionary words

另外使用不在字典内的单词

is even better against more sophisticated kinds of attacks,

被猜中的可能性更低

but we don’t have time to get into that here.

但我们没时间细说这个

Computerphile has a great video on choosing a password - link in the dooblydoo.

Computerphile 频道有个视频讲怎么选择好密码 - \N 链接请看 Youtube 描述

What you have authentication, on the other hand,

"你有什么"这种验证方式

is based on possession of a secret token that only the real user has.

是基于用户有特定物体

An example is a physical key and lock.

比如钥匙和锁

You can only unlock the door if you have the key.

如果你有钥匙，就能开门

This escapes this problem of being "guessable".

这避免了被人"猜中"的问题

And they typically require physical presence,

而且通常需要人在现场

so it’s much harder for remote attackers to gain access.

所以远程攻击就更难了

Someone in another country can’t gain access to your front door in Florida

另一个国家的人，得先来佛罗里达州

without getting to Florida first.

才能到你家前门

But, what you have authentication can be compromised if an attacker is physically close.

但如果攻击者离你比较近，那么也不安全

Keys can be copied, smartphones stolen, and locks picked.

钥匙可以被复制，手机可能被偷，锁可以撬开

Finally, what you are authentication is based on... you!

最后，"你是什么"这种验证，是基于你

You authenticate by presenting yourself to the computer.

把特征展示给计算机进行验证

Biometric authenticators, like fingerprint readers and iris scanners are classic examples.

生物识别验证器，\N 比如指纹识别器和虹膜扫描仪就是典型例子

These can be very secure, but the best technologies are still quite expensive.

这些非常安全，但最好的识别技术仍然很贵

Furthermore, data from sensors varies over time.

而且，来自传感器的数据每次会不同

What you know and what you have authentication have the nice property of being deterministic

"你知道什么"和"你有什么"。这两种验证是"确定性"的

– either correct or incorrect.

- 要么正确，要么错误

If you know the secret, or have the key, you’re granted access 100% of the time.

如果你知道密码，或有钥匙，那么100％能获得访问权限

If you don’t, you get access zero percent of the time.

如果没有，就绝对进不去

Biometric authentication, however, is probabilistic.There’s some chance the system won’t recognize you…

但"生物识别"是概率性的，系统有可能认不出你

maybe you’re wearing a hat or the lighting is bad.

可能你戴了帽子，或者光线不好

Worse, there’s some chance the system will recognize the wrong person as you

更糟的是，系统可能把别人错认成你

– like your evil twin!

比如你的邪恶双胞胎

Of course, in production systems, these chances are low, but not zero.

当然，在现实世界中几率很低，但不是零

Another issue with biometric authentication is it can’t be reset.

生物认证的另一个问题是无法重设

You only have so many fingers, so what happens if an attacker compromises your fingerprint data?

你只有这么多手指，如果攻击者拿到你的指纹数据怎么办

This could be a big problem for life.

你一辈子都麻烦了

And, recently, researchers showed it’s possible to forge your iris

最近还有研究人员表示，拍个照都有可能伪造虹膜

just by capturing a photo of you, so that’s not promising either.

所以也不靠谱

Basically, all forms of authentication have strengths and weaknesses,

所有认证方法都有优缺点，

and all can be compromised in one way or another.

它们都可以被攻破

So, security experts suggest using two or more forms of authentication

所以，对于重要账户，\N 安全专家建议用两种或两种以上的认证方式

for important accounts.

所以，对于重要账户，\N 安全专家建议用两种或两种以上的认证方式

This is known as two-factor or multi-factor authentication.

这叫"双因素"或"多因素"认证

An attacker may be able to guess your password or steal your phone:

攻击者可能猜出你密码，或偷走你的手机：

but it’s much harder to do both.

但两个都做到，会比较难

After authentication comes Access Control.

"身份验证"后，就来到了"访问控制"

Once a system knows who you are, it needs to know what you should be able to access,

一旦系统知道你是谁，它需要知道你能访问什么，

and for that there’s a specification of who should be able to see, modify and use what.

因此应该有个规范，\N 说明谁能访问什么，修改什么，使用什么。

This is done through Permissions or Access Control Lists (ACL),

这可以通过"权限"或"访问控制列表"（ACL）来实现

which describe what access each user has for every file, folder and program on a computer.

其中描述了用户对每个文件，文件夹和程序的访问权限

"Read" permission allows a user to see the contents of a file,

"读"权限允许用户查看文件内容，

"write" permission allows a user to modify the contents,

"写"权限允许用户修改内容，

and "execute" permission allows a user to run a file, like a program.

"执行"权限允许用户运行文件，比如程序

For organizations with users at different levels of access privilege

有些组织需要不同层级的权限

– like a spy agency – it’s especially important for Access Control Lists

比如间谍机构，"访问控制列表"的正确配置非常重要

to be configured correctly to ensure secrecy, integrity and availability.

以确保保密性，完整性和可用性

Let’s say we have three levels of access: public, secret and top secret.

假设我们有三个访问级别：公开，机密，顶级机密

The first general rule of thumb is that people shouldn’t be able to "read up".

第一个普遍的好做法是，\N 用户不能"读上", 不能读等级更高的信息

If a user is only cleared to read secret files, they shouldn’t be able to read top secret

如果用户能读"机密"文件\N 那么不应该有权限读"顶级机密"文件

files, but should be able to access secret and public ones.

但能访问"机密"和"公开"文件

The second general rule of thumb is that people shouldn’t be able to "write down".

第二个法则是用户不能"写下"

If a member has top secret clearance, then they should be able to

如果用户等级是"顶级机密"

write or modify top secret files, but not secret or public files.

那么能写入或修改"顶级机密"文件，\N 但不能修改"机密"或"公共"文件

It may seem weird that even with the highest clearance,

听起来好像很奇怪 \N 有最高等级也不能改等级更低的文件

you can’t modify less secret files.

听起来好像很奇怪 \N 有最高等级也不能改等级更低的文件

But, it guarantees that there’s no accidental leakage of

但这样确保了"顶级机密" \N 不会意外泄露到"机密"文件或"公共"文件里

top secret information into secret or public files.

但这样确保了"顶级机密" \N 不会意外泄露到"机密"文件或"公共"文件里

This "no read up, no write down" approach is called the Bell-LaPadula model.

这个"不能向上读，不能向下写"的方法\N 叫 Bell-LaPadula 模型

It was formulated for the U.S. Department of Defense’s Multi-Level Security policy.

它是为美国国防部"多层安全政策"制定的

There are many other models for access control – like the Chinese Wall model and Biba model.

还有许多其他的访问控制模型 - 比如"中国墙"模型和"比伯"模型

Which model is best depends on your use-case.

哪个模型最好，取决于具体情况

Authentication and access control help a computer determine who you are

"身份验证"和"访问控制"帮助计算机知道"你是谁"

and what you should access,

以及"你可以访问什么"，

but depend on being able to trust the hardware and software

但做这些事情的软硬件必须是可信的

that run the authentication and access control programs.

但做这些事情的软硬件必须是可信的

That’s a big dependence.

这个依赖很重要

If an attacker installs malicious software – called malware

如果攻击者给计算机装了恶意软件

– compromising the host computer’s operating system,

- 控制了计算机的操作系统

how can we be sure security programs don’t have a backdoor that let attackers in?

我们怎么确定安全程序没有给攻击者留后门？

The short answer is… we can’t.

短回答是...无法确定

We still have no way to guarantee the security of a program or computing system.

我们仍然无法保证程序或计算机系统的安全

That’s because even while security software might be "secure" in theory,

因为安全软件在理论上可能是"安全的"

implementation bugs can still result in vulnerabilities.

实现时可能会不小心留下漏洞

But, we do have techniques to reduce the likelihood of bugs,

但我们有办法减少漏洞出现的可能性

like quickly find and patch bugs when they do occur,

比如一找到就马上修复

and mitigate damage when a program is compromised.

以及当程序被攻破时尽可能减少损害

Most security errors come from implementation error.

大部分漏洞都是具体实现的时候出错了

To reduce implementation error, reduce implementation.

为了减少执行错误，减少执行

One of the holy grails of system level security is a "security kernel"

系统级安全的圣杯之一是"安全内核"

or a "trusted computing base": a minimal set of operating system software

或"可信计算基础"：一组尽可能少的操作系统软件

that’s close to provably secure.

安全性都是接近可验证的

A challenge in constructing these security kernels is deciding what should go into it.

构建安全内核的挑战在于 决定内核应该有什么

Remember, the less code, the better!

记住，代码越少越好！

Even after minimizing code bloat, it would be great to "guarantee"

在最小化代码数量之后，\N 要是能"保证"代码是安全的，会非常棒

that’s code is written in secure.

在最小化代码数量之后，\N 要是能"保证"代码是安全的，会非常棒

Formally verifying the security of code is an active area of research.

正式验证代码的安全性 是一个活跃的研究领域

The best we have right now is a process called Independent Verification and Validation.

我们现在最好的手段，叫"独立安全检查和质量验证"

This works by having code audited by a crowd of security-minded developers.

让一群安全行业内的软件开发者来审计代码

This is why security code is almost always open-sourced.

这就是为什么安全型代码几乎都是开源的

It’s often difficult for people who wrote the original code to find bugs,

写原始代码的人通常很难找到错误

but external developers, with fresh eyes and different expertise, can spot problems.

但外部开发人员有新鲜的眼光 \N和不同领域的专业知识，可以发现问题.

There are also conferences where like-minded hackers and security experts

另外还有一些安全大会，安全专家可以相互认识，分享想法.

can mingle and share ideas,

另外还有一些安全大会，安全专家可以相互认识，分享想法.

the biggest of which is DEF CON, held annually in Las Vegas.

一年一次在拉斯维加斯举办的 DEF CON \N 是全球最大的安全大会

Finally, even after reducing code and auditing it,

最后，即便尽可能减少代码 并进行了安全审计

clever attackers are bound to find tricks that let them in.

聪明的攻击者还是会找到方法入侵

With this in mind, good developers should take the approach that,

因为如此，优秀的开发人员

not if, but when their programs are compromised,

应该计划当程序被攻破后，\N如何限制损害，控制损害的最大程度

the damage should be limited and contained,

应该计划当程序被攻破后，\N如何限制损害，控制损害的最大程度

and not let it compromise other things running on the computer.

并且不让它危害到计算机上其他东西

This principle is called isolation.

这叫"隔离"

To achieve isolation, we can "sandbox" applications.

要实现隔离，我们可以"沙盒"程序

This is like placing an angry kid in a sandbox; when the kid goes ballistic,

这好比把生气的小孩放在沙箱里，

they only destroy the sandcastle in their own box,

他们只能摧毁自己的沙堡，不会影响到其他孩子

but other kids in the playground continue having fun.

他们只能摧毁自己的沙堡，不会影响到其他孩子

Operating Systems attempt to sandbox applications

操作系统会把程序放到沙盒里

by giving each their own block of memory that others programs can’t touch.

方法是给每个程序独有的内存块，其他程序不能动

It’s also possible for a single computer to run multiple Virtual Machines, essentially

一台计算机可以运行多个虚拟机

simulated computers, that each live in their own sandbox.

虚拟机模拟计算机，每个虚拟机都在自己的沙箱里

If a program goes awry, worst case is that it crashes or

如果一个程序出错，最糟糕的情况是它自己崩溃

compromises only the virtual machine on which it’s running.

或者搞坏它处于的虚拟机

All other Virtual Machines running on the computer are isolated and unaffected.

计算机上其他虚拟机是隔离的，不受影响

Ok, that’s a broad overview of some key computer security topics.

好，一些重要安全概念的概览 \N 我们到此就介绍完了

And I didn’t even get to network security, like firewalls.

我都还没讲网络安全，比如防火墙

Next episode, we’ll discuss some methods

下集我们会讨论 黑客侵入系统的一些方法

hackers use to get into computer systems.

下集我们会讨论 黑客侵入系统的一些方法

After that, we’ll touch on encryption.

然后我们学加密

Until then, make your passwords stronger, turn on 2-factor authentication,

在此之前，别忘了加强你的密码，打开两步验证

and NEVER click links in unsolicited emails!

永远不要点可疑邮件

I’ll see you next week.

我们下周见

Hi, I’m Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Last episode, we talked about the basics of computer security,

上集我们讲了计算机安全的基础知识，

principles and techniques used to keep computer systems safe and sound.

包括各种原则和技术

But, despite our best efforts, the news is full of stories of individuals, companies

但尽管尽了最大努力，新闻上还是各种 \N 个人，公司，政府被黑客攻击的故事

and governments getting cyberattacked by hackers, people who,

但尽管尽了最大努力，新闻上还是各种 \N 个人，公司，政府被黑客攻击的故事

with their technical knowledge, break into computer systems.

那些黑客凭技术知识 闯入计算机系统

Not all hackers are bad though.

不是所有黑客都是坏人

There are hackers who hunt for bugs and try to close security holes

有些黑客会寻找并修复软件漏洞 \N 让系统更安全

in software to make systems safer and more resilient.

有些黑客会寻找并修复软件漏洞 \N 让系统更安全

They’re often hired by companies and governments to perform security evaluations.

他们经常被公司和政府雇来做安全评估

These hackers are called White Hats, they’re the good guys.

这些黑客叫"白帽子"，他们是好人

On the flip side, there are Black Hats, malicious hackers with

另一方面，也有"黑帽"黑客，他们窃取，

intentions to steal, exploit and sell computer vulnerabilities and data.

利用和销售计算机漏洞和数据

Hackers’ motivations also differ wildly.

黑客的动机有很多种

Some hack for amusement and curiosity,

有些是好玩和好奇

while cybercriminals hack most often for monetary gain.

而网络罪犯一般是为了钱

And then there are hacktivists, who use their skills to promote a social or political goal.

还有的叫"黑客行动主义者"，\N 通过黑客手段影响社会或达到政治目的

And that’s just the tip of the iceberg.

这只是冰山一角

Basically, the stereotypical view of a hacker as some unpopular kid sitting in a dark room

一般对黑客的刻板印象是 \N 某个不受欢迎的小孩在黑暗的房间里

full of discarded pizza boxes probably better describes John Green in college than it does hackers.

到处都是吃完的比萨盒，这个印象是错的，\N形容约翰·格林的宿舍还更贴切些

Today, we’re not going to teach you how to be a hacker.

今天，我们不会教你如何成为黑客

Instead, we’ll discuss some classic examples of how hackers

而是讨论一些入侵原理，给你一个大概概念

break into computer systems to give you an idea of how it’s done.

而是讨论一些入侵原理，给你一个大概概念

The most common way hackers get into computer systems isn’t

黑客入侵最常见的方式

by hacking at all; it’s by tricking users into letting them in.

不是通过技术，而是欺骗别人

This is called social engineering, where a person is manipulated into divulging confidential

这叫"社会工程学"，欺骗别人让人泄密信息

information, or configuring a computer system so that it permits entry by attackers.

或让别人配置电脑系统，变得易于攻击

The most common type of attack is phishing, which you most often encounter as an email

最常见的攻击是网络钓鱼，你可能见过

asking you to login to an account on a website, say your bank.

银行发邮件叫你点邮件里的链接，登陆账号

You’ll be asked to click a link in the email, which takes you to a site that looks legit

然后你会进入一个像官网的网站

to the casual observer, but is really an evil clone.

但实际上是个假网站

When you enter your username and password, that information goes straight to the hackers,

当你输入用户名和密码时，信息会发给黑客，

who then can login to the real website as you.

然后黑客就可以假扮你登陆网站

Bad news!

坏消息！

Even with a 1/10th of one percent success rate, a million phishing emails might yield

即使成功率只有1/1000，发一百万封钓鱼邮件

a thousand compromised accounts.

也有一千个帐户中招

Another social engineering attack is pretexting, where attackers call up, let's say a company,

另一种方法叫 假托(Pretexting)，\N 攻击者给某个公司打电话

and then confidently pretend to be from their IT department.

假装是IT部门的人

Often attackers will call a first number, and then ask to be transferred to a second,

攻击者的第一通电话一般会叫人转接

so that the phone number appears to be internal to the company.

这样另一个人接的时候，电话看起来像内部的

Then, the attacker can instruct an unwitting user to configure their computer in a compromising

然后让别人把电脑配置得容易入侵

way, or get them to reveal confidential details, like passwords or network configurations.

或让他们泄露机密信息，比如密码或网络配置

Sorry, one sec…

不好意思，等一下

Oh. Hey, it's Susan from It.

嘿，我是 IT 部门的苏珊

We’re having some network issues down here, can you go ahead and check a setting for me?"

我们遇到一些网络问题，你能帮我检查一个配置吗？

... and it begins.

然后就开始了

Attackers can be very convincing, especially with a little bit

只要预先做一点研究，攻击者可以装得很像真的

of research beforehand to find things like key employees’ names.

比如关键员工的名字

It might take ten phone calls to find an victim, but you only need one to get in.

也许要10通电话才能找到一个受害者，\N 但只要一个人上当就够了

Emails are also a common delivery mechanism for trojan horses,

邮件里带"木马"也是常见手段

programs that masquerade as harmless attachments, like a photo or invoice,

木马会伪装成无害的东西，比如照片或发票

but actually contain malicious software, called malware.

但实际上是恶意软件

Malware can take many forms.

恶意软件有很多种

Some might steal your data, like your banking credentials.

有的会偷数据，比如银行凭证

Others might encrypt your files and demand a ransom, what's known as ransomware.

有的会加密文件，交赎金才解密，也就是"勒索软件"

If they can’t run malware or get a user to let them in,

如果攻击者无法用木马或电话欺骗

attackers have to force their way in through other means.

攻击者只能被迫用其他手段

One method, which we briefly discussed last episode, is to brute force a password

方法之一是暴力尝试，我们上集讨论过

try every combination of password until you gain entry.

尝试所有可能的密码，直到进入系统

Most modern systems defend against this type of attack by having you wait incrementally

大多数现代系统会加长等待时间，来抵御这种攻击

longer periods of time following each failed attempt,

每次失败就加长等待时间

or even lock you out entirely after a certain number of tries.

甚至失败超过一定次数后，完全锁住

One recent hack to get around this is called NAND Mirroring,

最近出现一种攻破方法叫 "NAND镜像"

where if you have physical access to the computer,

如果能物理接触到电脑

you can attach wires to the device's memory chip

可以往内存上接几根线

and make a perfect copy of its contents.

复制整个内存

With this setup, you can try a series of passwords, until the device starts making you wait.

复制之后，暴力尝试密码，直到设备让你等待

When this happens, you just reflash the memory with the original copy you made,

这时只要把复制的内容覆盖掉内存

essentially resetting it, allowing you to try more passwords immediately, with no waiting.

本质上重置了内存，就不用等待，可以继续尝试密码了

This technique was shown to be successful on an iPhone 5C,

这项方法在 iPhone 5C 上管用

but many newer devices include mechanisms to thwart this type of attack.

更新的设备有机制阻止这种攻击

If you don’t have physical access to a device,

如果你无法物理接触到设备

you have to find a way to hack it remotely, like over the internet.

就必须远程攻击，比如通过互联网.

In general, this requires an attacker to find and take advantage of a bug in a system, and

远程攻击一般需要攻击者利用系统漏洞

successfully utilizing a bug to gain capabilities or access is called an exploit.

来获得某些能力或访问权限，这叫"漏洞利用"(Exploit)

One common type of exploit is a buffer overflow.

一种常见的漏洞利用叫"缓冲区溢出"

Buffers are a general term for a block of memory reserved for storing data.

"缓冲区"是一种概称，指预留的一块内存空间

We talked about video buffers for storing pixel data in Episode 23.

我们在第23集，讨论过存像素数据的视频缓冲区

As a simple example, we can imagine an operating system’s login prompt,

举个简单例子，假设我们在系统登陆界面

which has fields for a username and password.

要输入用户名和密码

Behind the scenes, this operating system uses buffers for storing the text values that are entered.

在幕后，系统用缓冲区存输入的值

For illustration, let's say these buffers were specified to be of size ten.

假设缓冲区大小是10

In memory, the two text buffers would look something like this:

两个文本缓冲区看起来会像这样：

Of course, the operating system is keeping track of a lot more than just a username and

当然，操作系统记录的远不止用户名和密码

password, so there’s going to be data stored both before and after in memory.

所以缓冲区前后 肯定有其他数据

When a user enters a username and password, the values are copied into the buffers,

当用户输入用户名和密码时，这些值会复制到缓冲区

where they can be verified.

然后验证是否正确

A buffer overflow attack does exactly what the name suggests: overflows the buffer.

"缓冲区溢出"正如名字所暗示的：它会溢出缓冲区

In this case, any password longer than ten characters

在这个例子中，超过十个字符的密码

will overwrite adjacent data in memory.

会覆盖掉相邻的数据

Sometimes this will just cause a program or operating system to crash,

有时只会让程序或系统崩溃，

because important values are overwritten with gobbledygook.

因为重要值被垃圾数据覆盖了

Crashing a system is bad, and maybe that’s all that

系统崩溃是坏事

a mischievous hacker wants to do, be a nuisance.

但也许恶作剧黑客就只是想系统崩溃，当个讨厌鬼

But attackers can also exploit this bug more cleverly by injecting purposeful new values

但攻击者可以更巧妙地利用这个漏洞(bug)，\N 注入有意义的新值

into a program’s memory, for example, setting an "is admin" variable to true.

到程序的内存中，比如把"is\_admin"的值改成true

With the ability to arbitrarily manipulate a program’s memory,

有了任意修改内存的能力，

hackers can bypass things like login prompts,

黑客可以绕过"登录"之类的东西，

and sometimes even use that program to hijack the whole system.

甚至使用那个程序劫持整个系统

There are many methods to combat buffer overflow attacks.

有很多方法阻止缓冲区溢出

The easiest is to always test the length of input before copying it into a buffer,

最简单的方法是，复制之前先检查长度，

called bounds checking.

这叫 "边界检查"

Many modern programming languages implement bounds checking automatically.

许多现代编程语言自带了边界检查

Programs can also randomize the memory location of variables,

程序也会随机存放变量在内存中的位置，

like our hypothetical "is admin" flag,

比如我们之前假设的"is\_admin"

so that hackers don’t know what memory location to overwrite,

这样黑客就不知道应该覆盖内存的哪里

and are more likely to crash the program than gain access.

导致更容易让程序崩溃，而不是获得访问权限

Programs can also leave unused space after buffers,

程序也可以在缓冲区后，留一些不用的空间

and keep an eye on those values to see if they change;

然后跟踪里面的值，看是否发生变化

if they do, they know an attacker is monkeying around with memory.

如果发生了变化，说明有攻击者在乱来

These regions are called canaries, named after the small birds miners

这些不用的内存空间叫"金丝雀"，因为以前矿工会带

used to take underground to warn them of dangerous conditions.

金丝雀下矿，金丝雀会警告危险

Another classic hack is code injection.

另一种经典手段叫"代码注入"

It’s most commonly used to attack websites that use databases,

最常用于攻击用数据库的网站，

which pretty much all big websites do.

几乎所有大网站都用数据库

We won’t be covering databases in this series,

我们这个系列中不会讲解数据库，

so here’s a simple example to illustrate this type of attack.

所以以下是个简单例子

We’ll use Structured Query Language, S-Q-L, also called sequel, a popular database API.

我们会用"结构化查询语言"，也叫SQL，\N 一种流行的数据库API

Let’s imagine our login prompt is now running on a webpage.

假设网页上有登录提示

When a user clicks "login", the text values are sent to a server, which executes code

当用户点击"登录"时，值会发到服务器

that checks if that username exists, and if it does, verifies the password matches.

服务器会运行代码，检查用户名是否存在，\N 如果存在，看密码是否匹配

To do this, the server will execute code, known as a SQL query,

为了做检查，服务器会执行一段叫 "SQL查询" 的代码

that looks something like this.

看起来像这样

First, it needs to specify what data we’re retrieving from the database.

首先，语句要指定从数据库里查什么数据

In this case, we want to fetch the password.

在这个例子中，我们想查的是密码 (password) \N (SELECT password)

The server also needs to specify from what place in the database

还要指定从哪张表查数据 \N (FROM users)

to retrieve the value from.

还要指定从哪张表查数据 \N (FROM users)

In this case, let’s imagine all the users’ data is stored

在这个例子里，\N 我们假设所有用户数据都存在 "users" 表里

in a data structure called a table labeled "users".

在这个例子里，\N 我们假设所有用户数据都存在 "users" 表里

Finally, the server doesn’t want to get back a giant list of passwords for every user

最后，服务器不想每次取出一个巨大密码列表，\N 包含所有用户密码

in the database, so it specifies that it only wants data for the account

所以用 username = '用户名'\N 代表只要这个用户

whose username equals a certain value.

所以用 username = '用户名'\N 代表只要这个用户

That value is copied into the SQL query by the server, based on what the user typed in,

用户输的值会复制到"SQL查询"

so the actual command that’s sent to the SQL database would look something like this,

所以实际发到 SQL 数据库的命令，是这样的.

where username='philbin'

Where username='philbin'

Note also that SQL commands end with a semicolon.

还要注意，SQL命令以分号结尾

So how does someone hack this?

那怎么破解这个？

By sending in a malicious username, with embedded SQL commands!

做法是把"SQL命令"输入到用户名里！

Like, we could send the server this funky username:

比如我们可以发这个奇怪的用户名：

When the server copies this text into the SQL Query, it ends up looking like this:

当服务器把值复制到SQL查询中，会变成这样：

As I mentioned before, semicolons are used to separate commands,

正如之前提的，分号用于分隔命令，

so the first command that gets executed is this:

所以第一条被执行的命令是：

If there is a user named ‘whatever’, the database will return the password.

如果有个用户叫"whateer"，数据库将返回密码

Of course, we have no idea what ‘whatever’s’ password is,

当然，我们不知道密码是什么

so we’ll get it wrong and the server will reject us.

所以会出错，服务器会拒绝我们

If there’s no user named ‘whatever’, the database will return

如果没有一个用户叫"whatever"，数据库会返回 \N 空密码或直接错误，服务器也会拒绝我们

no password or provide an error, and the server will again reject us.

如果没有一个用户叫"whatever"，数据库会返回 \N 空密码或直接错误，服务器也会拒绝我们

Either way, we don’t care, because it’s the next SQL command we’re interested in:

总之 我们不在乎，\N 我们感兴趣的是下一个SQL命令：

"drop table users" – a command that we injected by manipulating the username field.

"drop table users" - 我们注入的命令

This command instructs the SQL database to delete the table containing all user data.

这条命令的意思是 删掉 users 这张表

Wiped clean!

全删干净！

Which would cause a lot of headaches at a place like a bank... or really anywhere.

这会造成很多麻烦，不管是银行或什么其他地方

And notice that we didn’t even break into the system –

注意，我们甚至不需要侵入系统

it’s not like we correctly guessed a username and password.

我们没有猜到正确的用户名和密码

Even with no formal access, we were able to create mayhem by exploiting a bug.

即使没有正式访问权限，还是可以利用 bug 来制造混乱

This is a very simple example of code injection,

这是代码注入的一个简单例子，

which almost all servers today have defenses against.

如今几乎所有服务器都会防御这种手段

With more sophisticated attacks, it’s possible to add records to the database

如果指令更复杂一些，也许可以添加新记录到数据库

– like a new administrator account –

- 比如一个新管理员帐户 -

or even get the database to reveal data, allowing hackers

甚至可以让数据库泄露数据，使得黑客

to steal things like credit card numbers, social security numbers

窃取信用卡号码，社会安全号码

and all sorts of nefarious goodies.

以及各种其他信息

But we’re not going to teach you how to do that.

但我们不会教你具体怎么做

As with buffer overflows, programmers should always assume input coming from the outside

就像缓冲区溢出攻击一样，应该总是假设外部数据

to be potentially dangerous, and examine it carefully.

是危险的，应该好好检查

Most username and password forms on the web don’t let you

很多用户名和密码表单，不让你输入

include special symbols like semicolons or quotes as a first level of defense.

特殊字符，比如分号或者括号，作为第一道防御

Good servers also sanitize input by removing or

好的服务器也会清理输入

modifying special characters before running database queries.

比如修改或删除特殊字符，然后才放到数据库查询语句里

Working exploits are often sold or shared online.

管用的漏洞利用(Exploits)一般会在网上贩卖或分享

The more prevalent the bug, or the more damaging the exploit,

如果漏洞很流行，或造成的危害很大

the higher the price or prestige it commands.

价格会越高，或者名气越大

Even governments sometimes buy exploits,

有时甚至政府也会买漏洞利用

which allow them to compromise computers for purposes like spying.

让他们侵入系统做间谍工作

When a new exploitable bug is discovered that the software creators weren’t aware of,

当软件制造者不知道软件有新漏洞被发现了

it’s called a zero day vulnerability.

那么这个漏洞叫 "零日漏洞"

Black Hat Hackers rush to use the exploit for maximum benefit

黑帽黑客经常赶时间，抢在白帽程序员做出补丁之前

before white hat programmers release a patch for the bug.

尽可能利用漏洞

This is why it’s so important to keep your computer’s software up to date;

所以保持系统更新非常重要

a lot of those downloads are security patches.

很多更新都是安全性补丁

If bugs are left open on enough systems, it allows hackers to

如果有足够多的电脑有漏洞

write a program that jump from computer to computer automatically

让恶意程序可以在电脑间互相传播

which are called worms.

那么叫"蠕虫"

If a hacker can take over a large number of computers, they can be used together,

如果黑客拿下大量电脑，这些电脑可以组成

to form what’s called a botnet.

"僵尸网络"

This can have many purposes, like sending huge volumes of spam,

可以用于很多目的，比如发大量垃圾邮件，

mining bitcoins using other people's computing power and electricity,

用别人电脑的计算能力和电费挖 Bitcoin，

and launching Distributed Denial of Service or DDoS attacks against servers.

或发起"拒绝服务攻击"简称DDoS，攻击服务器

DDoS is where all the computers in the botnet send a flood of dummy messages.

DDoS 就是僵尸网络里的所有电脑发一大堆垃圾信息

This can knock services offline, either to force owners

堵塞服务器，要么迫使别人交钱消灾

to pay a ransom or just to be evil.

或纯粹为了作恶

Despite all of the hard working white hats, exploits documented online,

尽管白帽黑客非常努力工作，漏洞利用的文档都在网上，

and software engineering best practices, cyberattacks happen on a daily basis.

编写软件有很多"最佳实践"，网络攻击每天都在发生

They cost the global economy roughly half a trillion dollars annually,

每年损害全球经济差不多5000亿

and that figure will only increase as we become more reliant on computing systems.

并且随着我们越来越依赖计算机系统，这个数字只会增加.

This is especially worrying to governments, as infrastructure is increasingly computer-driven,

这使得政府非常担心，因为基础设施越来越电脑化

like powerplants, the electrical grid, traffic lights, water treatment plants, oil refineries,

比如电力厂，电网，交通灯，水处理厂，炼油厂

air traffic control, and lots of other key systems.

空管，还有很多其他关键系统

Many experts predict that the next major war will be fought in cyberspace,

很多专家预测下一次大战会主要是网络战争

where nations are brought to their knees not by physical attack,

国家不是被物理攻击打败

but rather crippled economically and infrastructurally through cyberwarfare.

而是因为网络战争导致经济和基础设施崩溃

There may not be any bullets fired, but the potential for lives lost is still very high...

也许不会发射一颗子弹，但是人员伤亡的可能性依然很高

maybe even higher than conventional warfare.

甚至可能高于传统战争

So, we should all adopt good cybersecurity practices.

所以大家都应该知道一些方法保证网络安全

And, as a community interconnected over the internet,

全球社区因为互联网而互相连接，

we should ensure our computers are secured against those

我们应该确保自己的电脑安全

who wish to use their great potential for harm.

抵御其他想做坏事的人

So maybe stop ignoring that update notification?

也许不要再忽略更新提示？

I’ll see you next week.

我们下周见

Hi, I'm Carrie Anne, and welcome to CrashCourse Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

Over the past two episodes, we’ve talked a lot about computer security.

在过去两集，我们聊了很多计算机安全话题

But the fact is, there’s no such thing as a perfectly, 100% secure, computer system.

但事实是 世上不存在100%安全的系统

There will always be bugs and security experts know that.

总会有漏洞存在，而且安全专家知道这一点

So system architects employ a strategy called defence in depth

所以系统架构师会部署"多层防御"

which uses many layers of varying security mechanisms to frustrate attackers.

用多层不同的安全机制来阻碍攻击者

It’s a bit like how castles are designed

有点像城堡的设计一样

first you’ve got to dodge the archers

首先要避开弓箭手

then cross the moat, scale the walls, avoid the hot oil, get over the ramparts, and defeat the guards

穿过护城河,翻过城墙,避开热油,打败守卫

before you get to the throne room

才能达到王座

but in this case we’re talking about one of the most common forms of computer security

不过我们这里要说的是，计算机安全中最常见的防御形式

Cryptography

密码学

The word cryptography comes from the roots ‘crypto’ and ‘graphy’, roughly translating to "secret writing".

密码学(cryptography) 一词 \N 来自 crypto 和 graphy，大致翻译成"秘密写作"

In order to make information secret, you use a cipher – an algorithm that converts plain text into ciphertext

为了加密信息，要用加密算法(Cipher) 把明文转为密文

which is gibberish unless you have a key that lets you undo the cipher.

除非你知道如何解密，不然密文看起来只是一堆乱码

The process of making text secret is called encryption

把明文转成密文叫"加密"(encryption)

and the reverse process is called decryption

把密文恢复回明文叫"解密"(decryption)

Ciphers have been used long before computers showed up.

加密算法早在计算机出现前就有了

Julius Caesar used what’s now called a Caesar cipher, to encrypt private correspondence.

朱利叶斯·凯撒 用如今我们叫"凯撒加密"的方法 来加密私人信件

He would shift the letters in a message forward by three places.

他会把信件中的字母 向前移动三个位置

So, A became D, and the word "brutus" became this: "euxwxv".

所以A会变成D，brutus变成euxwxv

To decipher the message, recipients had to know both the algorithm and the number to shift by, which acted as the key.

为了解密，接收者要知道 \N 1. 用了什么算法 2. 要偏移的字母位数

The Caesar cipher is one example of a larger class of techniques called substitution ciphers.

有一大类算法叫"替换加密"，凯撒密码是其中一种

These replace every letter in a message with\Nsomething else according to a translation.

算法把每个字母替换成其他字母

A big drawback of basic substitution ciphers is that letter frequencies are preserved.

但有个巨大的缺点是，字母的出现频率是一样的

For example, E is the most common letter in English

举个例子，E在英语中是最常见的字母

so if your cipher translates E to an X

如果把E加密成X

then X will show up the most frequently in the ciphertext.

那么密文中 X 的出现频率会很高

A skilled cryptanalyst can work backwards from these kinds of statistics to figure out the message.

熟练的密码破译师可以从统计数据中发现规律，进而破译密码

Indeed, it was the breaking of a substitution cipher that led to the execution of Mary Queen of Scots,in 1587 for plotting to kill Queen Elizabeth.

1587年，正因为一个"替换加密"的密文被破译，\n导致杀伊丽莎白女王的阴谋暴露，使得玛丽女王被处决

Another fundamental class of techniques are permutation ciphers.

另一类加密算法叫 "移位加密"

Let’s look at a simple example, called a columnar transposition cipher.

我们来看一个简单例子叫 "列移位加密"

Here, we take a message, and fill the letters into a grid.

我们把明文填入网格

In this case, we’ve chosen 5 by 5

网格大小我们这里选择 5x5

To encrypt our message, we read out the characters in a different order

为了加密信息，我们换个顺序来读

let’s say from the bottom left, working upwards, one column at a time.

比如从左边开始，从下往上，一次一列。

The new letter ordering, what’s called a permutation, is the encrypted message.

加密后字母的排列不同了

The ordering direction, as well as the 5 by 5 grid size, serves as the key.

解密的关键是，知道读取方向和网格大小是5x5

Like before, if the cipher and key are known, a recipient can reverse the process to reveal the original message.

就像之前，如果接收者知道密文和加密方法 \N 才能解密得到原始消息

By the 1900s, cryptography was mechanized in the form of encryption machines.

到了1900年代，人们用密码学做了加密机器

The most famous was the German Enigma, used by the Nazis to encrypt their wartime communications.

其中最有名的是德国的英格玛（Enigma）\N 纳粹在战时用英格玛加密通讯信息

As we discussed back in Episode 15, the Enigma was a typewriter-like machine, with a keyboard and lampboard, both showing the full alphabet.

正如第15集中说过，Enigma 是一台像打字机的机器，\N 有键盘和灯板，两者都有完整的字母表

Above that, there was a series of configurable rotors that were the key to the Enigma’s encryption capability.

而且它有一系列"转子"(rotros) ，是加密的关键

First, let’s look at just one rotor.

首先，我们只看一个转子

One side had electrical contacts for all 26 letters.

它一面有26个接触点，代表26个字母

These connected to the other side of the rotor using cross-crossing wires that swapped one letter for another.

然后线会连到另一面，替换字母

If ‘H’ went in, ‘K’ might come out the other side.

如果输入'H'，'K'会从另一边出来

If "K’ went in, ‘F’ might come out, and so on.

如果输入'K'，'F'会从另一边出来，以此类推

This letter swapping behavior should sound familiar: it’s a substitution cipher!

这个字母替换过程你应该听起来很熟悉：它是"替换加密"!

But, the Enigma was more sophisticated becauseit used three or more rotors in a row, each feeding into the next.

但英格玛(Enigma)更复杂一些，因为它有3个或更多转子，\N 一个转子的输出作为下一个转子的输入。

Rotors could also be rotated to one of 26 possible starting positions

转子还有26个起始位置

and they could be inserted in different orders, providinga lot of different substitution mappings.

还可以按不同顺序放入转子，提供更多字母替换映射

Following the rotors was a special circuit called a reflector.

转子之后是一个叫"反射器"的特殊电路

Instead of passing the signal on to another rotor, it connected every pin to another,

它每个引脚会连到另一个引脚

and sent the electrical signal back through the rotors.

并把信号发回给转子

Finally, there was a plug board at the front of the machine

最后，机器前方有一个插板

that allowed letters coming from the keyboard to be optionally swapped,

可以把输入键盘的字母预先进行替换

adding another level of complexity.

又加了一层复杂度

With our simplified circuit, let’s encrypta letter on this example enigma configuration.

让我们用这里的简化版电路，加密一些字母

If we press the ‘H’ key, electricity flows through the plugboard, then the rotors

如果我们按下"H"键，电流会先通过插板，然后通过转子

hits the reflector, comes back through the rotorsand plugboard, and illuminates the letter ‘L’ on the lampboard.

到达反射器，然后回来转子，回来插板，并照亮键盘灯板的字母"L"。

So H is encrypted to L.

H 就加密成了 L

Note that the circuit can flow both ways –

注意, 电路是双向的

so if we typed the letter ‘L’, ‘H’ would light up.

所以如果我们按下 L，H 会亮起来

In other words, it’s the same process for encrypting and decrypting;

换句话说，加密和解密的步骤是一样的

you just have to make sure the sending and receiving machineshave the same initial configuration.

你只需要确保 发送机和接收机的初始配置一样就行

If you look carefully at this circuit, you’ll notice it’s impossible for a letter to be encrypted as itself

如果你有仔细观察，会注意到字母加密后一定会变成另一个字母

which turned out to be a fatal cryptographic weakness.

之后这成为最大的弱点

Finally, to prevent the Enigma from being a simple substitution cipher

最后，为了让英格玛不只是简单的"替换加密"

every single time a letter was entered, the rotors advanced by one spot, sort of like an odometer in a car.

每输入一个字母，转子会转一格，有点像汽车里程表。

So if you entered the text A-A-A, it might come out as B-D-K, where the substitution mapping changed with every key press.

如果你输入A-A-A，可能会变成B-D-K，\n映射会随着每次按键而改变

The Enigma was a tough cookie to crack, for sure.

英格玛当然是一块难啃的骨头

But as we discussed in Episode 15, Alan Turingand and his colleagues

但正如我们第15集中说的，艾伦·图灵和同事

at Bletchley Park were able to break Enigma codes and largely automate the process.

破解了英格玛加密，并把大部分破解流程做成了自动化

But with the advent of computers, cryptography moved from hardware into software.

但随着计算机出现，加密从硬件转往软件

One of the earliest software ciphers to become widespread

早期加密算法中，应用最广泛的

was the Data Encryption Standard developed by IBM and the NSA in 1977

是 IBM 和 NSA 于1977年开发的"数据加密标准"

DES, as it was known, originally used binary keys that were 56 bits long,

DES最初用的是56 bit长度的二进制密钥，

which means that there are 2 to the 56, or about 72 quadrillion different keys.

意味着有2的56次方，或大约72千万亿个不同密钥

Back in 1977, that meant that nobody – except perhaps the NSA –

在1977年时，也许 NSA 有这能力，

had enough computing power to brute-force all possible keys.

但没有其他人有足够计算能力 来暴力破解所有可能密钥。

But, by 1999, a quarter-million dollar computer could try every possible DES key in just two days, rendering the cipher insecure.

但到1999年，一台25万美元的计算机能在两天内 \N 把 DES 的所有可能密钥都试一遍，让 DES 算法不再安全

So, in 2001, the Advanced Encryption Standard(AES) was finalized and published.

因此 2001 年出了：高级加密标准（AES）

AES is designed to use much bigger keys – 128,192 or 256 bits in size – making brute force attacks much, much harder.

AES 用更长的密钥 - 128位/192位/256位 - 让暴力破解更加困难

For a 128-bit keys, you'd need trillions of years to try every combination, even if you used every single computer on the planet today.

128位的密钥，哪怕用现在地球上的所有计算机\N 也要上万亿年才能试遍所有组合

So you better get started!

你最好赶紧开始!

AES chops data up into 16-byte blocks, and then applies a series of substitutions and permutations,

AES将数据切成一块一块，每块16个字节，\N 然后用密钥进行一系列替换加密和移位加密

based on the key value plus some other operations to obscure the message,

再加上一些其他操作，进一步加密信息

and this process is repeated ten or more times for each block.

每一块数据，会重复这个过程10次或以上

You might be wondering: why only ten rounds?

你可能想知道：为什么只重复10次？

Or why only 128 bit keys, and not ten thousand bit keys?

为什么用128位密钥，而不是10000位？

Well, it’s a performance tradeoff.

这其实是基于性能的权衡

If it took hours to encrypt and send an email,or minutes to connect to a secure website, people wouldn't use it

如果要花几小时加密和发邮件，或几分钟载入网站，没人愿意用

AES balances performance and security to provide practical cryptography.

AES 在性能和安全性间取得平衡

Today, AES is used everywhere, from encrypting files on iPhones

如今AES被广泛使用，比如iPhone上加密文件

and transmitting data over WiFi with WPA2 to accessing websites using HTTPS.

用 WPA2 协议在 WiFi 中访问 HTTPS 网站

So far, the cryptographic techniques we’ve discussed rely on keys that are known by both sender and recipient.

到目前为止 我们讨论过的加密技术\N 依赖于发送者和接收者都知道密钥

The sender encrypts a message using a key, and the recipient decrypts it using the same key.

发件人用密钥加密，收件人用相同的密钥解密

In the old days, keys would be shared by voice, or physically;

以前，密钥可以口头约定，或依靠物品

for example, the Germans distributed codebooks with daily settings for their Enigma machines.

比如德国人给英格玛配了密码本，上面有每天的配置

But this strategy could never work in the internet era.

但互联网时代没法这样做

Imagine having to crack open a codebook to connect to youtube

你能想象 要打开密码本才能访问 YouTube 吗？

What’s needed is a way for a server to send a secret key over the public internet to a user wishing to connect securely.

我们需要某种方法 在公开的互联网上传递密钥给对方

It seems like that wouldn’t be secure, because if the key is sent in the open and intercepted by a hacker

这看起来好像不安全，如果密钥被黑客拦截了

couldn’t they use that to decrypt all communication between the two?

黑客不就能解密通信了吗？

The solution is key exchange!

解决方案是 "密钥交换"！

– An algorithm that lets two computers agreeon a key without ever sending one.

密钥交换是一种不发送密钥，\N 但依然让两台计算机在密钥上达成共识的算法

We can do this with one-way functions –

我们可以用"单向函数"来做

mathematical operations that are very easy to do in one direction, but hard to reverse.

单项函数是一种数学操作，很容易算出结果，\N 但想从结果逆向推算出输入非常困难

To show you how one-way functions work, let’ s use paint colors as an analogy.

为了让你明白单项函数，我们拿颜色作比喻

It’s easy to mix paint colors together, but it’s not so easy to figure

将颜色混合在一起很容易，

out the constituent colors that were used to make a mixed paint color.

但想知道混了什么颜色很难

You’d have to test a lot of possibilities to figure it out.

要试很多种可能才知道

In this metaphor, our secret key is a unique shade of paint.

用这个比喻，那么我们的密钥是一种独特的颜色

First, there’s a public paint color that everyone can see.

首先，有一个公开的颜色，所有人都可以看到

Then, John and I each pick a secret paint color.

然后，约翰和我各自选一个秘密颜色，只有自己知道.

To exchange keys, I mix my secret paint color with the public paint color.

为了交换密钥，我把我的 秘密颜色 和 公开颜色 混在一起

Then, I send that mixed color to John by anymeans – mail, carrier pigeon, whatever.

然后发给约翰，可以写信发，用信鸽发，什么方式都行.

John does the same – mixing his secret paint color with the public color, then sending that to me.

约翰也这样做，把他的秘密颜色和公开颜色混在一起，然后发我

When I receive John’s color, I simply add my private color to create a blend of all three paints.

我收到约翰的颜色之后，把我的秘密颜色加进去，\N 现在3种颜色混合在一起

John does the same with my mixed color.

John 也一样做

And Voila!

瞧!

We both end up with the same paint color!

我们有了一样的颜色

We can use this as a shared secret, even though we never sent each other our individual secret colors.

我们可以把这个颜色当密钥，尽管我们从来没有给对方发过这颜色

A snooping outside observer would know partial information, but they’d find it very difficult to figure out our shared secret color.

外部窥探者可以知道部分信息，但无法知道最终颜色

Of course, sending and mixing paint colors isn’t going to work well for transmitting computer data.

当然，计算机要传输数据时，混合颜料和发颜料不太合适

But luckily, mathematical one-way functions are perfect,

但幸运的是，数学单向函数是完美的

and this is what Diffie-Hellman Key Exchange uses.

我们可以用 "迪菲-赫尔曼密钥交换"

In Diffie-Hellman, the one-way function is modular exponentiation.

在 Diffie-Hellman 中，单向函数是模幂运算

This means taking one number, the base, to the power of another number,

意思是先做幂运算，拿一个数字当底数，拿一个数字当指数，比如 A

b

the exponent, and taking the remainder when dividing by a third number, the modulus.

然后除以第三个数字，最后拿到我们想要的余数

So, for example, if we wanted to calculate 3 to the 5th power, modulo 31,

举个例子，假设我们想算3的5次方，模31

we would calculate 3 to the 5th, which is 243,

我们先算3的5次方，得到243

then take the remainder when divided by 31, which is 26.

\N然后除31，取余数，得到26

The hard part is figuring out the exponent given only the result and the base.

重点是 如果只给余数和基数。很难得知指数是多少

If I tell you I raised 3 to some secret number, modulo 31, and got 7 as the remainder

如果我告诉你，3的某次方 模31，余数是7

you'd have to test a lot of exponents to know which one I picked.

你要试很多次，才能知道次方是多少

If we make these numbers big, say hundreds of digits long,

如果把数字变长一些，比如几百位长

then finding the secret exponent is nearly impossible.

想找到秘密指数是多少，几乎是不可能的。

Now let’s talk about how Diffie-Hellman

现在我们来讨论 Diffie-Hellman 是怎么

uses modular exponentiation to calculate a shared key.

用模幂运算 算出双方共享的密钥

First, there's a set of public values – the base and the modulus,

首先，我们有公开的值 - 基数和模数

that, like our public paint color, everyone gets to know... even the bad guys!

就像公开的油漆颜色，所有人都看的到，甚至坏人!

To send a message securely to John, I would pick a secret exponent: X.

为了安全向 John 发信息，我选一个秘密指数：X

Then, I’d calculate B to the power of X, modulo M.

然后算 B^X mod M 的结果

I send this big number over to John.

然后把这个大数字发给 John.

John does the same, picking a secret exponent Y, and sending me B to the Y modulo M.

John 也一样做，选一个秘密指数Y，然后把 B^Y mod M 的结果发我

To create a shared secret key,

为了算出 双方共用的密钥

I take what John sent me, and take it to the power of X, my secret exponent.

我把 John 给我的数，用我的秘密指数 X，进行模幂运算 (看上图)

This is mathematically equivalent to B to the XY modulus M.

数学上相等于 B的XY次方 模M

John does the same, taking what I sent to him to the power of Y, and we both end up with the exact same number!

John也一样做，拿我给他的数 进行模幂运算，最终得到一样的数

It’s a secret shared key, even though we never sent each other our secret number.

双方有一样的密钥，即使我们从来没给对方发过各自的秘密指数

We can use this big number as a shared key for encrypted communication, using something like AES for encryption.

我们可以用这个大数字当密钥，用 AES 之类的加密技术来加密通信

Diffie-Hellman key exchange is one method for establishing a shared key.

"Diffie-Hellman 密钥交换"是建立共享密钥的一种方法。

These keys that can be used by both sender and receiver, to encrypt and decrypt messages

双方用一样的密钥加密和解密消息，这叫"对称加密", 因为密钥一样

are called symmetric keys because the key is the same on both sides.

双方用一样的密钥加密和解密消息，这叫"对称加密", 因为密钥一样

The Caesar Cipher, Enigma and AES are all symmetric encryption.

凯撒加密，英格玛，AES 都是"对称加密"

There’s also asymmetric encryption, where there are two different keys

还有"非对称加密"，有两个不同的密钥

most often one that’s public and another that’s private.

一个是公开的，另一个是私有的

So, people can encrypt a message using a public key that

人们用公钥加密消息 \N 只有有私钥的人能解密

only the recipient, with their private key, can decrypt.

人们用公钥加密消息 \N 只有有私钥的人能解密

In other words, knowing the public key only lets you encrypt, but not decrypt – it’s asymmetric!

换句话说，知道公钥只能加密但不能解密 \N - 它是"不对称"的！

So, think about boxes with padlocks that you can open with a key.

想象一个可以锁上的盒子

To receive a secure message, I can give a sender a box and padlock.

为了收到安全的信息，我们可以给别人箱子和锁

They put their message in it and lock it shut.

别人把信息放箱子，然后锁起来

Now, they can send that box back to me and only I can open it, with my private key.

把盒子寄回给我，只有我的钥匙能打开

After locking the box, neither the sender,

上锁后，如果发件人或其他人想打开盒子 \N 除了暴力尝试没有其他办法.

nor anyone else who finds the box, can open it without brute force.

上锁后，如果发件人或其他人想打开盒子 \N 除了暴力尝试没有其他办法.

In the same way, a digital public key can encrypt something that can only be decrypted with a private key.

和盒子例子一样，公钥加密后只能私钥来解密.

The reverse is possible too: encrypting something with a

反过来也是可以的：私钥加密后 用公钥解密

private key that can be decrypted with a public key.

反过来也是可以的：私钥加密后 用公钥解密

This is used for signing, where a server encrypts data using their private key.

这种做法用于签名，服务器可以用私钥加密，

Anyone can decrypt it using the server's public key.

任何人都可以用服务器的公钥解密

This acts like an unforgeable signature,

就像一个不可伪造的签名

as only the owner, using their private key, can encrypt.

因为只有私钥的持有人 能加密

It proves that you're getting data from the right server or person, and not an imposter.

这能证明数据来自正确的服务器或个人，而不是某个假冒者

The most popular asymmetric encryption technique used today is RSA,

目前最流行的"非对称加密"技术是 RSA

named after its inventors: Rivest, Shamir and Adleman.

名字来自发明者： Rivest, Shamir, Adleman.

So, now you know all the "key" parts of modern cryptography:

现在你学会了现代密码学的所有"关键"部分：

symmetric encryption, key exchange and public-key cryptography.

对称加密，密钥交换，公钥密码学

When you connect to a secure website, like your bank,

当你访问一个安全的网站，比如银行官网

that little padlock icon means that your computer has used public key cryptography

绿色锁图标代表 用了公钥密码学

to verify the server key exchange to establish a secret temporary key,

验证服务器的密钥，然后建立临时密钥

and symmetric encryption to protect all the back-and-forth communication from prying eyes.

然后用对称加密保证通信安全

Whether you're buying something online, sending emails to BFFs,

不管你是网上购物，发邮件给朋友，还是看猫咪视频

or just browsing cat videos

不管你是网上购物，发邮件给朋友，还是看猫咪视频

cryptography keeps all that safe, private and secure.

密码学都在保护你的隐私和安全

Thanks cryptography!

谢啦密码学！

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne，欢迎收看计算机科学速成课！

As we've touched on many times in this series,

我们之前说过 \N 计算机很擅长存放，整理，获取和处理大量数据

computers are incredible at storing, organizing,

我们之前说过 \N 计算机很擅长存放，整理，获取和处理大量数据

fetching and processing huge volumes of data.

我们之前说过 \N 计算机很擅长存放，整理，获取和处理大量数据

That's perfect for things like e-commerce websites with millions of items for sale,

很适合有上百万商品的电商网站

and for storing billions of health records for quick access by doctors.

或是存几十亿条健康记录，方便医生看.

But what if we want to use computers not just to fetch and display data,

但如果想根据数据做决定呢？

but to actually make decisions about data?

但如果想根据数据做决定呢？

This is the essence of machine learning

这是机器学习的本质

algorithms that give computers the ability to learn from data,

机器学习算法让计算机可以从数据中学习，\N 然后自行做出预测和决定

and then make predictions and decisions.

机器学习算法让计算机可以从数据中学习，\N 然后自行做出预测和决定

Computer prosgrams with this ability

能自我学习的程序很有用 \N 比如判断是不是垃圾邮件

are extremely useful in answering questions like Is an email spam?

能自我学习的程序很有用 \N 比如判断是不是垃圾邮件

Does a person's heart have arrhythmia?

这人有心律失常吗？

or what video should youtube recommend after this one?

YouTube 的下一个视频该推荐哪个？

While useful, we probably wouldn't describe these programs as "intelligent"

虽然有用，但我们不会说它有人类一般的智能

in the same way we think of human intelligence.

虽然有用，但我们不会说它有人类一般的智能

So, even though the terms are often interchanged,

虽然 AI 和 ML 这两词经常混着用

most computer scientists would say that machine learning is a set of techniques

大多数计算机科学家会说 \N机器学习是为了实现人工智能这个更宏大目标的技术之一

that sits inside the even more ambitious goal of Artificial Intelligence,

大多数计算机科学家会说 \N机器学习是为了实现人工智能这个更宏大目标的技术之一

or AI for short.

人工智能简称 AI

Machine Learning and AI algorithms tend to be pretty sophisticated.

机器学习和人工智能算法一般都很复杂

So rather than wading into the mechanics of how they work,

所以我们不讲具体细节 重点讲概念

we're going to focus on what the algorithms do conceptually.

所以我们不讲具体细节 重点讲概念

Let's start with a simple example:

我们从简单例子开始：

deciding if a moth is a Luna Moth or an Emperor Moth.

判断飞蛾是"月蛾"还是"帝蛾"

This decision process is called classification,

这叫"分类"

and an algorithm that does it is called a classifier.

做分类的算法叫 "分类器"

Although there are techniques that can use raw data for training

虽然我们可以用 照片和声音 来训练算法

- like photos and sounds -

虽然我们可以用 照片和声音 来训练算法

many algorithms reduce the complexity of real world objects

很多算法会减少复杂性

and phenomena into what are called features.

把数据简化成 "特征"

Features are values that usefully characterize the things we wish to classify.

"特征"是用来帮助"分类"的值

For our moth example, we're going to use two features: "wingspan" and "mass".

对于之前的飞蛾分类例子\N 我们用两个特征:"翼展"和"重量"

In order to train our machine learning classifier to make good predictions,

为了训练"分类器"做出好的预测，

we're going to need training data.

我们需要"训练数据"

To get that,

为了得到数据

we'd send an entomologist out into a forest to collect data for both luna and emperor moths.

我们派昆虫学家到森林里 收集"月蛾"和"帝蛾"的数据

These experts can recognize different moths,

专家可以认出不同飞蛾，

so they not only record the feature values,

所以专家不只记录特征值，还会把种类也写上

but also label that data with the actual moth species.

所以专家不只记录特征值，还会把种类也写上

This is called labeled data.

这叫 "标记数据"

Because we only have two features,

因为只有两个特征

it's easy to visualize this data in a scatterplot.

很容易用散点图把数据视觉化

Here, I've plotted data for 100 Emperor Moths in red and 100 Luna Moths in blue.

红色标了100个帝蛾\N 蓝色标了100个月蛾

We can see that the species make two groupings, but.

可以看到大致分成了两组

there's some overlap in the middle

但中间有一定重叠

so it's not entirely obvious how to best separate the two.

所以想完全区分两个组比较困难

That's what machine learning algorithms do

所以机器学习算法登场

- find optimal separations!

- 找出最佳区分

I'm just going to eyeball it

我用肉眼大致估算下

and say anything less than 45 millimeters in wingspan is likely to be an Emperor Moth.

然后判断 翼展小于45毫米的 很可能是帝蛾

We can add another division that says additionally mass must be less than .75

可以再加一个条件，重量必须小于.75

in order for our guess to be Emperor Moth.

才算是帝蛾。

These lines that chop up the decision space are called decision boundaries.

这些线叫 "决策边界"

If we look closely at our data,

如果仔细看数据

we can see that 86 emperor moths would correctly end up inside the emperor decision region,

86只帝蛾在正确的区域

but 14 would end up incorrectly in luna moth territory.

但剩下14只在错误的区域

On the other hand, 82 luna moths would be correct,

另一方面，82只月蛾在正确的区域

with 18 falling onto the wrong side.

18个在错误的区域

A table, like this, showing where a classifier gets things right and wrong

这里有个表 记录正确数和错误数

is called a confusion matrix...

这表叫"混淆矩阵"

which probably should have also been the title of the last two movies in the Matrix Trilogy!

"黑客帝国三部曲"的后两部也许该用这个标题

Notice that there's no way for us to draw lines that give us 100% accuracy.

注意我们没法画出 100% 正确分类的线

If we lower our wingspan decision boundary,

降低翼展的决策边界，会把更多"帝蛾"误分类成"月蛾"

we misclassify more Emperor moths as Lunas.

降低翼展的决策边界，会把更多"帝蛾"误分类成"月蛾"

If we raise it, we misclassify more Luna moths.

如果提高，会把更多月蛾分错类.

The job of machine learning algorithms,

机器学习算法的目的

at a high level,

机器学习算法的目的

is to maximize correct classifications while minimizing errors

是最大化正确分类 + 最小化错误分类

On our training data, we get 168 moths correct, and 32 moths wrong,

在训练数据中，有168个正确，32个错误

for an average classification accuracy of 84%.

平均准确率84％

Now, using these decision boundaries,

用这些决策边界

if we go out into the forest and encounter an unknown moth,

如果我们进入森林，碰到一只不认识的飞蛾，

we can measure its features and plot it onto our decision space.

我们可以测量它的特征, 并绘制到决策空间上

This is unlabeled data.

这叫 "未标签数据"

Our decision boundaries offer a guess as to what species the moth is.

决策边界可以猜测飞蛾种类

In this case, we'd predict it's a Luna Moth.

这里我们预测是"月蛾"

This simple approach, of dividing the decision space up into boxes,

这个把决策空间 切成几个盒子的简单方法

can be represented by what's called a decision tree,

可以用"决策树"来表示

which would look like this pictorially or could be written in code using If-Statements, like this.

画成图像，会像左侧 \N 用 if 语句写代码，会像右侧

A machine learning algorithm that produces decision trees

生成决策树的 机器学习算法

needs to choose what features to divide on

需要选择用什么特征来分类

and then for each of those features, what values to use for the division.

每个特征用什么值

Decision Trees are just one basic example of a machine learning technique.

"决策树"只是机器学习的一个简单例子

There are hundreds of algorithms in computer science literature today.

如今有数百种算法，而且新算法不断出现

And more are being published all the time.

如今有数百种算法，而且新算法不断出现

A few algorithms even use many decision trees working together to make a prediction.

一些算法甚至用多个"决策树"来预测

Computer scientists smugly call those Forests

计算机科学家叫这个"森林"，因为有多颗树嘛

because they contain lots of trees.

计算机科学家叫这个"森林"，因为有多颗树嘛

There are also non-tree-based approaches,

也有不用树的方法，比如"支持向量机"

like Support Vector Machines,

也有不用树的方法，比如"支持向量机"

which essentially slice up the decision space using arbitrary lines.

本质上是用任意线段来切分"决策空间"

And these don't have to be straight lines;

不一定是直线

they can be polynomials or some other fancy mathematical function.

可以是多项式或其他数学函数

Like before, it's the machine learning algorithm's job

就像之前，机器学习算法负责

to figure out the best lines to provide the most accurate decision boundaries.

找出最好的线，最准的决策边界

So far, my examples have only had two features,

之前的例子只有两个特征，人类也可以轻松做到

which is easy enough for a human to figure out.

之前的例子只有两个特征，人类也可以轻松做到

If we add a third feature,

如果加第3个特征，比如"触角长度"

let's say, length of antennae,

如果加第3个特征，比如"触角长度"

then our 2D lines become 3D planes,

那么2D线段，会变成3D平面

creating decision boundaries in three dimensions.

在三个维度上做决策边界

These planes don't have to be straight either.

这些平面不必是直的

Plus, a truly useful classifier would contend with many different moth species.

而且 真正有用的分类器 会有很多飞蛾种类

Now I think you'd agree this is getting too complicated to figure out by hand

你可能会同意 现在变得太复杂了

But even this is a very basic example

但这也只是个简单例子

- just three features and five moth species.

- 只有3个特征和5个品种

We can still show it in this 3D scatter plot.

我们依然可以用 3D散点图 画出来

Unfortunately, there's no good way to visualize four features at once, or twenty features,

不幸的是，一次性看4个或20个特征，没有好的方法

let alone hundreds or even thousands of features.

更别说成百上千的特征了

But that's what many real-world machine learning problems face.

但这正是机器学习要面临的问题

Can YOU imagine trying to figure out the equation for a hyperplane

你能想象靠手工 在一个上千维度的决策空间里

rippling through a thousand-dimensional decision space?

给超平面(Hyperplane)找出一个方程吗

Probably not,

大概不行

but computers, with clever machine learning algorithms can

但聪明的机器学习算法可以做到

and they do, all day long, on computers at places like Google, Facebook, Microsoft and Amazon.

Google，Facebook，微软和亚马逊的计算机里\N 整天都在跑这些算法

Techniques like Decision Trees and Support Vector Machines are strongly rooted in the field of statistics,

"决策树"和"支持向量机"这样的技术 \N 发源自统计学

which has dealt with making confident decisions,

统计学早在计算机出现前，就在用数据做决定

using data, long before computers ever existed.

统计学早在计算机出现前，就在用数据做决定

There's a very large class of widely used statistical machine learning techniques,

有一大类机器学习算法用了统计学

but there are also some approaches with no origins in statistics.

但也有不用统计学的算法

Most notable are artificial neural networks,

其中最值得注意的是 人工神经网络

which were inspired by neurons in our brains!

灵感来自大脑里的神经元

For a primer of biological neurons,

想学习神经元知识的人，可以看这3集

check out our three-part overview here,

想学习神经元知识的人，可以看这3集

but basically neurons are cells

神经元是细胞

that process and transmit messages using electrical and chemical signals.

用电信号和化学信号 来处理和传输消息

They take one or more inputs from other cells,

它从其他细胞 得到一个或多个输入

process those signals,

然后处理信号并发出信号

and then emit their own signal.

然后处理信号并发出信号

These form into huge interconnected networks that are able to process complex information.

形成巨大的互联网络，能处理复杂的信息

Just like your brain watching this youtube video.

就像你的大脑 在看这个视频

Artificial Neurons are very similar.

人造神经元很类似

Each takes a series of inputs, combines them, and emits a signal.

可以接收多个输入，然后整合并发出一个信号

Rather than being electrical or chemical signals,

它不用电信号或化学信号

artificial neurons take numbers in, and spit numbers out.

而是吃数字进去，吐数字出来

They are organized into layers that are connected by links,

它们被放成一层层

forming a network of neurons, hence the name.

形成神经元网络，因此得名神经网络

Let's return to our moth example to see how neural nets can be used for classification.

回到飞蛾例子，看如何用神经网络分类

Our first layer - the input layer -

我们的第一层 - 输入层 -

provides data from a single moth needing classification.

提供需要被分类的单个飞蛾数据

Again, we'll use mass and wingspan.

同样，这次也用重量和翼展

At the other end, we have an output layer, with two neurons:

另一边是输出层，有两个神经元：

one for Emperor Moth and another for Luna Moth.

一个是帝蛾，一个是月蛾

The most excited neuron will be our classification decision.

2个神经元里最兴奋的 就是分类结果

In between, we have a hidden layer,

中间有一个隐藏层

that transforms our inputs into outputs, and does the hard work of classification.

负责把输入变成输出，负责干分类这个重活

To see how this is done,

为了看看它是如何分类的

let's zoom into one neuron in the hidden layer.

我们放大"隐藏层"里的一个神经元

The first thing a neuron does is multiply each of its inputs by a specific weight,

神经元做的第一件事 \N 是把每个输入乘以一个权重

let's say 2.8 for its first input, and .1 for it's second input.

假设2.8是第一个输入，0.1是第二个输入。

Then, it sums these weighted inputs together,

然后它会相加输入

which is in this case, is a grand total of 9.74.

总共是9.74

The neuron then applies a bias to this result

然后对这个结果，用一个偏差值处理

- in other words, it adds or subtracts a fixed value,

意思是 加或减一个固定值

for example, minus six, for a new value of 3.74.

比如-6，得到3.74

These bias and inputs weights are initially set to random values when a neural network is created.

做神经网络时，这些偏差和权重，\N一开始会设置成随机值

Then, an algorithm goes in, and starts tweaking all those values to train the neural network,

然后算法会调整这些值 来训练神经网络

using labeled data for training and testing.

使用"标记数据"来训练和测试

This happens over many interactions, gradually improving accuracy

逐渐提高准确性

- a process very much like human learning.

- 很像人类学习的过程

Finally, neurons have an activation function, also called a transfer function,

最后，神经元有激活函数，它也叫传递函数，

that gets applied to the output, performing a final mathematical modification to the result.

会应用于输出，对结果执行最后一次数学修改

For example, limiting the value to a range from negative one and positive one,

例如，把值限制在-1和+1之间

or setting any negative values to 0.

或把负数改成0

We'll use a linear transfer function that passes the value through unchanged,

我们用线性传递函数，它不会改变值

so 3.74 stays as 3.74.

所以3.74还是3.74

So for our example neuron,

所以这里的例子

given the inputs .55 and 82, the output would be 3.74.

输入0.55和82，输出3.74

This is just one neuron,

这只是一个神经元，

but this process of weighting, summing, biasing

但加权，求和，偏置，激活函数

and applying an activation function is computed for all neurons in a layer,

会应用于一层里的每个神经元

and the values propagate forward in the network, one layer at a time.

并向前传播，一次一层

In this example, the output neuron with the highest value is our decision:

数字最高的就是结果：

Luna Moth.

月蛾

Importantly, the hidden layer doesn't have to be just one layer

重要的是，隐藏层不是只能有一层，可以有很多层

it can be many layers deep.

重要的是，隐藏层不是只能有一层，可以有很多层

This is where the term deep learning comes from.

"深度学习"因此得名

Training these more complicated networks takes a lot more computation and data.

训练更复杂的网络 需要更多的计算量和数据

Despite the fact that neural networks were invented over fifty years ago,

尽管神经网络50多年前就发明了

deep neural nets have only been practical very recently,

深层神经网络直到最近才成为可能

thanks to powerful processors,

感谢强大的处理器和超快的GPU

but even more so, wicked fast GPUs.

感谢强大的处理器和超快的GPU

So, thank you gamers for being so demanding about silky smooth framerates!

感谢游戏玩家对帧率的苛刻要求！

A couple of years ago, Google and Facebook

几年前，Google和Facebook

demonstrated deep neural nets that could find faces in photos as well as humans

展示了深度神经网络 \N 在照片中识别人脸的准确率，和人一样高

- and humans are really good at this!

- 人类可是很擅长这个的！

It was a huge milestone.

这是个巨大的里程碑

Now deep neural nets are driving cars,

现在有深层神经网络开车，翻译，诊断医疗状况等等

translating human speech,

现在有深层神经网络开车，翻译，诊断医疗状况等等

diagnosing medical conditions and much more.

现在有深层神经网络开车，翻译，诊断医疗状况等等

These algorithms are very sophisticated,

这些算法非常复杂，但还不够"聪明"

but it's less clear if they should be described as "intelligent".

这些算法非常复杂，但还不够"聪明"

They can really only do one thing like classify moths, find faces, or translate languages.

它们只能做一件事，分类飞蛾，找人脸，翻译

This type of AI is called Weak AI or Narrow AI.

这种AI叫"弱AI"或"窄AI"，只能做特定任务

It's only intelligent at specific tasks.

这种AI叫"弱AI"或"窄AI"，只能做特定任务

But that doesn't mean it's not useful;

但这不意味着它没用

I mean medical devices that can make diagnoses,

能自动做出诊断的医疗设备，

and cars that can drive themselves are amazing!

和自动驾驶的汽车真是太棒了！

But do we need those computers to compose music

但我们是否需要这些计算机来创作音乐

and look up delicious recipes in their free time?

在空闲时间找美味食谱呢？

Probably not.

也许不要

Although that would be kinda cool.

如果有的话 还挺酷的

Truly general-purpose AI, one as smart and well-rounded as a human,

真正通用的，像人一样聪明的AI，叫 "强AI"

is called Strong AI.

真正通用的，像人一样聪明的AI，叫 "强AI"

No one has demonstrated anything close to human-level artificial intelligence yet.

目前没人能做出来 接近人类智能的 AI

Some argue it's impossible,

有人认为不可能做出来

but many people point to the explosion of digitized knowledge

但许多人说 数字化知识的爆炸性增长

- like Wikipedia articles, web pages, and Youtube videos -

- 比如维基百科，网页和Youtube视频 -

as the perfect kindling for Strong AI.

是"强 AI"的完美引燃物

Although you can only watch a maximum of 24 hours of youtube a day,

你一天最多只能看24小时的 YouTube \N 计算机可以看上百万小时

a computer can watch millions of hours.

你一天最多只能看24小时的 YouTube \N 计算机可以看上百万小时

For example, IBM's Watson consults and synthesizes information from 200 million pages of content,

比如，IBM 的沃森吸收了 2 亿个网页的内容

including the full text of Wikipedia.

包括维基百科的全文

While not a Strong AI, Watson is pretty smart,

虽然不是"强AI" 但沃森也很聪明 \N 在2011年的知识竞答中碾压了人类

and it crushed its human competition in Jeopardy way back in 2011.

虽然不是"强AI" 但沃森也很聪明 \N 在2011年的知识竞答中碾压了人类

Not only can AIs gobble up huge volumes of information,

AI不仅可以吸收大量信息 \N 也可以不断学习进步，而且一般比人类快得多

but they can also learn over time, often much faster than humans.

AI不仅可以吸收大量信息 \N 也可以不断学习进步，而且一般比人类快得多

In 2016, Google debuted AlphaGo,

2016 年 Google 推出 AlphaGo

a Narrow AI that plays the fiendishly complicated board game Go.

一个会玩围棋的窄AI

One of the ways it got so good and able to beat the very best human players,

它和自己的克隆版下无数次围棋 \N 从而打败最好的人类围棋选手

was by playing clones of itself millions and millions of times.

它和自己的克隆版下无数次围棋 \N 从而打败最好的人类围棋选手

It learned what worked and what didn't,

学习什么管用，什么不管用 \N 自己发现成功的策略

and along the way, discovered successful strategies all by itself.

学习什么管用，什么不管用 \N 自己发现成功的策略

This is called Reinforcement Learning,

这叫 "强化学习" 是一种很强大的方法

and it's a super powerful approach.

这叫 "强化学习" 是一种很强大的方法

In fact, it's very similar to how humans learn.

和人类的学习方式非常类似

People don't just magically acquire the ability to walk...

人类不是天生就会走路，是上千小时的试错学会的

it takes thousands of hours of trial and error to figure it out.

人类不是天生就会走路，是上千小时的试错学会的

Computers are now on the cusp of learning by trial and error,

计算机现在才刚学会反复试错来学习

and for many narrow problems,

对于很多狭窄的问题，强化学习已被广​​泛使用

reinforcement learning is already widely used.

对于很多狭窄的问题，强化学习已被广​​泛使用

What will be interesting to see, is if these types of learning techniques can be applied more broadly,

有趣的是，如果这类技术可以更广泛地应用

to create human-like, Strong AIs that learn much like how kids learn, but at super accelerated rates.

创造出类似人类的"强AI" \N 能像人类小孩一样学习，但学习速度超快

If that happens, there are some pretty big changes in store for humanity

如果这发生了，对人类可能有相当大的影响

- a topic we'll revisit later.

- 我们以后会讨论

Thanks for watching. See you next week.

感谢收看. 我们下周见

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨 我是Carrie Anne，欢迎收看计算机科学速成课

Today, let's start by thinking about how important vision can be.

今天 我们来思考视觉的重要性

Most people rely on it to prepare food,

大部分人靠视觉来做饭

walk around obstacles,

越过障碍

read street signs,

读路牌

watch videos like this,

看视频

and do hundreds of other tasks.

以及无数其它任务

Vision is the highest bandwidth sense,

视觉是信息最多的感官 \N 比如周围的世界是怎样的，如何和世界交互

and it provides a firehose of information about the state of the world and how to act on it.

视觉是信息最多的感官 \N 比如周围的世界是怎样的，如何和世界交互

For this reason, computer scientists have been trying to give computers vision for half a century,

因此半个世纪来\N 计算机科学家一直在想办法让计算机有视觉

birthing the sub-field of computer vision.

因此诞生了"计算机视觉"这个领域

Its goal is to give computers the ability

目标是让计算机理解图像和视频

to extract high-level understanding from digital images and videos.

目标是让计算机理解图像和视频

As everyone with a digital camera or smartphone knows,

用过相机或手机的都知道 \N 可以拍出有惊人保真度和细节的照片

computers are already really good at capturing photos with incredible fidelity and detail

用过相机或手机的都知道 \N 可以拍出有惊人保真度和细节的照片

- much better than humans in fact.

- 比人类强得多

But as computer vision professor Fei-Fei Li recently said,

但正如计算机视觉教授 李飞飞 最近说的

"Just like to hear is the not the same as to listen.

"听到"不等于"听懂"

To take pictures is not the same as to see."

"看到"不等于"看懂"

As a refresher, images on computers are most often stored as big grids of pixels.

复习一下，图像是像素网格

Each pixel is defined by a color, stored as a combination of three additive primary colors:

每个像素的颜色 通过三种基色定义：红，绿，蓝

red, green and blue.

每个像素的颜色 通过三种基色定义：红，绿，蓝

By combining different intensities of these three colors,

通过组合三种颜色的强度 \N 可以得到任何颜色, 也叫 RGB 值

we can represent any color. what's called a RGB value,

通过组合三种颜色的强度 \N 可以得到任何颜色, 也叫 RGB 值

Perhaps the simplest computer vision algorithm

最简单的计算机视觉算法

- and a good place to start -

最合适拿来入门的

is to track a colored object, like a bright pink ball.

是跟踪一个颜色物体，比如一个粉色的球

The first thing we need to do is record the ball's color.

首先，我们记下球的颜色，保存最中心像素的 RGB 值

For that, we'll take the RGB value of the centermost pixel.

首先，我们记下球的颜色，保存最中心像素的 RGB 值

With that value saved, we can give a computer program an image,

然后给程序喂入图像，让它找最接近这个颜色的像素

and ask it to find the pixel with the closest color match.

然后给程序喂入图像，让它找最接近这个颜色的像素

An algorithm like this might start in the upper right corner,

算法可以从左上角开始，逐个检查像素

and check each pixel, one at time,

算法可以从左上角开始，逐个检查像素

calculating the difference from our target color.

计算和目标颜色的差异

Now, having looked at every pixel,

检查了每个像素后，最贴近的像素，很可能就是球

the best match is very likely a pixel from our ball.

检查了每个像素后，最贴近的像素，很可能就是球

We're not limited to running this algorithm on a single photo;

不只是这张图片 \N 我们可以在视频的每一帧图片跑这个算法

we can do it for every frame in a video,

不只是这张图片 \N 我们可以在视频的每一帧图片跑这个算法

allowing us to track the ball over time.

跟踪球的位置

Of course, due to variations in lighting, shadows, and other effects,

当然，因为光线，阴影和其它影响

the ball on the field is almost certainly not going to be the exact same RGB value as our target color,

球的颜色会有变化，不会和存的 RGB 值完全一样

but merely the closest match.

但会很接近

In more extreme cases, like at a game at night,

如果情况更极端一些 \N 比如比赛是在晚上，追踪效果可能会很差

the tracking might be poor.

如果情况更极端一些 \N 比如比赛是在晚上，追踪效果可能会很差

And if one of the team's jerseys used the same color as the ball,

如果球衣的颜色和球一样，算法就完全晕了

our algorithm would get totally confused.

如果球衣的颜色和球一样，算法就完全晕了

For these reasons, color marker tracking and similar algorithms are rarely used,

因此很少用这类颜色跟踪算法

unless the environment can be tightly controlled.

除非环境可以严格控制

This color tracking example was able to search pixel-by-pixel,

颜色跟踪算法是一个个像素搜索 \N 因为颜色是在一个像素里

because colors are stored inside of single pixels.

颜色跟踪算法是一个个像素搜索 \N 因为颜色是在一个像素里

But this approach doesn't work for features larger than a single pixel,

但这种方法 不适合占多个像素的特征

like edges of objects, which are inherently made up of many pixels.

比如物体的边缘，是多个像素组成的.

To identify these types of features in images,

为了识别这些特征，算法要一块块像素来处理

computer vision algorithms have to consider small regions of pixels,

为了识别这些特征，算法要一块块像素来处理

called patches.

每一块都叫"块"

As an example, let's talk about an algorithm that finds vertical edges in a scene,

举个例子，找垂直边缘的算法

let's say to help a drone navigate safely through a field of obstacles.

假设用来帮无人机躲避障碍

To keep things simple, we're going to convert our image into grayscale,

为了简单，我们把图片转成灰度 \N 不过大部分算法可以处理颜色

although most algorithms can handle color.

为了简单，我们把图片转成灰度 \N 不过大部分算法可以处理颜色

Now let's zoom into one of these poles to see what an edge looks like up close.

放大其中一个杆子，看看边缘是怎样的

We can easily see where the left edge of the pole starts,

可以很容易地看到 杆子的左边缘从哪里开始

because there's a change in color that persists across many pixels vertically.

因为有垂直的颜色变化

We can define this behavior more formally by creating a rule

我们可以弄个规则说

that says the likelihood of a pixel being a vertical edge

某像素是垂直边缘的可能性 \N 取决于左右两边像素的颜色差异程度

is the magnitude of the difference in color

某像素是垂直边缘的可能性 \N 取决于左右两边像素的颜色差异程度

between some pixels to its left and some pixels to its right.

某像素是垂直边缘的可能性 \N 取决于左右两边像素的颜色差异程度

The bigger the color difference between these two sets of pixels,

左右像素的区别越大，这个像素越可能是边缘

the more likely the pixel is on an edge.

左右像素的区别越大，这个像素越可能是边缘

If the color difference is small, it's probably not an edge at all.

如果色差很小，就不是边缘

The mathematical notation for this operation looks like this

这个操作的数学符号 看起来像这样

it's called a kernel or filter.

这叫"核"或"过滤器"

It contains the values for a pixel-wise multiplication,

里面的数字用来做像素乘法

the sum of which is saved into the center pixel.

总和 存到中心像素里

Let's see how this works for our example pixel.

我们来看个实际例子

I've gone ahead and labeled all of the pixels with their grayscale values.

我已经把所有像素转成了灰度值

Now, we take our kernel, and center it over our pixel of interest.

现在把"核"的中心，对准感兴趣的像素

This specifies what each pixel value underneath should be multiplied by.

这指定了每个像素要乘的值

Then, we just add up all those numbers.

然后把所有数字加起来

In this example, that gives us 147.

在这里，最后结果是 147

That becomes our new pixel value.

成为新像素值

This operation, of applying a kernel to a patch of pixels,

把 核 应用于像素块，这种操作叫"卷积"

is call a convolution.

把 核 应用于像素块，这种操作叫"卷积"

Now let's apply our kernel to another pixel.

现在我们把"核"应用到另一个像素

In this case, the result is 1. Just 1.

结果是 1

In other words, it's a very small color difference, and not an edge.

色差很小，不是边缘

If we apply our kernel to every pixel in the photo,

如果把"核"用于照片中每个像素

the result looks like this,

结果会像这样

where the highest pixel values are where there are strong vertical edges.

垂直边缘的像素值很高

Note that horizontal edges, like those platforms in the background,

注意，水平边缘（比如背景里的平台）

are almost invisible.

几乎看不见

If we wanted to highlight those features,

如果要突出那些特征

we'd have to use a different kernel

要用不同的"核"

- one that's sensitive to horizontal edges.

用对水平边缘敏感的"核"

Both of these edge enhancing kernels are called Prewitt Operators,

这两个边缘增强的核叫"Prewitt 算子"

named after their inventor.

以发明者命名

These are just two examples of a huge variety of kernels,

这只是众多"核"的两个例子

able to perform many different image transformations.

"核"能做很多种图像转换

For example, here's a kernel that sharpens images.

比如这个"核"能锐化图像

And here's a kernel that blurs them.

这个"核"能模糊图像

Kernels can also be used like little image cookie cutters that match only certain shapes.

"核"也可以像饼干模具一样，匹配特定形状

So, our edge kernels looked for image patches

之前做边缘检测的"核"

with strong differences from right to left or up and down.

会检查左右或上下的差异

But we could also make kernels that are good at finding lines, with edges on both sides.

但我们也可以做出 擅长找线段的"核"

And even islands of pixels surrounded by contrasting colors.

或者包了一圈对比色的区域

These types of kernels can begin to characterize simple shapes.

这类"核"可以描述简单的形状

For example, on faces, the bridge of the nose tends to be brighter than the sides of the nose,

比如鼻梁往往比鼻子两侧更亮

resulting in higher values for line-sensitive kernels.

所以线段敏感的"核"对这里的值更高

Eyes are also distinctive

眼睛也很独特

- a dark circle sounded by lighter pixels -

- 一个黑色圆圈被外层更亮的一层像素包着

a pattern other kernels are sensitive to.

有其它"核"对这种模式敏感

When a computer scans through an image,

当计算机扫描图像时，最常见的是用一个窗口来扫

most often by sliding around a search window,

当计算机扫描图像时，最常见的是用一个窗口来扫

it can look for combinations of features indicative of a human face.

可以找出人脸的特征组合

Although each kernel is a weak face detector by itself,

虽然每个"核"单独找出脸的能力很弱 \N 但组合在一起会相当准确

combined, they can be quite accurate.

虽然每个"核"单独找出脸的能力很弱 \N 但组合在一起会相当准确

It's unlikely that a bunch of face-like features will cluster together if they're not a face.

不是脸但又有一堆脸的特征在正确的位置，\N 这种情况不太可能

This was the basis of an early and influential algorithm

这是一个早期很有影响力的算法的基础

called Viola-Jones Face Detection.

叫 维奥拉·琼斯 人脸检测算法

Today, the hot new algorithms on the block are Convolutional Neural Networks.

如今的热门算法是 "卷积神经网络"

We talked about neural nets last episode, if you need a primer.

我们上集谈了神经网络，如果需要可以去看看

In short, an artificial neuron

总之，神经网络的最基本单位，是神经元

- which is the building block of a neural network -

总之，神经网络的最基本单位，是神经元

takes a series of inputs, and multiplies each by a specified weight,

它有多个输入，然后会把每个输入 乘一个权重值

and then sums those values all together.

然后求总和

This should sound vaguely familiar, because it's a lot like a convolution.

听起来好像挺耳熟，因为它很像"卷积"

In fact, if we pass a neuron 2D pixel data, rather than a one-dimensional list of inputs,

实际上，如果我们给神经元输入二维像素

it's exactly like a convolution.

完全就像"卷积"

The input weights are equivalent to kernel values,

输入权重等于"核"的值

but unlike a predefined kernel,

但和预定义"核"不同

neural networks can learn their own useful kernels

神经网络可以学习对自己有用的"核"

that are able to recognize interesting features in images.

来识别图像中的特征

Convolutional Neural Networks use banks of these neurons to process image data,

"卷积神经网络"用一堆神经元处理图像数据

each outputting a new image, essentially digested by different learned kernels.

每个都会输出一个新图像，\N 本质上是被不同的"核"处理了

These outputs are then processed by subsequent layers of neurons,

输出会被后面一层神经元处理

allowing for convolutions on convolutions on convolutions.

卷积卷积再卷积

The very first convolutional layer might find things like edges,

第一层可能会发现"边缘"这样的特征

as that's what a single convolution can recognize, as we've already discussed.

单次卷积可以识别出这样的东西，之前说过

The next layer might have neurons that convolve on those edge features

下一层可以在这些基础上识别

to recognize simple shapes, comprised of edges, like corners.

比如由"边缘"组成的角落

A layer beyond that might convolve on those corner features,

然后下一层可以在"角落"上继续卷积

and contain neurons that can recognize simple objects,

下一些可能有识别简单物体的神经元

like mouths and eyebrows.

比如嘴和眉毛

And this keeps going, building up in complexity,

然后不断重复，逐渐增加复杂度

until there's a layer that does a convolution that puts it together:

直到某一层把所有特征放到一起：

eyes, ears, mouth, nose, the whole nine yards,

眼睛，耳朵，嘴巴，鼻子

and says "ah ha, it's a face!"

然后说："啊哈，这是脸！"

Convolutional neural networks aren't required to be many layers deep,

"卷积神经网络"不是非要很多很多层

but they usually are, in order to recognize complex objects and scenes.

但一般会有很多层，来识别复杂物体和场景

That's why the technique is considered deep learning.

所以算是"深度学习"

Both Viola-Jones and Convolutional Neural Networks can be applied to many image recognition problems,

"维奥拉·琼斯"和"卷积神经网络"\N 不只是认人脸，还可以识别手写文字

beyond faces, like recognizing handwritten text,

"维奥拉·琼斯"和"卷积神经网络"\N 不只是认人脸，还可以识别手写文字

spotting tumors in CT scans and monitoring traffic flow on roads.

在 CT 扫描中发现肿瘤，监测马路是否拥堵

But we're going to stick with faces.

但我们这里接着用人脸举例

Regardless of what algorithm was used, once we've isolated a face in a photo,

不管用什么算法，识别出脸之后

we can apply more specialized computer vision algorithms to pinpoint facial landmarks,

可以用更专用的计算机视觉算法 \N 来定位面部标志

like the tip of the nose and corners of the mouth.

比如鼻尖和嘴角

This data can be used for determining things like if the eyes are open,

有了标志点，判断眼睛有没有张开就很容易了

which is pretty easy once you have the landmarks

有了标志点，判断眼睛有没有张开就很容易了

it's just the distance between points.

只是点之间的距离罢了

We can also track the position of the eyebrows;

也可以跟踪眉毛的位置

their relative position to the eyes can be an indicator of surprise, or delight.

眉毛相对眼睛的位置 可以代表惊喜或喜悦

Smiles are also pretty straightforward to detect based on the shape of mouth landmarks.

根据嘴巴的标志点，检测出微笑也很简单

All of this information can be interpreted by emotion recognition algorithms,

这些信息可以用"情感识别算法"来识别

giving computers the ability to infer when you're happy, sad, frustrated, confused and so on.

让电脑知道你是开心，忧伤，沮丧，困惑等等

In turn, that could allow computers to intelligently adapt their behavior...

然后计算机可以做出合适的行为.

maybe offer tips when you're confused,

比如当你不明白时 给你提示

and not ask to install updates when you're frustrated.

你心情不好时，就不弹更新提示了

This is just one example of how vision can give computers the ability to be context sensitive,

这只是计算机通过视觉感知周围的一个例子

that is, aware of their surroundings.

这只是计算机通过视觉感知周围的一个例子

And not just the physical surroundings

不只是物理环境 - 比如是不是在上班，或是在火车上

- like if you're at work or on a train -

不只是物理环境 - 比如是不是在上班，或是在火车上

but also your social surroundings

还有社交环境 - 比如是朋友的生日派对，还是正式商务会议

- like if you're in a formal business meeting versus a friend's birthday party.

还有社交环境 - 比如是朋友的生日派对，还是正式商务会议

You behave differently in those surroundings, and so should computing devices,

你在不同环境会有不同行为，计算机也应如此

if they're smart.

如果它们够聪明的话...

Facial landmarks also capture the geometry of your face,

面部标记点 也可以捕捉脸的形状

like the distance between your eyes and the height of your forehead.

比如两只眼睛之间的距离，以及前额有多高

This is one form of biometric data,

做生物识别

and it allows computers with cameras to recognize you.

让有摄像头的计算机能认出你

Whether it's your smartphone automatically unlocking itself when it sees you,

不管是手机解锁 还是政府用摄像头跟踪人

or governments tracking people using CCTV cameras,

不管是手机解锁 还是政府用摄像头跟踪人

the applications of face recognition seem limitless.

人脸识别有无限应用场景

There have also been recent breakthroughs in landmark tracking for hands and whole bodies,

另外 跟踪手臂和全身的标记点，最近也有一些突破

giving computers the ability to interpret a user's body language,

让计算机理解用户的身体语言

and what hand gestures they're frantically waving at their internet connected microwave.

比如用户给联网微波炉的手势

As we've talked about many times in this series,

正如系列中常说的，抽象是构建复杂系统的关键

abstraction is the key to building complex systems,

正如系列中常说的，抽象是构建复杂系统的关键

and the same is true in computer vision.

计算机视觉也是一样

At the hardware level, you have engineers building better and better cameras,

硬件层面，有工程师在造更好的摄像头 \N 让计算机有越来越好的视力

giving computers improved sight with each passing year,

硬件层面，有工程师在造更好的摄像头 \N 让计算机有越来越好的视力

which I can't say for myself.

我自己的视力却不能这样

Using that camera data,

用来自摄像头的数据 可以用视觉算法找出脸和手

you have computer vision algorithms crunching pixels to find things like faces and hands.

用来自摄像头的数据 可以用视觉算法找出脸和手

And then, using output from those algorithms,

然后可以用其他算法接着处理，解释图片中的东西

you have even more specialized algorithms for interpreting things

然后可以用其他算法接着处理，解释图片中的东西

like user facial expression and hand gestures.

比如用户的表情和手势

On top of that, there are people building novel interactive experiences,

有了这些，人们可以做出新的交互体验

like smart TVs and intelligent tutoring systems,

比如智能电视和智能辅导系统 \N 会根据用户的手势和表情来回应

that respond to hand gestures and emotion.

比如智能电视和智能辅导系统 \N 会根据用户的手势和表情来回应

Each of these levels are active areas of research,

这里的每一层都是活跃的研究领域

with breakthroughs happening every year.

每年都有突破，这只是冰山一角

And that's just the tip of the iceberg.

每年都有突破，这只是冰山一角

Today, computer vision is everywhere

如今 计算机视觉无处不在

- whether it's barcodes being scanned at stores,

- 商店里扫条形码 \N 等红灯的自动驾驶汽车

self-driving cars waiting at red lights,

- 商店里扫条形码 \N 等红灯的自动驾驶汽车

or snapchat filters superimposing mustaches.

或是 Snapchat 里添加胡子的滤镜

And, the most exciting thing is that computer scientists are really just getting started,

令人兴奋的是 一切才刚刚开始

enabled by recent advances in computing, like super fast GPUs.

最近的技术发展，比如超快的GPU，\N 会开启越来越多可能性

Computers with human-like ability to see is going to totally change how we interact with them.

视觉能力达到人类水平的计算机 \N 会彻底改变交互方式

Of course, it'd also be nice if they could hear and speak,

当然，如果计算机能听懂我们然后回话，就更好了

which we'll discuss next week. I'll see you then.

我们下周讨论 到时见

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨 我是Carrie Anne，欢迎收看计算机科学速成课

Last episode we talked about computer vision

上集我们讨论了计算机视觉 - 让电脑能看到并理解

- giving computers the ability to see and understand visual information.

上集我们讨论了计算机视觉 - 让电脑能看到并理解

Today we're going to talk about how to give computers the ability to understand language.

今天我们讨论 怎么让计算机理解语言

You might argue they've always had this capability.

你可能会说：计算机已经有这个能力了

Back in Episodes 9 and 12,

在第9和第12集

we talked about machine language instructions,

我们聊了机器语言和更高层次的编程语言

as well as higher-level programming languages.

我们聊了机器语言和更高层次的编程语言

While these certainly meet the definition of a language,

虽然从定义来说 它们也算语言

they also tend to have small vocabularies and follow highly structured conventions.

但词汇量一般很少，而且非常结构化

Code will only compile and run if it's 100 percent free of spelling and syntactic errors.

代码只能在拼写和语法完全正确时，编译和运行

Of course, this is quite different from human languages

当然，这和人类语言完全不同 \N - 人类语言叫"自然语言"

- what are called natural languages -

当然，这和人类语言完全不同 \N - 人类语言叫"自然语言"

containing large, diverse vocabularies,

自然语言有大量词汇

words with several different meanings,

有些词有多种含义

speakers with different accents,

不同口音

and all sorts of interesting word play.

以及各种有趣的文字游戏

People also make linguistic faux pas when writing and speaking,

人们在写作和说话时也会犯错

like slurring words together, leaving out key details so things are ambiguous,

比如单词拼在一起发音 \N 关键细节没说 导致意思模糊两可

and mispronouncing things.

以及发错音

But, for the most part, humans can roll right through these challenges.

但大部分情况下，另一方能理解

The skillful use of language is a major part of what makes us human.

人类有强大的语言能力

And for this reason,

因此，让计算机拥有语音对话的能力

the desire for computers to understand and speak our language

因此，让计算机拥有语音对话的能力

has been around since they were first conceived.

这个想法从构思计算机时就有了

This led to the creation of Natural Language Processing, or NLP,

"自然语言处理"因此诞生，简称 NLP

an interdisciplinary field combining computer science and linguistics.

结合了计算机科学和语言学的 一个跨学科领域

There's an essentially infinite number of ways to arrange words in a sentence.

单词组成句子的方式有无限种

We can't give computers a dictionary of all possible sentences

我们没法给计算机一个字典，包含所有可能句子

to help them understand what humans are blabbing on about.

让计算机理解人类在嘟囔什么

So an early and fundamental NLP problem was deconstructing sentences into bite-sized pieces,

所以 NLP 早期的一个基本问题是 \N 怎么把句子切成一块块

which could be more easily processed.

这样更容易处理

In school, you learned about nine fundamental types of English words:

上学时，老师教你 英语单词有九种基本类型：

nouns, pronouns, articles, verbs, adjectives,

名词，代词，冠词，动词，形容词

adverbs, prepositions, conjunctions, and interjections.

副词，介词，连词和感叹词

These are called parts of speech.

这叫"词性"

There are all sorts of subcategories too,

还有各种子类，比如

like singular vs. plural nouns and superlative vs. comparative adverbs,

单数名词 vs 复数名词 \N 副词最高级 vs 副词比较级

but we're not going to get into that.

但我们不会深入那些.

Knowing a word's type is definitely useful,

了解单词类型有用

but unfortunately, there are a lot words that have multiple meanings - like "rose" and "leaves",

但不幸的是，很多词有多重含义 比如 rose 和 leaves

which can be used as nouns or verbs.

可以用作名词或动词

A digital dictionary alone isn't enough to resolve this ambiguity,

仅靠字典，不能解决这种模糊问题

so computers also need to know some grammar.

所以电脑也要知道语法

For this, phrase structure rules were developed, which encapsulate the grammar of a language.

因此开发了 "短语结构规则" 来代表语法规则

For example, in English there's a rule

例如，英语中有一条规则

that says a sentence can be comprised of a noun phrase followed by a verb phrase.

句子可以由一个名词短语和一个动词短语组成

Noun phrases can be an article, like "the",

名词短语可以是冠词，如 the

followed by a noun or they can be an adjective followed by a noun.

然后一个名词，或一个形容词后面跟一个名词

And you can make rules like this for an entire language.

你可以给一门语言制定出一堆规则

Then, using these rules, it's fairly easy to construct what's called a parse tree,

用这些规则，可以做出"分析树"

which not only tags every word with a likely part of speech,

它给每个单词标了可能是什么词性

but also reveals how the sentence is constructed.

也标明了句子的结构

These smaller chunks of data allow computers to more easily access,

数据块更小 更容易处理

process and respond to information.

数据块更小 更容易处理

Equivalent processes are happening every time you do a voice search,

每次语音搜索，都有这样的流程

like: "where's the nearest pizza".

比如 "最近的披萨在哪里"

The computer can recognize that this is a "where" question,

计算机能明白这是"哪里"（where）的问题

knows you want the noun "pizza",

知道你想要名词"披萨"（pizza）

and the dimension you care about is "nearest".

而且你关心的维度是"最近的"（nearest）

The same process applies to "what is the biggest giraffe?" or "who sang thriller?"

"最大的长颈鹿是什么？"或"Thriller是谁唱的？" \N 也是这样处理

By treating language almost like lego,

把语言像乐高一样拆分，方便计算机处理

computers can be quite adept at natural language tasks.

把语言像乐高一样拆分，方便计算机处理

They can answer questions and also process commands,

计算机可以回答问题 以及处理命令

like "set an alarm for 2:20"

比如"设 2:20 的闹钟"

or "play T-Swizzle on spotify".

或"用 Spotify 播放 T-Swizzle"

But, as you've probably experienced, they fail when you start getting too fancy,

但你可能体验过，如果句子复杂一点

and they can no longer parse the sentence correctly, or capture your intent.

计算机就没法理解了

Hey Siri... me thinks the mongols doth roam too much,

嘿Siri ...... 俺觉得蒙古人走得太远了

what think ye on this most gentle mid-summer's day?

在这个最温柔的夏日的日子里，你觉得怎么样？

Siri: I'm not sure I got that.

Siri：我没明白

I should also note that phrase structure rules, and similar methods that codify language,

还有，"短语结构规则"和其他把语言结构化的方法

can be used by computers to generate natural language text.

可以用来生成句子

This works particularly well when data is stored in a web of semantic information,

数据存在语义信息网络时，这种方法特别有效

where entities are linked to one another in meaningful relationships,

实体互相连在一起

providing all the ingredients you need to craft informational sentences.

提供构造句子的所有成分

Siri: Thriller was released in 1983 and sung by Michael Jackson

Siri：Thriller 于1983年发行，由迈克尔杰克逊演唱

Google's version of this is called Knowledge Graph.

Google 版的叫"知识图谱"

At the end of 2016,

在2016年底

it contained roughly seventy billion facts about, and relationships between, different entities.

包含大概七百亿个事实，以及不同实体间的关系

These two processes, parsing and generating text,

处理, 分析, 生成文字 \N 是聊天机器人的最基本部件

are fundamental components of natural language chatbots

处理, 分析, 生成文字 \N 是聊天机器人的最基本部件

- computer programs that chat with you.

- 聊天机器人就是能和你聊天的程序

Early chatbots were primarily rule-based,

早期聊天机器人大多用的是规则.

where experts would encode hundreds of rules mapping what a user might say,

专家把用户可能会说的话，和机器人应该回复什么，\N 写成上百个规则

to how a program should reply.

专家把用户可能会说的话，和机器人应该回复什么，\N 写成上百个规则

Obviously this was unwieldy to maintain and limited the possible sophistication.

显然，这很难维护，而且对话不能太复杂.

A famous early example was ELIZA, created in the mid-1960s at MIT.

一个著名早期例子叫 Eliza\N 1960年代中期 诞生于麻省理工学院

This was a chatbot that took on the role of a therapist,

一个治疗师聊天机器人

and used basic syntactic rules to identify content in written exchanges,

它用基本句法规则 来理解用户打的文字

which it would turn around and ask the user about.

然后向用户提问

Sometimes, it felt very much like human-human communication,

有时候会感觉像和人类沟通一样

but other times it would make simple and even comical mistakes.

但有时会犯简单 甚至很搞笑的错误

Chatbots, and more advanced dialog systems,

聊天机器人和对话系统

have come a long way in the last fifty years, and can be quite convincing today!

在过去五十年发展了很多，如今可以和真人很像!

Modern approaches are based on machine learning,

如今大多用机器学习

where gigabytes of real human-to-human chats are used to train chatbots.

用上GB的真人聊天数据 来训练机器人

Today, the technology is finding use in customer service applications,

现在聊天机器人已经用于客服回答

where there's already heaps of example conversations to learn from.

客服有很多对话可以参考

People have also been getting chatbots to talk with one another,

人们也让聊天机器人互相聊天

and in a Facebook experiment, chatbots even started to evolve their own language.

在 Facebook 的一个实验里，\N 聊天机器人甚至发展出自己的语言

This experiment got a bunch of scary-sounding press,

很多新闻把这个实验 报导的很吓人

but it was just the computers crafting a simplified protocol to negotiate with one another.

但实际上只是计算机 \N 在制定简单协议来帮助沟通

It wasn't evil, it's was efficient.

这些语言不是邪恶的，而是为了效率

But what about if something is spoken

但如果听到一个句子

- how does a computer get words from the sound?

- 计算机怎么从声音中提取词汇？

That's the domain of speech recognition,

这个领域叫"语音识别"

which has been the focus of research for many decades.

这个领域已经重点研究了几十年

Bell Labs debuted the first speech recognition system in 1952,

贝尔实验室在1952年推出了第一个语音识别系统

nicknamed Audrey, the automatic digit recognizer.

绰号 Audrey，自动数字识别器

It could recognize all ten numerical digits,

如果你说得够慢，它可以识别全部十位数字

if you said them slowly enough.

如果你说得够慢，它可以识别全部十位数字

5

9

7？

The project didn't go anywhere

这个项目没有实际应用，因为手输快得多

because it was much faster to enter telephone numbers with a finger.

这个项目没有实际应用，因为手输快得多

Ten years later, at the 1962 World's Fair,

十年后，1962年的世界博览会上

IBM demonstrated a shoebox-sized machine capable of recognizing sixteen words.

IBM展示了一个鞋盒大小的机器，能识别16个单词

To boost research in the area,

为了推进"语音识别"领域的研究

DARPA kicked off an ambitious five-year funding initiative in 1971,

DARPA 在1971年启动了一项雄心勃勃的五年筹资计划

which led to the development of Harpy at Carnegie Mellon University.

之后诞生了卡内基梅隆大学的 Harpy

Harpy was the first system to recognize over a thousand words.

Harpy 是第一个可以识别1000个单词以上的系统

But, on computers of the era,

但那时的电脑

transcription was often ten or more times slower than the rate of natural speech.

语音转文字，经常比实时说话要慢十倍或以上

Fortunately, thanks to huge advances in computing performance in the 1980s and 90s,

幸运的是，1980,1990年代 计算机性能的大幅提升

continuous, real-time speech recognition became practical.

实时语音识别变得可行

There was simultaneous innovation in the algorithms for processing natural language,

同时也出现了处理自然语言的新算法

moving from hand-crafted rules,

不再是手工定规则

to machine learning techniques

而是用机器学习

that could learn automatically from existing datasets of human language.

从语言数据库中学习

Today, the speech recognition systems with the best accuracy are using deep neural networks,

如今准确度最高的语音识别系统 用深度神经网络

which we touched on in Episode 34.

我们在第34集讲过

To get a sense of how these techniques work,

为了理解原理

let's look at some speech, specifically,

我们来看一些对话声音

the acoustic signal.

我们来看一些对话声音

Let's start by looking at vowel sounds,

先看元音

like aaaaa and eeeeee.

比如 a 和 e

These are the waveforms of those two sounds, as captured by a computer's microphone.

这是两个声音的波形

As we discussed in Episode 21 - on Files and File Formats -

我们在第21集（文件格式）说过

this signal is the magnitude of displacement,

这个信号来自 麦克风内部隔膜震动的频率

of a diaphragm inside of a microphone, as sound waves cause it to oscillate.

这个信号来自 麦克风内部隔膜震动的频率

In this view of sound data, the horizontal axis is time,

在这个视图中，横轴是时间

and the vertical axis is the magnitude of displacement, or amplitude.

竖轴是隔膜移动的幅度，或者说振幅

Although we can see there are differences between the waveforms,

虽然可以看到2个波形有区别

it's not super obvious what you would point at to say,

但不能看出

"ah ha! this is definitely an eeee sound".

"啊！这个声音肯定是 e"

To really make this pop out, we need to view the data in a totally different way:

为了更容易识别，我们换个方式看：

a spectrogram.

谱图

In this view of the data, we still have time along the horizontal axis,

这里横轴还是时间

but now instead of amplitude on the vertical axis,

但竖轴不是振幅

we plot the magnitude of the different frequencies that make up each sound.

而是不同频率的振幅

The brighter the color, the louder that frequency component.

颜色越亮，那个频率的声音越大

This conversion from waveform to frequencies is done with a very cool algorithm called

这种波形到频率的转换 是用一种很酷的算法做的

a Fast Fourier Transform.

快速傅立叶变换（FFT）

If you've ever stared at a stereo system's EQ visualizer,

如果你盯过立体声系统的 EQ 可视化器

it's pretty much the same thing.

它们差不多是一回事

A spectrogram is plotting that information over time.

谱图是随着时间变化的

You might have noticed that the signals have a sort of ribbed pattern to them

你可能注意到，信号有种螺纹图案

that's all the resonances of my vocal tract.

那是我声道的回声

To make different sounds,

为了发出不同声音

I squeeze my vocal chords, mouth and tongue into different shapes,

我要把声带，嘴巴和舌头变成不同形状

which amplifies or dampens different resonances.

放大或减少不同的共振

We can see this in the signal, with areas that are brighter, and areas that are darker.

可以看到有些区域更亮，有些更暗

If we work our way up from the bottom, labeling where we see peaks in the spectrum

如果从底向上看，标出高峰

- what are called formants -

- 叫"共振峰" -

we can see the two sounds have quite different arrangements.

可以看到有很大不同

And this is true for all vowel sounds.

所有元音都是如此

It's exactly this type of information that lets computers recognize spoken vowels,

这让计算机可以识别元音

and indeed, whole words.

然后识别出整个词

Let's see a more complicated example,

让我们看一个更复杂的例子

like when I say: "she.. was.. happy"

当我说"她..很开心"的时候

We can see our "eee" sound here, and "aaa" sound here.

可以看到 e 声，和 a 声

We can also see a bunch of other distinctive sounds,

以及其它不同声音

like the "shh" sound in "she",

比如 she 中的 shh 声

the "wah" and "sss" in "was", and so on.

was 中的 wah 和 sss，等等

These sound pieces, that make up words,

这些构成单词的声音片段

are called phonemes.

叫"音素"

Speech recognition software knows what all these phonemes look like.

语音识别软件 知道这些音素

In English, there are roughly forty-four,

英语有大概44种音素

so it mostly boils down to fancy pattern matching.

所以本质上变成了音素识别

Then you have to separate words from one another,

还要把不同的词分开

figure out when sentences begin and end...

弄清句子的开始和结束点

and ultimately, you end up with speech converted into text,

最后把语音转成文字

allowing for techniques like we discussed at the beginning of the episode.

使这集视频开头里讨论的那些技术成为可能

Because people say words in slightly different ways,

因为口音和发音错误等原因

due to things like accents and mispronunciations,

人们说单词的方式略有不同

transcription accuracy is greatly improved when combined with a language model,

所以结合语言模型后，语音转文字的准确度会大大提高

which contains statistics about sequences of words.

里面有单词顺序的统计信息

For example "she was" is most likely to be followed by an adjective, like "happy".

比如："她"后面很可能跟一个形容词，\N 比如"很开心"

It's uncommon for "she was" to be followed immediately by a noun.

"她"后面很少是名词

So if the speech recognizer was unsure between, "happy" and "harpy",

如果不确定是 happy 还是 harpy，会选 happy

it'd pick "happy",

如果不确定是 happy 还是 harpy，会选 happy

since the language model would report that as a more likely choice.

因为语言模型认为可能性更高

Finally, we need to talk about Speech Synthesis,

最后, 我们来谈谈 "语音合成"

that is, giving computers the ability to output speech.

让计算机输出语音

This is very much like speech recognition, but in reverse.

它很像语音识别，不过反过来

We can take a sentence of text, and break it down into its phonetic components,

把一段文字，分解成多个声音

and then play those sounds back to back, out of a computer speaker.

然后播放这些声音

You can hear this chaining of phonemes very clearly with older speech synthesis technologies,

早期语音合成技术，可以清楚听到音素是拼在一起的

like this 1937, hand-operated machine from Bell Labs.

比如这个1937年贝尔实验室的手动操作机器

Say, "she saw me" with no expression.

不带感情的说"她看见了我"

She saw me.

她看见了我

Now say it in answer to these questions.

现在回答问题

Who saw you?

谁看见你了？

She saw me.

她看见了我

Who did she see?

她看到了谁？

She saw me.

她看见了我

Did she see you or hear you?

她看到你还是听到你说话了？

She saw me.

她看见了我

By the 1980s, this had improved a lot,

到了1980年代，技术改进了很多

but that discontinuous and awkward blending of phonemes

但音素混合依然不够好，产生明显的机器人声

still created that signature, robotic sound.

但音素混合依然不够好，产生明显的机器人声

Thriller was released in 1983 and sung by Michael Jackson.

Thriller 于1983年发行，迈克尔·杰克逊 演唱.

Today, synthesized computer voices, like Siri, Cortana and Alexa,

如今，电脑合成的声音，比如 Siri, Cortana, Alexa

have gotten much better, but they're still not quite human.

好了很多，但还不够像人

But we're soo soo close,

但我们非常非常接近了

and it's likely to be a solved problem pretty soon.

这个问题很快会被解决

Especially because we're now seeing an explosion of voice user interfaces on our phones,

现在语音界面到处都是，手机里

in our cars and homes, and maybe soon, plugged right into our ears.

汽车里，家里，也许不久之后耳机也会有.

This ubiquity is creating a positive feedback loop,

这创造一个正反馈循环

where people are using voice interaction more often,

人们用语音交互的频率会提高

which in turn, is giving companies like Google, Amazon and Microsoft

这又给了谷歌，亚马逊，微软等公司

more data to train their systems on.

更多数据来训练语音系统.

Which is enabling better accuracy,

提高准确性

which is leading to people using voice more,

准确度高了，人们更愿意用语音交互

which is enabling even better accuracy and the loop continues!

越用越好，越好越用

Many predict that speech technologies will become as common a form of interaction

很多人预测，语音交互会越来越常见

as screens, keyboards, trackpads and other physical input-output devices that we use today.

就像如今的屏幕，键盘，触控板等设备

That's particularly good news for robots,

这对机器人发展是个好消息

who don't want to have to walk around with keyboards in order to communicate with humans.

机器人就不用走来走去时 带个键盘和人类沟通

But, we'll talk more about them next week. See you then.

下周我们讲机器人. 到时见

Hi, I’m Carrie Anne, and welcome to Crash Course Computer Science!

嗨，我是 Carrie Anne，欢迎收看计算机速成课

Today we’re going to talk about robots.

今天 我们要讨论机器人

The first image that jumps to your mind is probably a humanoid robot,

你脑中冒出来的第一个印象估计是 类人机器人

like we usually see in shows or movies.

经常在电视剧和电影里看到.

Sometimes they’re our friends and colleagues,

有时候它们是朋友和同事

but more often, they're sinister, apathetic and battle-hardened.

但更常见的是阴险无情，身经百战

We also tend to think of robots as a technology of the future.

我们经常把机器人看成未来科技

But the reality is: they’re already here – by the millions

但事实是：机器人时代已经来临了

– and they're our workmates,

- 它们是同事

helping us to do things harder, better, faster, and stronger.

帮我们把困难的工作，做得更快更好

There are many definitions for robots, but in general,

机器人的定义有很多种，但总的来说，

these are machines capable of carrying out a series of actions automatically

机器人由计算机控制，可以自动执行一系列动作的机器

guided by computer control.

机器人由计算机控制，可以自动执行一系列动作的机器

How they look isn’t part of the equation –

外观并不重要

robots can be industrial arms that spray paint cars,

可以是给汽车喷漆的机械臂

drones that fly, snake-like medical robots that assist surgeons,

无人机，或辅助外科医生的蛇状机器人

as well as humanoid robotic assistants.

以及人形机器人

Although the term "robot" is sometimes

有时我们叫虚拟人物"机器人"

applied to interactive virtual characters,

有时我们叫虚拟人物"机器人"

it’s more appropriate to call these "bots", or even better, "agents."

但叫 bot 甚至 agent 会更合适

That’s because the term "robot" carries a physical connotation

因为"机器人"的潜在含义是 存在于现实世界中的机器

a machine that lives in and acts on the real world.

因为"机器人"的潜在含义是 存在于现实世界中的机器

The word "robot" was first used in a 1920 Czech play

robot (机器人) 一词 \N 首次出现在1920年的一部捷克戏剧

to denote artificial, humanoid characters.

代表人造的类人角色

The word was derived from "robota", the slavic-language word for a forced laborer,

robot 源于斯拉夫语词汇 robota \N 代表强迫劳动

indicating peasants in compulsory service in feudal, nineteenth century Europe.

代表农民在十九世纪 \N 欧洲封建社会的强迫劳动

The play didn’t go too much into technological details.

戏剧没讲太多技术细节

But, even a century later, it’s still a common portrayal:

但即使一个世纪后，这种描述依然很普遍：

mass-produced, efficient, tireless creatures that look human-esque,

机器人都是大规模生产，高效不知疲倦，看起来像人的东西

but are emotionless, indifferent to self-preservation and lack creativity.

但毫无情感，不会保护自己，没有创造力

The more general idea of self-operating machines

更广义的自动运行机器，早在1920年代前就有了

goes back even further than the 1920s.

更广义的自动运行机器，早在1920年代前就有了

Many ancient inventors created mechanical devices that

很多古代发明家 \N 发明了能自动运行的机械装置

performed functions automatically,

很多古代发明家 \N 发明了能自动运行的机械装置

like keeping the time and striking bells on the hour.

比如计时和定时敲钟

There are plenty of examples of automated animal and humanoid figures,

有很多装置 有动物和人类的形象 \N 能跳舞，唱歌，打鼓等

that would perform dances, sing songs, strike drums and do other physical actions.

有很多装置 有动物和人类的形象 \N 能跳舞，唱歌，打鼓等

These non-electrical and certainly non-electronic machines were called automatons.

这些不用电，而且肯定没有电子部件的机器，叫"自动机"

For instance, an early automaton created in 1739

举个例子 \N 1739年法国人 Jacques de Vaucans 做了个自动机

by the Frenchman Jacques de Vaucanson

举个例子 \N 1739年法国人 Jacques de Vaucans 做了个自动机

was the Canard Digerateur or Digesting Duck,

法语叫 Canard Digerateur，翻译过来是 "吃饭鸭"

a machine in the shape of a duck that appeared

一个像鸭子的机器，能吃东西然后排便

to eat grain and then defecate.

一个像鸭子的机器，能吃东西然后排便

In 1739 Voltaire wrote,

伏尔泰在1739年写

"Without the voice of le Maure and Vaucanson’s duck,

"如果没有吃饭鸭的声音

you would have nothing to remind you of the glory of France."

还有什么能提醒你法国的荣光呢？"

One of the most infamous examples was the "Mechanical Turk":

一个名声很臭的例子是"土耳其行棋傀儡"

a chess-playing, humanoid automaton.

一个能下国际象棋的人形机器人

After construction in 1770, it toured all over Europe,

在1770年建造完成后，就在欧洲各地展览

wowing audiences with its surprisingly good chess-playing.

好棋艺惊叹观众

It appeared to be a mechanical, artificial intelligence.

像某种机械人工智能

Unfortunately, it was a hoax – there was a dainty human stuffed inside the machine.

不幸的是，这是个骗局 - 机器里有人控制

The first machines controlled by computers emerged in the late 1940s.

第一台计算机控制的机器，出现在1940年代晚期

These Computer Numerical Control, or CNC machines,

这些计算机数控的机器，简称 CNC 机器

could run programs that instructed a machine to perform a series of operations.

可以执行一连串 程序指定的操作

This level of control also enabled the creation of new manufactured goods,

精细的控制 让我们能生产之前很难做的物品

like milling a complex propellor design out of a block of aluminum

比如从一整块铝 加工出复杂的螺旋桨

– something that was difficult to do using standard machine tools,

- 这用普通机械工具很难做到

and with tolerances too small to be done by hand.

并且误差容忍度很小，无法手工加工

CNC machines were a huge boon to industry,

CNC 机器大大推进了制造业

not just due to increased capability and precision,

不仅提高了制造能力和精确度 还降低了生产成本

but also in terms of reducing labor costs by automating human jobs

不仅提高了制造能力和精确度 还降低了生产成本

– a topic we'll revisit in a later episode.

- 我们之后会深入讨论这个（第40集）

The first commercial deployment was a programmable industrial robot

第一个商业贩卖的 可编程工业机器人

called the Unimate, sold to General Motors in 1960

叫 Unimate，于1960年卖给通用汽车公司

to lift hot pieces of metal from a die casting machine and stack them.

它可以把压铸机做出来的热金属成品提起来，然后堆起来

This was the start of the robotics industry.

机器人行业由此开始

Soon, robots were stacking pallets, welding parts, painting cars and much more.

很快，机器人开始堆叠货盘，焊接，给汽车喷漆等等

For simple motions – like a robotic gripper that moves back and forth on a track

对于简单运动 - 比如机器爪子 在轨道上来回移动

a robot can be instructed to move to a particular position,

可以指示它移动到特定位置

and it'll keep moving in that direction until the desired position is reached

它会一直朝那个方向移动，直到到达 \N 然后停下来

at which point it’ll stop.

它会一直朝那个方向移动，直到到达 \N 然后停下来

This behavior can be achieved through a simple control loop.

这种行为 可以用简单控制回路做

First, sense the robot position.

首先，判断机器人的位置

Are we there yet?

我们到了吗？

Nope.

没有

So keep moving.

那么继续前进

Now sense position again.

再次判断位置

Are we there yet?

我们到了吗？

Nope, so keep moving.

没有，所以继续前进

Are we there yet?

我们到了吗？

Yes!

是的!

So we can stop moving, and also please be quiet!

现在可以停下来了，别问了！

Because we’re trying to minimize the distance between

因为我们在不断缩小 当前位置和目标位置的距离

the sensed position and the desired position,

因为我们在不断缩小 当前位置和目标位置的距离

this control loop is, more specifically, a negative feedback loop.

这个控制回路 更准确的叫"负反馈回路"

A negative feedback control loop has three key pieces.

负反馈回路 有三个重要部分

There’s a sensor, that measures things in the real world,

首先是一个传感器，可以测量现实中的东西

like water pressure, motor position, air temperature,

比如水压，马达位置，气温，

or whatever you’re trying to control.

或任何你想控制的东西

From this measurement, we calculate how far we are from

根据传感器，计算和目标值相差多大

where we want to be – the error.

得到一个"错误"

The error is then interpreted by a controller,

然后"控制器"会处理这个"错误"

which decides how to instruct the system to minimize that error.

决定怎么减小错误

Then, the system acts on the world though pumps, motors,

然后用泵，电机，加热元件，或其他物理组件来做出动作

heating elements, and other physical actuators.

然后用泵，电机，加热元件，或其他物理组件来做出动作

In tightly controlled environments, simple control loops, like this, work OK.

在严格控制的环境中，这种简单控制回路也够用了

But in many real world applications, things are a tad more complicated.

但在很多现实应用中，情况复杂得多

Imagine that our gripper is really heavy, and even when the control loop says to stop,

假设爪子很重，哪怕控制回路叫停了

momentum causes the gripper to overshoot the desired position.

惯性让爪子超过了预期位置

That would cause the control loop to take over again,

然后控制回路又开始运行

this time backing the gripper up.

叫爪子移动回去

A badly tuned control loop might overshoot and overshoot and overshoot,

一个糟糕的控制回路 可能会让爪子不断来回移动

and maybe even wobble forever.

甚至永远循环

To make matters worse, in real world settings,

更糟糕的是，现实世界中

there are typically external and variable forces acting on a robot,

机器人会受到各种外力影响

like friction, wind and items of different weight.

比如摩擦力，风，等等

To handle this gracefully, more sophisticated control logic is needed.

为了处理这些外力，我们需要更复杂的控制逻辑

A widely used control-loop, feedback mechanism is a

一个使用广泛的机制，有控制回路和反馈机制。

proportional–integral–derivative controller.

叫 "比例-积分-微分控制器"

That’s a bit of a mouthful, so people call them PID controllers.

这个有点绕口，所以一般简称 "PID控制器"

These used to be mechanical devices, but now it’s all done in software.

它以前是机械设备，现在全是纯软件了

Let’s imagine a robot that delivers coffee.

想象有一个机器人，端咖啡给客人

Its goal is to travel between customers at two meters per second,

设计目标是 每秒两米的速度在顾客间穿行

which has been determined to be the ideal speed

这个速度是理想速度

that’s both safe and expedient.

安全又合适

Of course, the environment doesn’t always cooperate.

当然，环境是会变化的

Sometimes there’s wind, and sometimes there's uphills and downhills

有时候有风，有时候有上坡下坡

and all sorts of things that affect the speed of the robot.

以及其他影响机器人速度的因素

So, it’s going to have to increase and decrease power

所以，给马达的动力要加大或减少，以保持目标速度

to its motors to maintain the desired speed.

所以，给马达的动力要加大或减少，以保持目标速度

Using the robot's speed sensor, we can keep track of its

用机器人的速度传感器，我们可以

actual speed and plot that alongside its desired speed.

把当前速度和目标速度画张图

PID controllers calculate three values from this data.

PID 控制器根据这些数据，算出3个值

First is the proportional value, which is the difference between

首先是"比例值"，就是"实际值"和"理想值"差多少

the desired value and the actual value

首先是"比例值"，就是"实际值"和"理想值"差多少

at the most recent instant in time or the present.

"实际值"可能有一定滞后，或者是实时的。

This is what our simpler control loop used before.

之前的简单控制回路，用的就是这个值

The bigger the gap between actual and desired,

"实际值"和"理想值"的差距越大，

the harder you'll push towards your target.

就越用力

In other words, it’s proportional control.

换句话说，它是"比例控制"的

Next, the integral value is computed,

接下来，算"积分值"

which is the sum of error over a window of time,

就是一段时间内 误差的总和

like the last few seconds.

比如最近几秒

This look back helps compensate for steady state errors,

帮助弥补误差

resulting from things like motoring up a long hill.

比如上坡时可能就会产生误差

If this value is large, it means proportional control is not enough,

如果这个值很大，说明比例控制不够，

and we have to push harder still.

要继续用力前进

Finally, there’s the derivative value,

最后有"导数值"

which is the rate of change between the desired and actual values.

是期望值与实际值之间的变化率

This helps account for possible future error,

有助于解决 未来可能出现的错误，

and is sometimes called "anticipatory control".

有时也叫"预期控制"

For example, if you are screaming in towards your goal too fast,

比如前进的太快

you'll need to ease up a little to prevent overshoot.

要稍微放松一点，避免冲过头

These three values are summed together, with different relative weights,

这三个值会一起使用，它们有不同权重

to produce a controller output that’s passed to the system.

然后用来控制系统

PID controllers are everywhere,

PID 控制器到处都是

from the cruise control in your car,

比如汽车里的巡航控制

to drones that automatically adjust their rotor speeds to maintain level flight,

无人机调整螺旋桨速度，以保持水平

as well as more exotic robots,

以及一些更奇怪的机器人，

like this one that balances on a ball to move around.

比如这个用球来平衡和移动的机器人

Advanced robots often require many control loops running in parallel,

更高级的机器人一般需要多个控制回路同时运行

working together, managing everything from robot balance to limb position.

来保持机器人平衡，调整肢体位置，等等

As we’ve discussed, control loops are responsible for

之前说过，控制回路负责

getting robot attributes like location to desired values.

把机器人的属性（比如当前位置）变成期望值

So, you may be wondering where these values come from.

你可能好奇这些值 是哪里来的

This is the responsibility of higher-level robot software,

这是更高层软件的责任

which plans and executes robot actions,

软件负责做出计划 并让机器人执行动作，

like plotting a path around sensed obstacles, or breaking down physical tasks,

比如制定一条路线来绕过障碍物，或者把任务分成一步步

like picking up a ball, into simple, sequential motions.

比如把拿起一个球，分解成一个个简单连续动作

Using these techniques, robots have racked up some impressive achievements

用这些技术，机器人已经取得不少令人印象深刻的成就

– they've been to the deepest depths of Earth's oceans

- 它们潜到了海洋最深处

and roved around on Mars for over a decade.

在火星上跑了十几年

But interestingly, lots of problems that are trivial for many humans

但有趣的是，许多对人类来说很简单的任务

have turned out to be devilishly difficult for robots:

对机器人很困难：

like walking on two legs, opening a door, picking up objects

比如两条腿走路，开门，拿东西时不要捏碎了

without crushing them, putting on a t-shirt, or petting a dog.

或是穿T恤，或是摸狗

These are tasks you may be able to do without thinking,

这些你可能想都不用想

but a supercomputer-powered robot fails at spectacularly.

但有超级计算机能力的机器人却做不到

These sorts of tasks are all active areas of robotics research.

机器人研究领域在全力解决这些问题

Artificial intelligence techniques,

我们前几集聊过的 人工智能

which we discussed a few episodes ago, are perhaps

我们前几集聊过的 人工智能

the most promising avenue to overcome these challenges.

最有可能解决这些问题

For example, Google has been running an experiment

例如，谷歌在进行一项实验

with a series of robotic arms that spend their days

让一堆机器人手臂把各种东西

moving miscellaneous objects from one box to another, learning from trial and error.

从一个盒子拿到另一个盒子，不断试错学习

After thousands of hours of practice, the robots had cut their error rate in half.

经过数千小时的练习，机器人把错误率降低了一半

Of course, unlike humans, robots can run twenty-four hours a day

不像人类，机器人可以24小时全天运行

and practice with many arms at the same time.

而且多个手臂同时练习

So, it may just be a matter of time until they become adept at grasping things.

所以机器人擅长抓东西只是时间问题

But, for the time being, toddlers can out-grasp them.

但现在，小婴儿都比机器人更会抓东西

One of the biggest and most visible robotic breakthrough

近年最大的突破之一

in recent years has been self-driving, autonomous cars.

是无人驾驶汽车

If you think about it, cars don’t have too many system inputs

如果你仔细想想，汽车没几个输入

– you can speed up or slow down, and you can steer left or right.

- 只是加速减速，左转右转

The tough part is sensing lanes, reading signs,

难的问题是 判断车道，理解路标

and anticipating and navigating traffic, pedestrians,

预测车流，车流中穿行，留心行人和骑自行车的。

bicyclists, and a whole host of obstacles.

以及各种障碍

In addition to being studded with proximity sensors,

车上布满了传感器

these robotic vehicles heavily rely

无人驾驶汽车非常依赖

on Computer Vision algorithms, which we discussed in Episode 35.

计算机视觉算法，我们在第35集讨论过

We’re also seeing the emergence of very primitive androids

现在也开始出现类人机器人

– robots that look and act like humans.

- 外貌和行为像人类的机器人

Arguably, we’re not close on either of those goals,

不过现在两个目标都没接近（外貌和行为）

as they tend to look pretty weird and act even weirder.

因为看起来一般怪怪的，行为也怪怪的.

At least we’ll always have Westworld.

但至少有《西部世界》可以看看

But anyway, these remain a tantalizing goal for roboticists

无论如何，对机器人研究者来说，把各种技术结合起来

combine many computer science topics

比如人工智能，计算机视觉和自然语言处理

we’ve touched on over the last few episodes, like artificial intelligence,

比如人工智能，计算机视觉和自然语言处理

computer vision and natural language processing.

来让机器人越来越像人，是个诱人的目标

As for why humans are so fascinated by

至于人类为什么如此着迷 做出和我们一样的机器人

creating artificial embodiments of ourselves.

至于人类为什么如此着迷 做出和我们一样的机器人

you'll have to go to Crash Course Philosophy for that.

你得去看《哲学速成课》

And for the foreseeable future,

在未来好一段时间里

realistic androids will continue to be the stuff of science fiction.

和人类一样的机器人 依然只能存在科幻小说里。

Militaries also have a great interest in robots –

军队也对机器人很有兴趣 -

they're not only replaceable, but can surpass humans

因为机器人可以替换，

in attributes like strength, endurance, attention, and accuracy.

而且力量，耐力，注意力，准确性可以远超人类

Bomb disposal robots and reconnaissance drones are fairly common today.

拆弹机器人和无人侦察机如今很常见

But fully autonomous, armed-to-the-teeth robots are slowly appearing,

但完全自主决定，全副武装的机器人也在慢慢出现

like the Samsung SGR-A1 sentry gun deployed by South Korea.

比如韩国的三星 SGR-A1 哨兵炮

Robots with the intelligence and capability to take human lives

有智力并且可以杀人的机器人

are called lethal autonomous weapons.

叫 "致命自主武器"

And they’re widely considered a complex and thorny issue.

这种武器是复杂又棘手的问题

Without doubt, these systems could save soldiers lives

毫无疑问，它们可以把士兵从战场带离 挽救生命

by taking them off the battlefield and out of harm’s way.

毫无疑问，它们可以把士兵从战场带离 挽救生命

It might even discourage war all together.

甚至阻止战争的发生

Though it’s worth noting that people said the same thing

值得注意的是 人们对炸药和核弹也说过一样的话

about dynamite and nuclear weapons.

值得注意的是 人们对炸药和核弹也说过一样的话

On the flip side, we might be creating ruthlessly

另一方面，我们可能会不小心创造出 \N 无情又高效的杀人机器

efficient killing machines that don't apply human judgment

没有人类般的判断力和同情心

or compassion to complex situations.

没有人类般的判断力和同情心

And the fog of war is about as complex and murky as they come.

战争的硝烟会变得更加黑暗和复杂

These robots would be taking orders and executing them

机器人会接受命令并高效执行

as efficiently as they can and sometimes

机器人会接受命令并高效执行

human orders turn out to be really bad.

但有时人类的命令是错的

This debate is going to continue for a long time,

这场辩论会持续很长时间，

and pundits on both sides will grow louder as robotic technology improves.

而且随着机器人技术的进步，两边的辩论会越来越激烈

It’s also an old debate –

这也是个老话题了

the danger was obvious to science fiction writer Isaac Asimov,

科幻作家 艾萨克·阿西莫夫 早预见了这种危险

who introduced a fictional "Three Laws of Robotics" in his 1942 short story "Runaround".

他在1942年短篇小说 Runaround 中写了"机器人三定律"

And then, later he added a zeroth rule.

之后又加了"定律0"

In short, it’s a code of conduct or moral compass for robots –

简单说 这些定律指导机器人的行为准则 或者说道德指南

guiding them to do no harm, especially to humans.

让机器人不要伤害，特别是不要伤害人类

It’s pretty inadequate for practical application and it leaves plenty of room for equivocation.

这些规则实践起来相当不足，并且有很多模糊的地方

But still, Asimov’s laws inspired a ton of science fiction and academic discussion,

但阿西莫夫三定律 激发了大量科幻小说讨论和学术讨论，

and today there are whole conferences on robot ethics.

如今有专门讨论机器人伦理的会议

Importantly, Asimov crafted his fictional rules

重要的是，阿西莫夫写这些虚构规则

as a way to push back on "Robot as a Menace" memes

是为了反对 "机器人都很邪恶" 这种常见描述

common in fiction from his childhood.

他童年读的小说里，这样的场景很常见

These were stories where robots went off the rails,

机器人脱离控制，然后伤害甚至毁灭创造者

harming or even destroying their creators in the process.

机器人脱离控制，然后伤害甚至毁灭创造者

Asimov, on the other hand, envisioned robots as useful,

阿西莫夫认为 机器人有用，可靠，甚至可以让人喜爱

reliable, and even loveable machines.

阿西莫夫认为 机器人有用，可靠，甚至可以让人喜爱

And it’s this duality I want to leave you thinking about today.

我想让你思考这种两面性

Like many of the technologies we’ve discussed throughout this series,

我们讨论过的许多技术，有好的一面也有坏的一面

there are benevolent and malicious uses.

我们讨论过的许多技术，有好的一面也有坏的一面

Our job is to carefully reflect on computing's potential and peril,

我们要认真思考计算机的潜力和危害

and wield our inventive talents to improve the state of the world.

来改善这个世界

And robots are one of the most potent reminders of this responsibility.

而机器人最能提醒我们这一点了

I’ll see you next week.

我 们 下 周 见

Hi, I’m Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne \N 欢迎收看计算机科学速成课！

So, over the course of this series,

在这个系列中，

we’ve focused almost exclusively on computers –

我们聊的话题几乎全是计算机-比如电路和算法

the circuits and algorithms that make them tick.

我们聊的话题几乎全是计算机-比如电路和算法

Because...this is Crash Course Computer Science.

毕竟这是\*计算机\*速成课

But ultimately, computers are tools employed by people.

但归根结底，计算机只是给人用的工具

And humans are… well… messy.

而人类有点... 乱

We haven’t been designed by human engineers from the ground up

人类不是被工程师设计的，没有具体性能规格

with known performance specifications.

人类不是被工程师设计的，没有具体性能规格

We can be logical one moment and irrational the next.

我们一会儿是理性的，一会儿是不理性的

Have you ever gotten angry at your navigation system? Surfed wikipedia aimlessly?

你有没有对导航生过气？或是漫无目的的刷维基百科？

Begged your internet browser to load faster?

求浏览器加载快点？

Nicknamed your roomba?

给扫地机器人取名？

These behaviors are quintessentially human!

这些是人类行为！

To build computer systems that are useful, usable and enjoyable,

为了做出使用愉快的计算机

we need to understand the strengths and weaknesses of both computers and humans.

我们需要了解计算机和人类的优缺点

And for this reason, when good system designers are creating software,

优秀的系统设计师在创造软件时

they employ social, cognitive, behavioral, and perceptual psychology principles.

会运用社会心理学，认知心理学，\N 行为心理学，感知心理学的原理

No doubt you’ve encountered a physical or computer interface

你肯定见过难用的物理界面/计算机界面 \N 阻碍你做事

that was frustrating to use, impeding your progress.

你肯定见过难用的物理界面/计算机界面 \N 阻碍你做事

Maybe it was so badly designed that you couldn’t figure it out and just gave up.

甚至糟糕到放弃使用

That interface had poor usability.

那个界面的"易用度"很差

Usability is the degree to which a human-made artifact – like software

"易用度"指的是人造物体，比如软件 \N 达到目的的效率有多高

– can be used to achieve an objective effectively and efficiently.

"易用度"指的是人造物体，比如软件 \N 达到目的的效率有多高

To facilitate human work, we need to understand humans

为了帮助人类工作，我们需要了解人类 \N - 怎么看，思考，反应和互动

- from how they see and think, to how they react and interact.

为了帮助人类工作，我们需要了解人类 \N - 怎么看，思考，反应和互动

For instance, the human visual system has been well studied by Psychologists.

举个例子，心理学家已经对 \N 人类的视觉系统做了全面的研究

Like, we know that people are good at ordering intensities of colors.

我们知道人类擅长给颜色强度排序

Here are three.

这里有三个颜色

Can you arrange these from lightest to darkest?

你能从浅色到深色排序吗？

You probably don’t have to think too much about it.

你可以轻易做到

Because of this innate ability, color intensity is a great choice

所以颜色强度很适合显示连续值

for displaying data with continuous values.

所以颜色强度很适合显示连续值

On the other hand, humans are terrible at ordering colors.

另一方面，人类很不擅长排序颜色

Here’s another example for you to put in order.

这是另一个例子

is orange before blue, or after blue?

把橙色放到蓝色前面还是后面？绿色放哪里？

Where does green go?

把橙色放到蓝色前面还是后面？绿色放哪里？

You might be thinking we could order this by wavelength of light,

你可能想通过光的波长排序 \N 就像彩虹一样，但这样太累了

like a rainbow, but that’s a lot more to think about.

你可能想通过光的波长排序 \N 就像彩虹一样，但这样太累了

Most people are going to be much slower and error-prone at ordering.

大部分人会很慢而且容易出错

Because of this innate ineptitude of your visual system,

由于视觉系统天生是这样

displaying continuous data using colors can be a disastrous design choice.

所以用不同颜色显示连续性数据，是个糟糕的选择

You’ll find yourself constantly referring back to a color legend to compare items.

你得经常看表格来对比数据

However, colors are perfect for when the data is discrete with no ordering,

然而，如果数据没有顺序，用不同颜色就很合适

like categorical data.

比如分类数据

This might seem obvious, but you’d be amazed at

也许这些看起来很明显 \N 但你会惊讶有多少设计把这些基本事情搞错

how many interfaces get basic things like this wrong.

也许这些看起来很明显 \N 但你会惊讶有多少设计把这些基本事情搞错

Beyond visual perception, understanding human cognition helps us

除了视觉 \N 理解人类的认知系统能帮我们设计更好的界面

design interfaces that align with how the mind works.

除了视觉 \N 理解人类的认知系统能帮我们设计更好的界面

Like, humans can read, remember and process information more effectively

比如，如果信息分块了 \N 会更容易读，更容易记

when it's chunked–that is, when items are put together into small, meaningful groups.

分块是指把信息分成更小，更有意义的块

Humans can generally juggle seven items, plus-or-minus two, in short-term memory.

人类的短期记忆能记住5到9个东西

To be conservative, we typically see groupings of five or less.

保守一点，分组一般是5个或更少

That’s why telephone numbers are broken into chunks, like 317, 555, 3897.

所以电话号码一般分块，比如 317-555-3897

Instead of being ten individual digits that we’d likely forget, it’s three chunks,

10个连续数可能会忘，分成3块更好记

which we can handle better.

10个连续数可能会忘，分成3块更好记

From a computer's standpoint, this needlessly takes more time and space,

从计算机的角度来看，分块更费时费空间

so it's less efficient.

效率更低

But, it’s way more efficient for us humans –

但这对人类更有效率

a tradeoff we almost always make in our favor,

- 碰到这种抉择时，我们总是以人类优先

since we’re the ones running the show...for now.

现在我们还是老大.. 暂时啦

Chunking has been applied to computer interfaces for things

界面设计用了分块 \N 比如下拉菜单 和带按钮的菜单栏

like drop-down menu items and menu bars with buttons.

界面设计用了分块 \N 比如下拉菜单 和带按钮的菜单栏

It’d be more efficient for computers to just pack all those together, edge to edge

对电脑来说，全部挤在一起更有效率

– it’s wasted memory and screen real estate.

分块浪费内存 浪费屏幕

But designing interfaces in this way makes them much easier

但这样设计更容易扫视，记住和访问

to visually scan, remember and access.

但这样设计更容易扫视，记住和访问

Another central concept used in interface design is affordances.

界面设计中另一个重点概念是"直观功能"

According to Don Norman, who popularized the term in computing,

Don Norman 让这个词在计算机界流行起来，根据他的说法

"affordances provide strong clues to the operations of things.

"直观功能 为如何操作物体提供线索

Plates are for pushing.

平板用来推

Knobs are for turning.

旋钮用来转

Slots are for inserting things into.

插槽用来插东西

[...] When affordances are taken advantage of, the user knows what to do just by looking:

[...]直观功能做的好，用户只需要看一眼就知道怎么做：

no picture, label, or instruction needed."

不需要图片，标签或指南来说明"

If you’ve ever tried to pull a door handle, only to realize that you have to push it open,

如果你拉过门把手打不开，然后意识到要推开才对

you’ve discovered a broken affordance.

那么你发现了一个坏掉的"直观功能"

On the other hand, a door plate is a better design

平板是更好的设计

because it only gives you the option to push.

因为只能推开

Doors are pretty straightforward – if you need to put written instructions on them,

门是简单的东西，如果你要贴指示让人们明白怎么用.

you should probably go back to the drawing board.

那么也许你应该重新设计

Affordances are used extensively in graphical user interfaces,

"直观功能"广泛用于图形界面

which we discussed in episode 26.

我们在第26集讨论过

It’s one of the reasons why computers became so much easier to use than with command lines.

这是图形界面比命令行更容易用的原因之一

You don’t have to guess what things on-screen are clickable, because they look like buttons.

你不用猜测屏幕上什么东西是可点的，\N 可点的会看起来像按钮

They pop out, just waiting for you to press them!

他们弹出来，只是等着你压他们！

One of my favorite affordances, which suggests to users that an on-screen element is draggable,

我最喜欢的"直观功能"之一，是向用户表明元素是可拖动的

is knurling – that texture added to objects

"滚花" - 一种视觉纹理

to improve grip and show you where to best grab them.

告诉用户哪里可以拖动

This idea and pattern was borrowed from real world physical tools.

这个点子来自现实世界中的工具

Related to the concept of affordances is the psychology of recognition vs recall.

和"直观功能"相关的一个心理学概念是 "认出与回想"

You know this effect well from tests –

如果你考过试，肯定感受过这个

it's why multiple choice questions are easier than fill-in-the-blank ones.

这就是为什么选择题比填空题容易

In general, human memory is much better when it’s triggered by a sensory cue,

一般来说，用感觉来触发记忆会容易得多

like a word, picture or sound.

比如文字，图片或声音

That’s why interfaces use icons – pictorial representations of functions

所以我们用图标代表功能

– like a trash can for where files go to be deleted.

- 比如"垃圾桶"图标 代表里面放着被删除的文件

We don’t have to recall what that icon does, we just have to recognise the icon.

我们不用去回想图标的功能是什么，只要能认出来就行了

This was also a huge improvement over command line interfaces,

比命令行好得多

where you had to rely on your memory for what commands to use.

命令行得依靠记忆来输命令

Do I have to type "delete", or "remove", or... "trash", or… shoot, it could be anything!

到底是输入"删除""移除""垃圾"还是"射出"？\N 可能是任何命令！

It’s actually "rm" in linux,

顺带一说，在 Linux 里删除文件的命令是 "rm"

but anyway, making everything easy to discover and learn sometimes means slow to access,

回到正题，\N 让所有菜单选项好找好记，有时候意味着用的时候会慢一些

which conflicts with another psychology concept: expertise.

这与另一个心理学概念冲突："专业知识"

As you gain experience with interfaces, you get faster,

当你用界面熟悉之后，速度会更快一些

building mental models of how to do things efficiently.

建立如何高效完成事情的"心理模型"

So, good interfaces should offer multiple paths to accomplish goals.

所以 好的界面应该提供多种方法来实现目标

A great example of this is copy and paste, which can be found in the edit dropdown menu

一个好例子是复制粘贴，可以在"编辑"的下拉菜单中找到

of word processors, and is also triggered with keyboard shortcuts.

也可以用快捷键

One approach caters to novices, while the other caters to experts, slowing down neither.

一种适合新手，一种适合专家，两者都不耽误

So, you can have your cake and eat it too!

鱼和熊掌兼得！

In addition to making humans more efficient,

除了让人类做事更高效，

we'd also like computers to be emotionally intelligent –

我们也希望电脑能有一点情商

adapting their behavior to respond appropriately

能根据用户的状态做出合适地反应

to their users' emotional state – also called affect.

能根据用户的状态做出合适地反应

That could make experiences more empathetic, enjoyable, or even delightful.

让使用电脑更加愉快

This vision was articulated by Rosalind Picard in her 1995 paper on Affective Computing,

Rosalind Picard 在 1995 年关于"情感计算"的论文中，阐述了这一愿景

which kickstarted an interdisciplinary field combining aspects

这篇论文开创了心理学，社会科学和计算机科学的跨学科结合

of psychology, social and computer sciences.

这篇论文开创了心理学，社会科学和计算机科学的跨学科结合

It spurred work on computing systems that could recognize,

促进了让计算机理解人类情感的研究

interpret, simulate and alter human affect.

促进了让计算机理解人类情感的研究

This was a huge deal, because we know emotion influences cognition and perception

这很重要，因为情绪会影响日常活动

in everyday tasks like learning, communication, and decision making.

比如学习，沟通和决策

Affect-aware systems use sensors, sometimes worn, that capture things like speech and

情感系统会用传感器，录声音，

video of the face, as well as biometrics, like sweatiness and heart rate.

录像（你的脸）以及生物指标，比如出汗和心率

This multimodal sensor data is used in conjunction with computational models that represent how

得到的数据和计算模型结合使用

people develop and express affective states, like happiness and frustration,

模型会写明人类如何表达情感，怎么是快乐 怎么是沮丧

and social states, like friendship and trust.

以及社交状态，比如友谊和信任

These models estimate the likelihood of a user being in a particular state,

模型会估算用户的情绪

and figure out how to best respond to that state,

以及怎样以最好的回应用户

in order to achieve the goals of the system.

以达到目标

This might be to calm the user down, build trust, or help them get their homework done.

比如让用户冷静下来，建立信任，或帮忙完成作业

A study, looking at user affect, was conducted by Facebook in 2012.

Facebook 在 2012 年进行了一项"影响用户"的研究

For one week, data scientists altered the content

数据科学家在一个星期内

on hundreds of thousands of users' feeds.

修改了很多用户 时间线上显示的内容

Some people were shown more items with positive content,

有些人会看到更多积极向上的内容

while others were presented with more negative content.

有些人会看到更多负面消极的内容

The researchers analyzed people's posts during that week,

研究人员分析了那一周内人们的发帖

and found that users who were shown more positive content,

发现看到积极向上内容的用户，

tended to also post more positive content.

发的帖子往往更正面

On the other hand, users who saw more negative content, tended to have more negative posts.

另一方面，看到负面内容的用户，发的内容也更负面

Clearly, what Facebook and other services show you

显然，Facebook和其他网站向你展示的内容

can absolutely have an affect on you.

绝对会对你有影响

As gatekeepers of content, that’s a huge opportunity and responsibility.

作为信息的守门人，这是巨大的机会 同时也是责任

Which is why this study ended up being pretty controversial.

研究结果相当有争议性.

Also, it raises some interesting questions about

而且它还产生了一个有趣的问题：

how computer programs should respond to human communication.

计算机程序如何回应人类

If the user is being negative, maybe the computer shouldn’t be

如果用户的情绪比较负面，也许电脑不应该

annoying by responding in a cheery, upbeat manner.

以一种烦人的 "你要振作起来呀" 的态度回答问题.

Or, maybe the computer should attempt to evoke a positive response,

或者，也许电脑应该试着积极正面的回应用户

even if it's a bit awkward.

即使这有点尴尬.

The "correct" behavior is very much an open research question.

什么行为是"正确的"，是个开放性的研究问题

Speaking of Facebook, it’s a great example of computer-mediated communication, or CMC,

既然说到Facebook，\N 这是一个"以计算机为媒介沟通"的好例子，简称 "CMC"

another large field of research.

也是一个很大的研究领域

This includes synchronous communication – like video calls, where all participants are online

这包括同步通信 - 所有参与者同时在线进行视频通话

simultaneously – as well as asynchronous communication – like tweets, emails, and

以及异步通信 - 比如推特，邮件，

text messages, where people respond whenever they can or want.

短信，人们可以随时随地回复信息

Researchers study things like the use of emoticons, rules such as turn-taking,

研究人员还研究用户怎么用表情包，怎么轮换发言，

and language used in different communication channels.

以及用不同沟通渠道时，用词有什么区别.

One interesting finding is that people exhibit higher levels of self-disclosure

一个有趣的发现是，比起面对面沟通，

– that is, reveal personal information –in computer-mediated conversations,

人们更愿意在网上透露自己的信息

as opposed to face-to-face interactions.

人们更愿意在网上透露自己的信息

So if you want to build a system that knows how many hours a user truly spent

所以如果想知道用户 \N 真正花了多少小时看"大英烘培大赛"(电视节目)

watching The Great British Bakeoff, it might be better to build a chatbot

比起做个带脸的虚拟助理 \N 做 聊天机器人 是个更好的选择

than a virtual agent with a face.

比起做个带脸的虚拟助理 \N 做 聊天机器人 是个更好的选择

Psychology research has also demonstrated that eye gaze is

心理学研究也表明，如果想说服，讲课，或引起注意 \N 眼神注视非常重要

extremely important in persuading, teaching and getting people's attention.

心理学研究也表明，如果想说服，讲课，或引起注意 \N 眼神注视非常重要

Looking at others while talking is called mutual gaze.

在谈话时看着别人叫 相互凝视

This has been shown to boost engagement and help achieve the goals of a conversation,

这被证明可以促进参与感 帮助实现谈话目标，

whether that’s learning, making a friend, or closing a business deal.

不管是学习，交朋友，还是谈生意

In settings like a videotaped lecture, the instructor rarely, if ever, looks into the

在录像讲座中，老师很少直视相机 \N 一般是看在场学生

camera, and instead generally looks at the students who are physically present.

在录像讲座中，老师很少直视相机 \N 一般是看在场学生

That’s ok for them, but it means people who

对他们没问题，但这会让在线看视频的人 没什么参与感

watch the lectures online have reduced engagement.

对他们没问题，但这会让在线看视频的人 没什么参与感

In response, researchers have developed computer vision

为此，研究人员开发了计算机视觉和图形软件 \N 来纠正头部和眼睛

and graphics software that can warp the head and eyes,

为此，研究人员开发了计算机视觉和图形软件 \N 来纠正头部和眼睛

making it appear as though the instructor is looking into the camera

视频时会觉得对方在直视摄像头，看着他们

– right at the remote viewer.

视频时会觉得对方在直视摄像头，看着他们

This technique is called augmented gaze.

这叫"增强凝视"

Similar techniques have also been applied to video conference calls, to correct for

类似技术也用于视频会议

the placement of webcams, which are almost always located above screens.

纠正摄像头位置，因为摄像头几乎总在屏幕上方

Since you’re typically looking at the video of your conversation partner,

因为你一般会盯着屏幕上的另一方 \N 而不是盯着摄像头

rather than directly into the webcam,

因为你一般会盯着屏幕上的另一方 \N 而不是盯着摄像头

you'll always appear to them as though you're looking downwards –

所以视频里看起来像在向下看

breaking mutual gaze – which can create all kinds of

没有相互凝视 - 这会导致各种不幸的副作用，比如权力不平衡

unfortunate social side effects, like a power imbalance.

没有相互凝视 - 这会导致各种不幸的副作用，比如权力不平衡

Fortunately, this can be corrected digitally, and appear to participants

幸运的是 可以用软件修正

as though you're lovingly gazing into their eyes.

看起来像在凝视着对方的眼睛

Humans also love anthropomorphizing objects, and computers are no exception,

人类也喜欢"拟人化"的物体，对计算机也不例外

especially if they move, like our Robots from last episode.

特别是会动的计算机，比如上集说的机器人

Beyond industrial uses that prevailed over the last century,

在过去一个世纪，除了工业用途机器人

robots are used increasingly in medical, education, and entertainment settings,

有越来越多机器人用于医疗，教育和娱乐 \N 它们经常和人类互动

where they frequently interact with humans.

有越来越多机器人用于医疗，教育和娱乐 \N 它们经常和人类互动

Human-Robot Interaction – or HRI

人机交互，简称 HRI

– is a field dedicated to studying these interactions,

- 是一个研究人类和机器人交互的领域，

like how people perceive different robots behaviors and forms,

比如人类如何感受 机器人的不同形式和不同行为

or how robots can interpret human social cues to blend in and not be super awkward.

或是机器人如何明白人类暗示来社交，而不是尴尬的互动

As we discussed last episode, there’s an ongoing quest to make

正如上集说的，我们有追求

robots as human-like in their appearance and interactions as possible.

把机器人的外表和行为，做得尽可能像人一样

When engineers first made robots in the 1940s and 50s, they didn’t look very human at all.

工程师在1940 1950年代刚开始做机器人时，\N 看起来完全不像人

They were almost exclusively industrial machines with no human-likeness.

是完完全全的工业机器

Over time, engineers got better and better at making human-like robots

随着时间的推移，工程师越来越擅长做类人机器人

– they gained heads and walked around on two legs,

它们有头，而且用两条腿走路

but… they couldn't exactly go to restaurants and masquerade as humans.

但它们做不到伪装成人类去餐馆点餐

As people pushed closer and closer to human likeness,

随着机器人可以做得越来越像人类

replacing cameras with artificial eyeballs, and covering metal chassis with synthetic flesh,

用人造眼球代替摄像头，用人工肌肉盖住金属骨架

things started to get a bit... uncanny...

事情会开始变得有些.. 奇怪.. \N 引起一种怪异不安的感觉

eliciting an eerie and unsettling feeling.

事情会开始变得有些.. 奇怪.. \N 引起一种怪异不安的感觉

This dip in realism between almost-human and actually-human became known as the uncanny valley.

这个"几乎像人类"和"真的人类"之间的小曲线，叫 "恐怖谷"

There’s debate over whether robots should act like humans too.

对于机器人是否应该有人类一样的行为，也存在争议

Lots of evidence already suggests that even if robots don’t act like us,

很多证据表明，即使机器人的行为不像人类

people will treat them as though they know our social conventions.

人类也会用社交习俗对待它们

And when they violate these rules – such as not apologizing if they cut in front of

而当机器人违反习俗时 - 比如插队或踩了脚不道歉 \N 人们会很生气！

you or roll over your foot – people get really mad!

而当机器人违反习俗时 - 比如插队或踩了脚不道歉 \N 人们会很生气！

Without a doubt, psychology and computer science are a potent combination,

毫无疑问，心理学+计算机科学是强大的组合

and have tremendous potential to affect our everyday lives.

可以影响日常生活的巨大潜力

Which leaves us with a lot of question, like you might lie to your laptop,

这也带来了很多开放式问题，比如你可能会对计算机撒谎

but should your laptop lie to you?

但计算机应不应该对你撒谎？

What if it makes you more efficient or happy?

如果撒谎能让你更高效更快乐呢？

Or should social media companies curate the content they show you to

或社交媒体公司 \N 是否应该精心挑选展示给你的内容

make you stay on their site longer to make you buy more products?

让你在网站上多待一会儿，买更多东西？

They do by the way.

顺带一说，他们的确有这样做

These types of ethical considerations aren’t easy to answer, but psychology can at least

这类道德问题不容易回答，但心理学至少可以

help us understand the effects and implications of design choices in our computing systems.

帮助我们理解不同选择 带来的影响和意义

But, on the positive side, understanding the psychology behind design

但从积极的方面来说，了解设计背后的心理学

might lead to increased accessibility.

能增加易用性

A greater number of people can understand and use computers

让更多人可以明白和使用电脑，如今计算机比以往更加直观

now that they're more intuitive than ever.

让更多人可以明白和使用电脑，如今计算机比以往更加直观

Conference calls and virtual classrooms are becoming more agreeable experiences.

线上会议和虚拟教室的体验越来越好

As robot technology continues to improve, the population

随着机器人技术不断提高，互动也会越来越舒适

will grow more comfortable in those interactions.

随着机器人技术不断提高，互动也会越来越舒适

Plus, thanks to psychology, we can all bond over our love of knurling.

另外，感谢心理学，让我们能分享对"滚花"的热爱

I’ll see you next week.

我们下周见

Hi, I’m Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne \N 欢迎收看计算机科学速成课！\N

One of the most dramatic changes enabled by computing technology

计算机带来的最大改变之一 \N 是信息的创造和传播能力

has been the creation and widespread availability of information.

计算机带来的最大改变之一 \N 是信息的创造和传播能力

There are currently 1.3 billion websites on the internet.

目前有13亿个网站在互联网上

Wikipedia alone has five million English language articles,

仅维基百科就有500万篇英文文章

spanning everything from the Dancing Plague of 1518

涵盖从"1518年的舞蹈瘟疫"

to proper toilet paper roll orientation.

到"正确的纸卷方向"

Every day, Google serves up four billion searches to access this information.

每天，Google提供40亿次搜索来访问这些信息

And every minute, 3.5 million videos are viewed on Youtube,

Youtube上每分钟有350万个视频被观看.

and 400 hours of NEW video get uploaded by users.

每分钟用户上传400小时的新视频

Lots of these views are people watching Gangnam Style and Despacito.

很多观看量都是 Gangnam Style 和 Despacito

But another large percentage could be considered educational,

但剩下的 大部分是教育型内容

like what you’re doing right now.

就像你现在看的这个.

This amazing treasure trove of information can be accessed

如今只要手机上点几下 就能访问到这些宝藏

with just a few taps on your smartphone.

如今只要手机上点几下 就能访问到这些宝藏

Anywhere, anytime.

任何时间，任何地点

But, having information available isn’t the same as learning from it.

但能获取到信息和学习不是一回事

To be clear, we here at Crash Course we are big fans of interactive in-class learning,

先说清楚，我们 Crash Course 喜欢互动式课堂学习

directed conversations, and hands-on experiences as powerful tools for learning.

课上提问，以及上手实践，它们是很棒的学习途径

But we also believe in the additive power of educational technology

但我们也相信教育型技术在课内课外带来的帮助

both inside and outside the classroom.

但我们也相信教育型技术在课内课外带来的帮助

So today we’re going to go a little meta,

今天我们要在这个教育型视频里 \N 聊教育型科技

and talk specifically about how computer science

具体讲解计算机怎么帮助我们学习

can support learning with educational technology.

具体讲解计算机怎么帮助我们学习

Technology, from paper and pencil to recent machine-learning-based intelligent systems,

从纸和笔 到用机器学习的智能系统，

has been supporting education for millennia -

科技几千年来一直在辅助教育

even as early as humans drawing cave paintings

甚至早期人类 在洞穴里画狩猎场景也是为了后代

to record hunting scenes for posterity.

甚至早期人类 在洞穴里画狩猎场景也是为了后代

Teaching people at a distance has long been a driver of educational technology.

远距离教育一直推动着教育科技的发展

For example, around 50 CE, St. Paul was sending epistles

例如公元前50年左右，圣保罗就发书信 \N 给亚洲设立的新教堂提供宗教课程

that offered lessons on religious teachings

例如公元前50年左右，圣保罗就发书信 \N 给亚洲设立的新教堂提供宗教课程

for new churches being set up in Asia.

例如公元前50年左右，圣保罗就发书信 \N 给亚洲设立的新教堂提供宗教课程

Since then, several major waves of technological advances

从那以后，有几大技术浪潮，自称要改变教育

have each promised to revolutionize education,

从那以后，有几大技术浪潮，自称要改变教育

from radio and television, to DVDs and laserdiscs.

从广播和电视，到DVD和光碟

In fact, as far back as 1913, Thomas Edison predicted,

事实上，在1913年 托马斯·爱迪生 预测说

"Books will soon be obsolete in the schools…

"书籍很快会过时.. 用影片来教授所有知识是可能的

It is possible to teach every branch of human knowledge with the motion picture.

"书籍很快会过时.. 用影片来教授所有知识是可能的

Our school system will be completely changed in the next ten years."

学校体系将在未来十年彻底改变"

Of course, you know that didn’t happen.

当然，他的预测没有成真

But distributing educational materials in formats like video has become more and more popular.

但发布教育视频变得越来越流行

Before we discuss what educational technology research can do for you,

在讨论教育技术可以帮你做什么之前

there are some simple things research has shown you can do,

有研究表明 有些简单事情 \N 可以显著提高学习效率

while watching an educational video like this one,

有研究表明 有些简单事情 \N 可以显著提高学习效率

significantly increase what you learn and retain.

有研究表明 有些简单事情 \N 可以显著提高学习效率

First, video is naturally adjustable, so make sure the pacing is right for you,

1. 把速度调整到适合你，YouTube 的速度设置在右下角

by using the video speed controls.

1. 把速度调整到适合你，YouTube 的速度设置在右下角

On YouTube, you can do that in the right hand corner of the screen.

1. 把速度调整到适合你，YouTube 的速度设置在右下角

You should be able to understand the video and have enough time to reflect on the content.

让你能理解视频 有足够的时间思考

Second, pause!

2. 暂停！在困难的部分暂停

You learn more if you stop the video at the difficult parts.

2. 暂停！在困难的部分暂停

When you do, ask yourself questions about what you’ve watched, and see if you can answer.

问自己一些问题，看能不能回答

Or ask yourself questions about what might be coming up next,

或想想视频接下来可能讲什么 \N 然后继续播放，看猜对没有

and then play the video to see if you’re right.

或想想视频接下来可能讲什么 \N 然后继续播放，看猜对没有

Third, try any examples or exercises that are presented in the video on your own.

3. 做视频中的提供的练习

Even if you aren’t a programmer, write pseudocode on paper,

即使不是程序员，你也可以在纸上写伪代码，或试试学编程

and maybe even give coding a try.

即使不是程序员，你也可以在纸上写伪代码，或试试学编程

Active learning techniques like these

这些主动学习的技巧已被证明 \N 可以把学习效率提升10倍或以上

have been shown to increase learning by a factor of ten.

这些主动学习的技巧已被证明 \N 可以把学习效率提升10倍或以上

And if you want more information like this - we’ve got a whole course on it here.

如果想学学习技巧，有整个系列专门讲这个

The idea of video as a way to spread quality education

把高质量教育内容做成视频传播 \N 在过去一个世纪吸引了很多人

has appealed to a lot of people over the last century.

把高质量教育内容做成视频传播 \N 在过去一个世纪吸引了很多人

What’s just the latest incarnation of this idea

这个老想法的新化身

came in the form of Massive Open Online Courses, or MOOCs.

以"大型开放式在线课程"（MOOC）的形式出现

In fact, the New York Times declared 2012 the Year of the MOOC!

纽约时报宣称 2012 年是 MOOC 年！

A lot of the early forms were just videos of lectures from famous professors.

很多早期视频 直接录制著名教授上课

But for a while, some people thought this might mean the end of universities as we know them.

有段时间，有些人以为大学要终结了

Whether you were worried about this idea or excited by it,

不管你是担心还是开心，这暂时还没成为现实

that future also hasn’t really come to pass

不管你是担心还是开心，这暂时还没成为现实

and most of the hype has dissipated.

现在热度也淡去了

This is probably mostly because when you try to scale up learning

这可能是因为加大规模时 同时教百万名学生

using technology to include millions of students simultaneously

这可能是因为加大规模时 同时教百万名学生

with small numbers of instructional staff - or even none

但老师数量很少，甚至完全没有老师

- you run into a lot of problems.

- 会遇到很多问题

Fortunately, these problems have intrigued computer scientists and more specifically,

幸运的是，这引起了计算机科学家，\N 或具体一点 "教育科技家"的兴趣

educational technologists, who are finding ways to solve them.

他们在想办法解决这些问题

For example, effective learning involves getting timely and relevant feedback

比如，为了有效学习，学生要及时获得反馈

but how do you give good feedback

但如果有几百万学生，只有一名老师，

when you have millions of learners and only one teacher?

怎么提供好的反馈？

For that matter, how does a teacher grade a million assignments?

一个老师怎么给一百万份作业打成绩？

Solving many of these problems means creating hybrid, human-technology systems.

为了解决问题，很多时候需要把科技和人类都用上

A useful, but controversial insight,

一种有用 但有些争议的做法是

was that students could be a great resource to give each other feedback.

学生互相之间提供反馈

Unfortunately, they’re often pretty bad at doing so –

不幸的是，学生一般做不好

they’re neither experts in the subject matter, nor teachers.

他们既不是专家也不是老师

However, we can support their efforts with technology.

但我们可以用技术来帮助他们

Like, by using algorithms, we can match perfect learning partners together,

比如通过算法，从数百万个选择里 \N 匹配出最完美的学习伙伴

out of potentially millions of groupings.

比如通过算法，从数百万个选择里 \N 匹配出最完美的学习伙伴

Also, parts of the grading can be done with automated systems while humans do the rest.

另外，有些部分可以机器打分，剩下的让人类打分

For instance, computer algorithms that grade the

例如，给 SAT 写作部分打分的电脑算法

writing portions of the SATs have been found to be

已被证实和人工打分一样准确

just as accurate as humans hired to grade them by hand.

已被证实和人工打分一样准确

Other algorithms are being developed that provide personalized learning experiences,

还有些算法提供个性化学习体验

much like Netflix’s personalized movie recommendations or Google’s personalized search results.

类似于 Netflix 的电影推荐 \N 或 Google 的个性化搜索结果

To achieve this, the software needs to understand what a learner knows and doesn’t know.

为了个性化推荐，\N 软件需要了解用户知道什么，不知道什么

With that understanding, the software can present the right material, at the right time,

在正确的时间提供正确的资料，

to give each particular learner practice on the things that are hardest for them,

让用户练习没理解的难的部分

rather than what they’re already good at.

而不是给出用户已经学会的内容

Such systems – most often powered by Artificial Intelligence –

这种系统一般用 AI 实现

are broadly called Intelligent Tutoring Systems.

泛称叫法是"智能辅导系统"

Let’s break down a hypothetical system that follows common conventions.

我们现在讲一个假想的辅导系统

So, imagine a student is working on this algebra problem in our hypothetical tutoring software.

假设学生在这个假想的辅导系统中，研究一个代数问题

The correct next step to solve it, is to subtract both sides by 7.

正确的下一步是两边-7

The knowledge required to do this step can be represented by something called a production rule.

我们可以用 "判断规则" 来表示这一步

These describe procedures as IF-THEN statements.

用 IF-THEN 语句来描述

The pseudo code of a production rule for this step would say

伪代码是

IF there is a constant on the same side as the variable,

\*如果\* 变量和常数在同一边

THEN subtract that constant from both sides.

\*那么\* 两侧都减去这个常数

The cool thing about production rules is that they can also be used

"判断规则" 酷的地方是也可以用来

to represent common mistakes a student might make.

代表学生的常犯错误

These production rules are called "buggy rules".

这些"判断规则"叫"错误规则"

For example, instead of subtracting the constant,

例如，学生可能不去减常数

the student might mistakenly try to subtract the coefficient.

而是去减系数

No can do!

这不行！

It’s totally possible that multiple competing production rules

学生做完一个步骤后可能触发多个"判断规则"

are triggered after a student completes a step –

学生做完一个步骤后可能触发多个"判断规则"

it may not be entirely clear what misconception has led to a student’s answer.

系统不能完全弄清 是什么原因让学生选了那个答案

So, production rules are combined with an algorithm that selects the most likely one.

所以"判断规则"会和算法结合使用，判断可能原因

That way, the student can be given a helpful piece of feedback.

让学生得到有用反馈

These production rules, and the selection algorithm,

"判断规则"+选择算法，组合在一起成为 "域模型"

combine to form what’s called a Domain Model,

"判断规则"+选择算法，组合在一起成为 "域模型"

which is a formal representation of the knowledge,

它给知识，解决步骤和一门学科 比如代数，\N 用一种"正式写法"来表示

procedures and skills of a particular discipline - like algebra.

它给知识，解决步骤和一门学科 比如代数，\N 用一种"正式写法"来表示

Domain models can be used to assist learners on any individual problem,

域模型可以用来 帮助学习者解决特定问题

but they’re insufficient for helping learners move through a whole curriculum

但它无法带着学习者 \N 以正确顺序搞定整个学科该上的所有课程

because they don’t track any progress over time.

因为域模型不记录进度

For that, intelligent tutoring systems build and maintain a student model

因此智能辅导系统 负责创建和维护学生模型

– one that tracks, among other things, what production rules a student has mastered,

- 记录学生已经掌握的判断规则

and where they still need practice.

以及还需练习的生疏部分

This is exactly what we need to properly personalize the tutor.

这正是个性化辅导系统需要的。

That doesn’t sound so hard,

听起来好像不难，

but it’s actually a big challenge to figure out what a student knows and doesn’t know

但只靠学生对一些问题的回答，\N 来弄清学生知道什么，不知道什么，是很大的挑战

based only on their answers to problems.

但只靠学生对一些问题的回答，\N 来弄清学生知道什么，不知道什么，是很大的挑战

A common technique for figuring this out is Bayesian knowledge tracing.

"贝叶斯知识追踪" 常用来解决这个问题

The algorithm treats student knowledge as a set of latent variables,

这个算法把学生的知识 当成一组隐藏变量

which are variables whose true value is hidden from

这些变量的值，对外部是不可见的

an outside observer, like our software.

比如我们的软件

This is also true in the physical world,

这在现实中也是一样的

where a teacher would not know for certain whether

老师无法知道 学生是否完全掌握了某个知识点

a student knows something completely.

老师无法知道 学生是否完全掌握了某个知识点

Instead, they might probe that knowledge using a test

老师会出考题，测试学生能否答对

to see if the student gets the right answer.

老师会出考题，测试学生能否答对

Similarly, Bayesian knowledge tracing updates its estimate of the students’ knowledge

同样，"贝叶斯知识追踪" \N 会看学生答题的正确度，更新学生掌握程度的估算值

by observing the correctness of each interaction using that skill.

同样，"贝叶斯知识追踪" \N 会看学生答题的正确度，更新学生掌握程度的估算值

To do this, the software maintains four probabilities..

它会记录四个概率

First is the probability that a student has learned how to do a particular skill.

首先是 "学生已经学会的概率"

For example, the skill of subtracting constants from both sides of an algebraic equation.

比如从代数方程的两边减去常数

Let’s say our student correctly subtracts both sides by 7.

假设学生正确将两边-7

Because she got the problem correct,

做对了

we might assume she knows how to do this step.

我们可以假设她知道怎么做

But there’s also the possibility that the student got it correct by accident,

但也有可能她是瞎蒙的

and doesn’t actually understand how to solve the problem.

没有真的学会怎么解决问题

This is the probability of guess.

这叫 "瞎猜的概率"

Similarly, if the student gets it wrong,

类似的，如果学生答错了，

you might assume that she doesn’t know how to do the step.

你可能会假设她不会做

But, there’s also the possibility that she knows it,

但她可能知道答案，只是不小心犯了个错

but made a careless error or other slip-up.

但她可能知道答案，只是不小心犯了个错

This is called the probability of slip.

这叫 "失误的概率"

The last probability that Bayesian knowledge tracing calculates

最后一个概率

is the probability that the student started off the problem

是学生一开始不会做，\N 但是在解决问题的过程中，学会了怎么做

not knowing how to do the step, but learned how to do

是学生一开始不会做，\N 但是在解决问题的过程中，学会了怎么做

it as a result of working through the problem.

是学生一开始不会做，\N 但是在解决问题的过程中，学会了怎么做

This is called the probability of transit.

这叫 "做题过程中学会的概率"

These four probabilities are used in a set of equations that update the student model,

有一组方程，会用这四个概率，更新学生模型

keeping a running assessment for each skill the student is supposed to know.

对学生应该学会的每项技能进行持续评估

The first equation asks:

第一个等式问：

what’s the probability that the student has learned a particular skill

学生已经知道某技能的概率是多少？

which takes into account the probability that it was

等式里有

already learned previously and the probability of transit.

"之前已经学会的概率"和"做题过程中学会的概率"

Like a teacher, our estimate of this probability that it was already learned previously

就像老师一样，"之前已经学会的概率"

depends on whether we observe a student getting a question correct or incorrect,

取决于学生回答问题正确与否，

and so we have these two equations to pick from.

回答正确和错误分别有2个公式

After we compute the right value, we plug it into our first equation,

算出结果之后，我们把结果放到第一个方程

updating the probability that a student has learned a particular skill,

更新"之前已经学会的概率"

which then gets stored in their student model.

然后存到学生模型里.

Although there are other approaches,

虽然存在其他方法，

intelligent tutoring systems often use Bayesian knowledge tracing

但"智能辅导系统"通常用 贝叶斯知识追踪

to support what’s called mastery learning, where students practice skills,

让学生练习技能，直到掌握

until they’re deeply understood.

让学生练习技能，直到掌握

To do this most efficiently, the software selects the

为了高效做到这点，软件要选择合适的问题

best problems to present to the student to achieve mastery,

呈现给学生，让学生学

what’s called adaptive sequencing,

这叫：自适应式程序

which is one form of personalization.

个性化算法的形式之一

But, our example is still just dealing with data from one student.

但我们的例子只是一个学生的数据

Internet-connected educational apps or sites

现在有 App 或网站

now allow teachers and researchers the ability

让教师和研究人员 收集上百万学习者的数据

to collect data from millions of learners.

让教师和研究人员 收集上百万学习者的数据

From that data, we can discover things like common pitfalls and where students get frustrated.

从数据中可以发现常见错误\N 一般哪里难倒学生

Beyond student responses to questions,

除了学生的回答，还可以看回答前暂停了多久

this can be done by looking at how long they pause

除了学生的回答，还可以看回答前暂停了多久

before entering an answer, where they speed up a video,

哪个部分加速视频，

and how they interact with other students on discussion forums.

以及学生如何在论坛和其他人互动

This field is called Educational Data Mining,

这个领域叫 "教育数据挖掘"

and it has the ability to use all those face palms and "ah ha" moments

它能用上学生所有的"捂脸"和"啊哈"时刻

to help improve personalized learning in the future.

帮助改善未来的个性化学习

Speaking of the future, educational technologists have often

谈到未来，教育技术人员经常从科幻小说中获得灵感

drawn inspiration for their innovations from science fiction.

谈到未来，教育技术人员经常从科幻小说中获得灵感

In particular, many researchers were inspired by the future envisioned in the book

具体来说，Neal Stephenson 的"钻石时代"这本书\N 激励了很多研究人员

"The Diamond Age" by Neal Stephenson.

具体来说，Neal Stephenson 的"钻石时代"这本书\N 激励了很多研究人员

It describes a young girl who learns from a book

里面说一个年轻女孩从书中学习

that has a set of virtual agents who interact with her

书中有一些虚拟助手会和她互动，教她知识

in natural language acting as coaches, teachers,

书中有一些虚拟助手会和她互动，教她知识

and mentors who grow and change with her as she grows up.

这些助手和她一起成长

They can detect what she knows and how’s she’s feeling,

直到她学会了什么，以及感觉如何，

and give just the right feedback and support to help her learn.

给她正确的反馈和支持，帮助她学习

Today, there are non-science-fiction researchers, such as Justine Cassell,

如今 有非科幻小说研究者，比如 贾斯汀卡塞尔，

crafting pedagogical virtual agents

在制作虚拟教学助手

that can "exhibit the verbal and bodily behaviors found in

助手可以"像人类一样沟通 有人类一样的行为

conversation among humans, and in doing so, build trust,

在陪伴过程中和学习者建立信任，

rapport and even friendship with their human students."

相处融洽，甚至和人类学生成为朋友"

Maybe Crash Course in 2040 will have a little John Green A.I. that lives on your iPhone 30.

2040年的"速成课" \N 可能会有一个 John Green AI，活在你的 iPhone 30 上

Educational technology and devices are now moving off of laptop and desktop computers,

教育科技和设备 \N如今在逐渐扩展到笔记本和台式电脑之外

onto huge tabletop surfaces, where students can collaborate in groups,

比如巨大桌面设备，让学生可以团队合作

and also tiny mobile devices, where students can learn on the go.

以及小型移动设备，让学生路上也能学习

Virtual reality and augmented reality are also getting people excited

"虚拟现实"和"增强现实"也让人们兴奋不已

and enabling new educational experiences for learners –

它们可以为学习者提供全新的体验 -

diving deep under the oceans, exploring outer space,

深潜海洋，探索太空，

traveling through the human body, or interacting with cultures

漫游人体，或是和现实中难以遇见的生物互动

they might never encounter in their real lives.

漫游人体，或是和现实中难以遇见的生物互动

If we look far into the future, educational interfaces might disappear entirely,

如果猜想遥远的未来，教育可能会完全消失，

and instead happen through direct brain learning,

直接在大脑层面进行

where people can be uploaded with new skills, directly into their brains.

把新技能直接下载到大脑

This might seem really far fetched,

这看起来可能很遥远，

but scientists are making inroads already - such as detecting

但科学家们已经在摸索 - 比如

whether someone knows something just from their brain signals.

仅仅通过检测大脑信号，得知某人是否知道什么

That leads to an interesting question:

这带来了一个有趣的问题：

if we can download things INTO our brains,

如果我们可以把东西下载到大脑里

could we also upload the contents of our brains?

我们能不能上传大脑里的东西？

We’ll explore that in our series finale next week about the far future of computing.

下周的最后一集，我们会讨论计算的未来

I'll see you then.

到时见

Hi, I’m Carrie Anne, and welcome to Crash Course Computer Science!

(｡･∀･)ﾉﾞ嗨，我是 Carrie Anne \N 欢迎收看计算机科学速成课！

We’re here: the final episode!

我们到了 最后一集！

If you’ve watched the whole series,

如果你看了整个系列，

hopefully you’ve developed a newfound appreciation

希望你对计算机影响的深度和广度 \N 有全新的认知和欣赏

for the incredible breadth of computing applications and topics.

希望你对计算机影响的深度和广度 \N 有全新的认知和欣赏

It’s hard to believe we’ve worked up from mere transistors and logic gates,

难以相信 我们从简单的晶体管和逻辑门开始

all the way to computer vision, machine learning, robotics and beyond.

一直到计算机视觉，机器学习，机器人以及更多

We’ve stood on the shoulders of giants

我们站在巨人的肩膀上

like Babbage and Lovelace, Hollerith and Turing,

Charles Babbage \N Ada Lovelac \N Herman Hollerith \N Alan Turing

Eckert and Hopper, Sutherland and Engelbart,

J. Presper Eckert \N Grace Hopper \N Ivan Sutherland \N Douglas Engelbart

Bush and Berners Lee, Gates and the Woz,

Vannevar Bush (Memex) \N Berners-Lee (万维网) \N Bill Gates (微软)\N Steve Wozniak (苹果)

and many other computing pioneers.

和许多其他先驱

My biggest hope is that these episodes have inspired you to

我最大的希望是 这些视频能激励你 \N 去了解这些东西如何影响你的人生

learn more about how these subjects affect your life.

我最大的希望是 这些视频能激励你 \N 去了解这些东西如何影响你的人生

Maybe you’ll even pick up programming or choose a career in computing.

甚至开始学编程，或找一份计算机职业

It’s awesome!

这很棒！

It’s also a skill of the future.

这是未来的技能

I said in the very first episode that computer science isn’t magic, but it sort of is!

我在第一集说过，计算机科学不是魔法\N 但它有点像魔法

Knowing how to use and program computers is sorcery of the 21st century.

学习使用电脑和编程，是21世纪的巫术

Instead of incantations and spells, it’s scripts and code.

只不过用的不是咒语 而是代码

Those who know how to wield that tremendous power will be able to craft great things,

懂得运用的人，能创造出伟大的东西

not just to improve their own lives, but also their communities and humanity at large.

不仅改善自己的生活，还有当地社区乃至整体人类

Computing is also going to be literally everywhere –

计算机会随处可见 -

not just the computers we see today, sitting on desks and countertops,

不仅是放在桌上 带在包里

and carried in pockets and bags – but inside every object imaginable.

而是在所有可想象的东西里

Inside all your kitchen appliances, embedded in your walls, nanotagged in your food,

厨房用具里，墙里，食物里

woven into your clothes, and floating around inside your body.

编织进衣服里，在你的血液里

This is the vision of the field of Ubiquitous Computing.

这是"普适计算"的愿景

In some ways, it’s already here, and in other ways, we’ve got many decades to go.

从某种角度来讲 它已经来临了\N 而换一个角度 还要几十年

Some might view this eventuality as dystopian,

有些人把这种未来看成 反乌托邦

with computers everywhere surveilling us and competing for our attention.

到处都有监视器，有无数东西想吸引我们的注意力

But the late Mark Weiser, who articulated this idea in the 1990s,

但 1990 年代提出这个想法的 马克·维泽尔

saw the potential very differently:

看到了非常不同的潜力：

"For [fifty] years, most interface design, and most computer design,

"[五十]年来，大多数界面和计算机设计，

has been headed down the path of the "dramatic" machine.

都是朝"戏剧性"方向前进

Its highest idea is to make a computer so exciting, so wonderful,

想把计算机做得超好，让人一刻也不想离开

so interesting, that we never want to be without it.

想把计算机做得超好，让人一刻也不想离开

A less-traveled path I call the "invisible";

另一条少有人走的路 是"无形"的

its highest idea is to make a computer so imbedded, so fitting,

把计算机整合到所有东西里 \N 用的时候很自然 完全注意不到

so natural, that we use it without even thinking about it …

把计算机整合到所有东西里 \N 用的时候很自然 完全注意不到

The most profound technologies are those that disappear.

最厉害的科技是看不见的科技

They weave themselves into the fabric of everyday life

它们融入到日常生活的每一部分 直到无法区分"

until they are indistinguishable from it."

它们融入到日常生活的每一部分 直到无法区分"

That doesn’t describe computing of today

如今我们还没达到这样

– where people sit for hours upon end in front of computer monitors,

- 人们在电脑前连续坐好几小时

and social media notifications interrupt us at dinner.

吃晚餐被手机推送通知打扰

But, it could describe computing of the future, our final topic.

但它可以描述计算的未来 \N 本系列最后一个主题

When people think of computing in the future,

人们思考计算机的未来时 经常会直接想到人工智能

they often jump right to Artificial Intelligence.

人们思考计算机的未来时 经常会直接想到人工智能

No doubt there will be tremendous strides made in AI in the coming years,

毫无疑问，接下来几十年人工智能会有巨大进步

but not everything will be, or need to be, AI-powered.

但不是所有东西都要做成 AI ，或需要 AI

Your car might have an AI to self-drive, but the door locks

车有自动驾驶AI，但门锁依然会很简单

might continue to be powered by what are essentially if-statements.

车有自动驾驶AI，但门锁依然会很简单

AI technology is just as likely to enhance existing devices,

人工智能可能只是增强现有设备

like cars, as it is to open up entirely new product categories.

比如汽车，AI 带来了一个全新的产品种类

The exact same thing happened with the advent of electrical power – lightbulbs replaced candles.

刚出现电力时也是这样，灯泡取代了蜡烛.

But electrification also led to the creation of hundreds of new electrically-powered gadgets.

但电气化也导致上百种新的电动小工具诞生

And of course, we still have candles today.

当然 我们如今仍然有蜡烛

It’s most likely that AI will be yet another tool

最可能的情况是 AI 变成 \N 计算机科学家手中的另一门新工具

that computer scientists can draw upon to tackle problems.

最可能的情况是 AI 变成 \N 计算机科学家手中的另一门新工具

What really gets people thinking, and sometimes sweating,

但真正让人深思和担忧的是

is whether Artificial Intelligence will surpass human intelligence.

人工智能是否会超越人类智能？

This is a really tricky question for a multitude of reasons,

这个问题很难 有多方面原因

including most immediately: "what is intelligence?"

比如 "智能的准确定义是什么？"

On one hand, we have computers that can drive cars,

一方面，有会开车的计算机

recognize songs with only a few seconds of audio,

几秒就能识别歌的 App

translate dozens of languages, and totally dominate at games like chess, Jeopardy, and Go.

翻译几十种语言，\N 还称霸了一些游戏，比如象棋，知识竞答和围棋

That sounds pretty smart!

听起来很聪明！

But on the other hand, computers fail at some basic tasks,

但另一方面，计算机连一些简单事情都做不了

like walking up steps, folding laundry,

比如走楼梯，叠衣服，

understanding speech at a cocktail party, and feeding themselves.

在鸡尾酒派对和人聊天，喂饱自己

We’re a long way from Artificial Intelligence that’s as general purpose and capable as a human.

人工智能成长到和人类一样通用，还有很长的路

With intelligence being somewhat hard to quantify,

因为"智能"是难以量化的指标

people prefer to characterize computers and creatures

人们更喜欢用处理能力来区分

by their processing power instead,

人们更喜欢用处理能力来区分

but that’s a pretty computing-centric view of intelligence.

但这种衡量智能的方法比较"以计算为中心"

Nonetheless, if we do this exercise,

但如果把视频中出现过的电脑和处理器 画张图

plotting computers and processors we’ve talked about in this series,

但如果把视频中出现过的电脑和处理器 画张图

we find that computing today has very roughly equivalence in calculating

可以看到 如今的计算能力粗略等同于一只老鼠

power to that of a mouse...

可以看到 如今的计算能力粗略等同于一只老鼠

which, to be fair, also can’t fold laundry, although that would be super cute!

公平点说，老鼠也不会叠衣服\N 但如果真的会叠 就太可爱了

Human calculating power is up here, another 10 to the 5,

人类的计算能力在这儿，多10的5次方

or 100,000 times more powerful than computers today.

也就是比如今电脑强10万倍

That sounds like a big gap, but with the rate of change in computing technologies,

听起来差距很大，但按如今的发展速度，

we might meet that point in as early as a decade,

也许十几年就可以赶上了

even though processor speeds are no longer following Moore’s Law,

虽然现在处理器的速度 不再按摩尔定律增长了

like we discussed in Episode 17.

我们在第17集讨论过

If this trend continues, computers would have more processing power/intelligence,

假设趋势继续保持下去，在本世纪结束前

than the sum total of all human brains combined before the end of this century.

计算机的处理能力/智能 会比全人类加起来还多

And this could snowball as such systems need less human input,

然后人的参与会越来越少，人工超级智能会开始改造自己

with an artificial superintelligence designing and training new versions of itself.

然后人的参与会越来越少，人工超级智能会开始改造自己

This runaway technological growth, especially with respect to an intelligence explosion,

智能科技的失控性发展叫 "奇点"

is called the singularity.

智能科技的失控性发展叫 "奇点"

The term was first used by our old friend from Episode 10,

第10集 约翰·冯·诺伊曼 最早用这个词

John von Neumann, who said:

他说：

"The accelerating progress of technology and changes in the mode of human life,

"越来越快的技术发展速度和人类生活方式的改变，

give the appearance of approaching some essential singularity

看起来会接近人类历史中某些重要的奇点

in the history of the race beyond which human affairs,

看起来会接近人类历史中某些重要的奇点

as we know them, could not continue."

这个势头不会永远继续下去"

And Von Neumann suggested this back in the 1950s,

冯诺依曼在 1950 年代说的这话.

when computers were trillions of times slower than they are today.

那时计算机比现在慢得多

Sixty years later, though, the singularity is

六十年后的今天，奇点仍然在遥远的地平线上

still just a possibility on the horizon.

六十年后的今天，奇点仍然在遥远的地平线上

Some experts believe this progress is going to level off,

一些专家认为 发展趋势会更平缓一些

and be more of an S curve than an exponential one,

更像是S型，而不是指数型

where as complexity increases, it becomes more difficult to make additional progress.

而随着复杂度增加，进步会越来越难

Microsoft co-founder Paul Allen calls it a "complexity brake".

微软联合创始人 保罗·艾伦 叫这个"复杂度刹车"

But, as a thought experiment,

但当作思维练习

let’s just say that superintelligent computers will emerge.

我们假设 超智能计算机会出现。

What that would mean for humanity is a hotly debated topic.

这对人类意味着什么，是个讨论激烈的话题

There are people who eagerly await it,

有些人迫不及待

and those who are already working to stop it from happening.

有些人则努力阻止它

Probably the most immediate effect would be technological unemployment,

最直接的影响可能是"技术性失业"

where workers in many job sectors are rendered obsolete

很多工作被计算机，比如AI和机器人，给代替掉了

by computers – like AIs and Robots –

很多工作被计算机，比如AI和机器人，给代替掉了

that can do their work better and for less pay.

它们的效率更高，成本更低

Although computers are new, this effect is not.

虽然计算机出现没多久，但"技术性失业"不是新事

Remember Jacquard's Loom from Episode 10?

还记得第10集里 雅卡尔的织布机 吗？

That automated the task of skilled textile workers back in the 1800s, which led to riots.

它让1800年代的纺织工人失业，导致了骚乱

Also, back then, most of the population of the US and Europe were farmers.

当时美国和欧洲 大部分人都是农民

That’s dropped to under 5% today,

如今农民占人口比例<5％

due to advances like synthetic fertilizers and tractors.

因为有合成肥料和拖拉机等等技术

More modern examples include telephone switchboard operators

时间更近一些的例子是"电话接线员"

being replaced with automatic switchboards in 1960,

在1960年被自动接线板代替了

and robotic arms replacing human painters in car factories in the 1980s.

还有1980年代的"机器喷漆臂"替代了人工喷漆

And the list goes on and on.

这样的例子还有很多.

On one hand, these were jobs lost to automation.

一方面，因为自动化失去了工作

And on the other hand, clothes, food, bicycles, toys,

另一方面，我们有大量产品，\N 衣服，食物，自行车，玩具等

and a myriad of other products are all plentiful today

另一方面，我们有大量产品，\N 衣服，食物，自行车，玩具等

because they can be cheaply produced thanks to computing.

因为可以廉价生产

But, experts argue that AI, robots and computing technologies in general,

但专家认为人工智能，机器人 以及更广义的计算

are going to be even more disruptive than these historical examples.

比之前更有破坏性

Jobs, at a very high level, can be summarized along two dimensions.

工作可以用两个维度概括

First, jobs can be either more manual – like assembling toys

首先，手工型工作，比如组装玩具

– or more cognitive – like picking stocks.

- 或思维型工作 - 比如选股票

These jobs can also be routine – the same tasks over and over again

还有重复性工作，一遍遍做相同的事

or non-routine, where tasks vary and workers need to problem solve and be creative.

或非重复性，需要创造性的解决问题

We already know that routine-manual jobs can be automated by machines.

我们知道 重复性手工工作，可以让机器自动化

It has already happened for some jobs and is happening right now for others.

现在有些已经替代了，剩下的在逐渐替代

What’s getting people worried is that non-routine manual jobs,

让人担心的是"非重复性手工型工作"

like cooks, waiters and security guards, may get automated too.

比如厨师，服务员，保安。

And the same goes for routine cognitive work,

思维型工作也一样

like customer service agents, cashiers, bank tellers, and office assistants.

比如客服，收银员，银行柜员和办公室助理

That leaves us with just one quadrant that might be safe,

剩下一个暂时比较安全的象限

at least for a little while:

剩下一个暂时比较安全的象限

non-routine cognitive work,

非重复性思维型工作

which includes professions like teachers and artists,

包括教师和艺术家，

novelists and lawyers, and doctors and scientists.

小说家和律师，医生和科学家

These types of jobs encompass roughly 40% of the US workforce.

这类工作占美国劳动力大概40％

That leaves 60% of jobs vulnerable to automation.

意味着剩下60％工作容易受自动化影响

People argue that technological unemployment at this scale

有人认为这种规模的技术失业

would be unprecedented and catastrophic,

是前所未有的，会导致灾难性的后果，

with most people losing their jobs.

大部分人会失业

Others argue that this will be great,

其他人则认为很好，

freeing people from less interesting jobs to pursue better ones,

让人们从无聊工作解脱，去做更好的工作，

all while enjoying a higher standard of living with the bounty of food and products

同时享受更高生活水平，有更多食物和物品

that will result from computers and robots doing most of the hard work.

都是计算机和机器人生产的.

No one really knows how this is going to shake out,

没人知道未来到底会怎样

but if history is any guide, it’ll probably be ok in the long run.

但如果历史有指导意义，长远看 一切会归于平静

Afterall, no one is advocating that 90% of people

毕竟，现在没人嚷嚷着让90％的人 回归耕田和纺织

go back to farming and weaving textiles by hand.

毕竟，现在没人嚷嚷着让90％的人 回归耕田和纺织

The tough question, which politicians are now discussing,

政界在讨论的棘手问题是

is how to handle hopefully-short-term economic disruption,

怎么处理数百万人突然失业 \N 造成的短期经济混乱

for millions of people that might be suddenly out of a job.

怎么处理数百万人突然失业 \N 造成的短期经济混乱

Beyond the workplace, computers are also very likely to change our bodies.

除了工作，计算机很可能会改变我们的身体

For example, futurist Ray Kurzweil believes that

举个例子, 未来学家 Ray Kurzweil 认为

"The Singularity will allow us to transcend

"奇点会让我们超越 肉体和大脑的局限性

[the] limitations of our biological bodies and brains.

"奇点会让我们超越 肉体和大脑的局限性

We will gain power over our fates.

我们能掌控自己的命运

... We will be able to live as long as we want.

可以想活多久活多久 我们能完全理解并扩展大脑思维

We will fully understand human thinking and will vastly extend and expand its reach."

可以想活多久活多久 我们能完全理解并扩展大脑思维

Transhumanists see this happening in the form of cyborgs,

超人类主义者认为会出现"改造人"

where humans and technology merge, enhancing our intellect and physiology.

人类和科技融合在一起，增强智力和身体

There are already brain computer interfaces in use today.

如今已经有脑电接口了

And wearable computers, like Google Glass and Microsoft Hololens,

而 Google Glass 和 微软 Hololens \N 这样的穿戴式计算机 也在模糊这条界线

are starting to blur the line too.

而 Google Glass 和 微软 Hololens \N 这样的穿戴式计算机 也在模糊这条界线

There are also people who foresee "Digital Ascension",

也有人预见到"数字永生"

which, in the words of Jaron Lanier,

Jaron Lanier 的说法是

"would involve people dying in the flesh and being uploaded into a computer and remaining conscious".

"人类的肉体死去，意识上传到计算机"

This transition from biological to digital beings

从生物体变成数字体 可能是下一次进化跨越

might end up being our next evolutionary step...

从生物体变成数字体 可能是下一次进化跨越

and a new level of abstraction.

一层新的抽象

Others predict humans staying largely human,

其他人则预测 人类大体会保持原样

but with superintelligent computers as a benevolent force,

但超智能电脑会照顾我们，帮我们管农场

emerging as a caretaker for humanity – running all the farms,

但超智能电脑会照顾我们，帮我们管农场

curing diseases, directing robots to pick-up trash,

治病，指挥机器人收垃圾，

building new homes and many other functions.

建房子 以及很多其他事情

This would allow us to simply enjoy our time on this lovely pale blue dot.

让我们在这个可爱蓝点上(地球) 好好享受

Still others view AI with more suspicion –

另一些人对 AI 持怀疑态度 -

why would a superintelligent AI waste its time taking care of us?

为什么超级人工智能 会费时间照顾我们？

It’s not like we’ve taken on the role of being the benevolent caretaker of ants.

人类不也没照顾蚂蚁吗？

So maybe this play out like so many Sci-Fi movies

也许会像许多科幻电影一样，和计算机开战

where we’re at war with computers, our own creation having turned on us.

也许会像许多科幻电影一样，和计算机开战

It’s impossible to know what the future holds,

我们无法知道未来到底会怎样

but it’s great that this discussion and debate is already happening,

但现在已经有相关讨论了，这非常好

so as these technologies emerge, we can plan and react intelligently.

所以等这些技术出现后，我们可以更好地计划

What’s much more likely, regardless of whether you see computers as future friend or foe,

不论你把计算机视为未来的朋友或敌人

is that they will outlive humanity.

更有可能的是，它们的存在时间会超过人类

Many futurists and science fiction writers have speculated

许多未来学家和科幻作家猜测

that computers will head out into space and colonize the galaxy,

机器人会去太空殖民

ambivalent to time scales, radiation,

无视时间，辐射 \N 以及一些其他让人类难以长时间太空旅行的因素.

and all that other stuff that makes

无视时间，辐射 \N 以及一些其他让人类难以长时间太空旅行的因素.

long-distance space travel difficult for us humans.

无视时间，辐射 \N 以及一些其他让人类难以长时间太空旅行的因素.

And when the sun is burned up and the Earth is space dust,

亿万年后太阳燃尽 地球成为星尘 \N 也许我们的机器人孩子 会继续努力探索宇宙每一个角落

maybe our technological children will be hard at work

亿万年后太阳燃尽 地球成为星尘 \N 也许我们的机器人孩子 会继续努力探索宇宙每一个角落

exploring every nook and cranny of the universe,

亿万年后太阳燃尽 地球成为星尘 \N 也许我们的机器人孩子 会继续努力探索宇宙每一个角落

hopefully in honor of their parents’ tradition to build knowledge,

以纪念它们的父母，同时让宇宙变得更好，

improve the state of the universe,

以纪念它们的父母，同时让宇宙变得更好，

and to boldly go where no one has gone before!

大胆探索无人深空

In the meantime, computers have a long way to go,

与此同时，计算机还有很长的路要走

and computer scientists are hard at work advancing

计算机科学家们在努力推进 过去40集谈到的话题

all of the topics we talked about over the past forty episodes.

计算机科学家们在努力推进 过去40集谈到的话题

In the next decade or so,

在接下来的十几年

we’ll likely see technologies like virtual and augmented reality,

VR 和 AR，无人驾驶车，无人机，可穿戴计算机，

self-driving vehicles, drones, wearable computers,

VR 和 AR，无人驾驶车，无人机，可穿戴计算机，

and service robots go mainstream.

和服务型机器人 会变得主流

The internet will continue to evolve new services,

互联网会继续诞生新服务

stream new media, and connect people in different ways.

在线看新媒体. 用新方式连接人们

New programming languages and paradigms will be developed

会出现新的编程语言和范例，帮助创造令人惊叹的新软件

to facilitate the creation of new and amazing software.

会出现新的编程语言和范例，帮助创造令人惊叹的新软件

And new hardware will make complex operations blazingly fast,

而新硬件能让复杂运算快如闪电 \N 比如神经网络和3D图形

like neural networks and 3D graphics.

而新硬件能让复杂运算快如闪电 \N 比如神经网络和3D图形

Personal computers are also ripe for innovation,

个人电脑也会创新

perhaps shedding their forty-year old desktop metaphor

不像过去40年着重宣传 "桌面" 电脑

and being reborn as omnipresent and lifelong virtual assistants.

而是变成无处不在的虚拟助手

And there’s so much we didn’t get to talk about in this series,

这个系列 我们还有很多话题没谈

like cryptocurrencies, wireless communication,

比如加密货币，无线通讯，3D打印，生物信息学和量子计算

3D printing, bioinformatics, and quantum computing.

比如加密货币，无线通讯，3D打印，生物信息学和量子计算

We’re in a golden age of computing

我们正处于计算机的黄金时代

and there’s so much going on, it’s impossible to summarize.

有很多事情在发生，全部总结是不可能的

But most importantly, you can be a part of this amazing transformation and challenge,

但最重要的是 你可以学习计算机 \N 成为这个惊人转型的一部分

by learning about computing, and taking what’s arguably humanity’s greatest invention,

但最重要的是 你可以学习计算机 \N 成为这个惊人转型的一部分

to make the world a better place.

把世界变得更好

Thanks for watching.

感谢收看