苏菲拉底 THE SOPHIE LAB

本期追问

极简规则反复自我应用，为何能长出无限复杂的图形?

代码里没有自调用函数时，递归藏在哪一层?

有限的面积为什么能被无限长的边界围住?

视频精读 VIDEO STUDY

MIT Godel Escher Bach Lecture 2

来源：YouTube 视频（https://www.youtube.com/watch?v=HqmUuHnvJ98）

节目发布：2012-12-02 · 全文 110 段 · 页边注释 64 条 · 生词词组 89 条

好视频，值得精读

目录 CONTENTS

导读　视频总结

本期讲者

Curran Kelleher　MIT 开放课程《哥德尔、埃舍尔、巴赫》暑期研讨课讲者，本讲代替 Justin Curry 主讲；讲义及树、科赫曲线、蕨类、曼德博集合等示例程序均由其编写，后长期从事数据可视化开发与教学。

学生（提问者）　课堂现场的听众（该课程面向高中生），围绕无限周长、随机性与递归判定等问题多次提问，推动了讲解的深入。

第一部分 全文对照 · 附页边注释与生词标注

01 开场：递归与分形是什么

0:00 Curran Kelleher

The following content is provided under a Creative Commons license. Your support will help MIT Open Courseware continue to offer high-quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT Open Courseware at ocw.mit.edu. So, my name's Curran Kelleher. I'll be lecturing today about recursion and fractals. Justin Curry's not here today. So, I'm going to fill in. So, today I'm going to just do a bunch of example programs, computer programs that are recursive. Some of them don't make pictures and some of them do. And when they make pictures, they're fractals.

以下内容基于知识共享许可协议提供。您的支持将帮助 MIT 开放课程继续免费提供高质量的教育资源。如需捐赠或查看数百门 MIT 课程的更多资料，请访问 MIT 开放课程网站 ocw.mit.edu。我叫 Curran Kelleher。今天由我来讲递归和分形。Justin Curry 今天不在，所以我来代课。那么，今天我打算演示一堆示例程序，都是递归的计算机程序。有些不会画图，有些会。而当它们画图的时候，画出来的就是分形。

本讲属 MIT OpenCourseWare 上的《哥德尔、埃舍尔、巴赫》暑期研讨课（面向高中生），第 2 讲主题为递归与分形，由 Curran Kelleher 代替原讲者 Justin Curry 授课。

fill in *phr. v.*
代课；临时顶替（fill in for sb.）

recursive /rɪˈkɜːrsɪv/ *adj.*
递归的；本讲核心术语，指自我调用或自我指涉的

0:40

Frac– fractals are things that are self–similar at different scales that you can zoom in on. we're going to take a break at 4:00 for 10 minutes. And then towards the end, hopefully, I'll show you a bunch of examples of fractals and play some Bach music. So, first of all, let's consider a recursive mathematical function, factorial. something factorial, like three factorial is 3 * 2 * 1. Four factorial is 4 * 3 * 2 * 1. So, like and this is factorial, the exclamation point.

分形是那种在不同尺度上都自相似的东西，你可以不断放大去看。我们会在 4 点休息 10 分钟。然后接近尾声的时候，希望我能给你们看一堆分形的例子，还会放一些巴赫的音乐。首先，我们来看一个递归的数学函数，阶乘。比如某个数的阶乘，3 的阶乘就是 3 × 2 × 1。4 的阶乘就是 4 × 3 × 2 × 1。这就是阶乘，写成一个感叹号。

分形（fractal）的定义：在不同尺度上自相似、可无限放大的结构。该术语由 Mandelbrot 于 1975 年创造。分形、递归与巴赫音乐的关联正是 GEB 一书的主线之一。

self–similar /ˌsɛlfˈsɪmələr/ *adj.*
自相似的（局部与整体形状相同），分形的定义性特征

zoom in on *phr. v.*
放大观察；聚焦于

factorial /fækˈtɔːriəl/ *n.*
阶乘（n! = n×(n–1)×…×1）

exclamation point *phr.*
感叹号（美式说法；英式为 exclamation mark）

02 阶乘与斐波那契：递归入门

1:24

So, the way this is defined is actually recursive. So, if you take anything factorial, say n, factorial is n minus 1 factorial. say Let's say n is four, and then n minus 1 would be three. So, four factorial is Wait. n Yeah, times n. So, it's going to be Well, n times n minus 1 factorial. So, four is n times this whole thing is three factorial. It's n minus 1 factorial. So, this is a recursive definition. And if you look in the handout, I wrote a little computer program on page three, I think, that does the factorial. So, it goes like this.

它的定义其实是递归的。所以如果你取任意一个数的阶乘，比如 n，n 的阶乘就是 n 减 1 的阶乘……比如说 n 是 4，那 n 减 1 就是 3。所以 4 的阶乘是……等一下，n……对，乘以 n。所以应该是 n 乘以 n 减 1 的阶乘。所以，4 就是 n，而这一整块就是 3 的阶乘。也就是 n 减 1 的阶乘。所以这是一个递归定义。如果你看讲义的话，我在第三页写了一个小程序，我记得是第三页，就是算阶乘的。它是这样的。

阶乘 n! = n × (n–1)! 是递归定义的最简范例：把问题化归为规模更小的同类问题，这一思路贯穿全讲所有例子。

handout /ˈhændaʊt/ *n.*
（课堂发的）讲义、资料

2:40

So, just some for you for those of you who don't know much about programming, def means define. So, we're defining a function called factorial. And it takes as an argument n. So, you can call this function, pass it a number. And inside the function, it's referred to as n. So, fact this factorial function says if n is greater than 1, return n times factorial of n minus 1.

为了照顾那些不太懂编程的同学，def 的意思是 define，定义。所以我们在定义一个叫 factorial 的函数。它接受一个参数 n。你可以调用这个函数，传给它一个数字。在函数内部，这个数字就叫 n。所以这个 factorial 函数说：如果 n 大于 1，就返回 n 乘以 factorial(n – 1)。

def 是 Python、Groovy 等语言中定义函数的关键字。argument 在编程语境中指传入函数的参数，而非「争论」。

argument /ˈɑːrgjəmənt/ *n.* （编程）参数、实参；注意不是「争论」义

3:19

Else, return 1.

否则，返回 1。

3:29

So, what makes this function recursive is the fact that it calls itself. Factorial is defined as n times factorial of something else. So, a recursive function is a function that calls itself. So, let me give an example. So, if n is five, say, the way we call this is we say factorial of five. So, when we when we say this, it calls this function and gives n the number five. So, it's what it's going to do is n is going to be five. So, n is greater than 1, so we return n times factorial of n minus 1.

让这个函数成为递归的，正是它调用自身这一点。factorial 被定义为 n 乘以另一个factorial。所以，递归函数就是调用自身的函数。嗯，我举个例子。假设 n 是 5，我们调用的方式就是写 factorialof five。所以当我们这样写的时候，它就调用这个函数，并把 5 传给 n。那它会做什么呢？ n 等于 5。n 大于 1，所以我们返回 n 乘以 factorial(n减 1)。

递归函数的判定标准在此点明：函数在自身函数体内调用自身。每次调用时参数减小（n 变 n–1），逐步逼近终止条件。

4:18

So, say we're doing this algorithm, five is n. So, we're going to return five times factorial of n minus 1. So, we call factorial with the value four. And that's also greater than 1. So, we return four times factorial three. So, five times four times factorial three. So, we loop we call itself a bunch of times until we get down to 1.

假设我们执行这个算法，n 是 5。那我们就返回 5 乘以 factorial(n 减 1)。也就是用 4 这个值去调用 factorial。而 4 也大于 1。所以我们返回 4 乘以 factorial(3)。于是就是 5 乘以 4 乘以 factorial(3)。所以我们就这样循环，它不断地调用自己，直到降到 1 为止。

4:57

So, that's what actually ends up happening. This is recursion. So, another simple example, which is sort of like the factorial, is the Fibonacci numbers, Fibonacci sequence.

这就是实际发生的过程。这就是递归。另一个类似阶乘的简单例子，就是斐波那契数，斐波那契数列。呃……

5:19

So, the Fibonacci numbers, it goes 1 1 and then the next one, you add the first two together. So, it goes 1 1 2. 1 + 2 is 3. 2 + 3 is 5. And so on. 8 Blah. These are the Fibonacci numbers. So, this is a recursive definition. let's say this is

斐波那契数是这样的：1、1，然后下一个就是把前两个加起来。所以是 1、1、2。1 加 2 等于 3。2 加 3 等于 5。以此类推，8，等等等等。这些就是斐波那契数。这是一个递归定义。呃，我们说这个是……

> 斐波那契数列：1,1,2,3,5,8…每项等于前两项之和。自然界中花瓣数、松果与向日葵的螺线排列常呈现该数列，与分形一样是「自然中的数学」。

5:53

These This is like the number of the element. Like, they're just numbers, index indices, if you will. So, Fibonacci of two is Fibonacci of zero plus Fibonacci of one. And so, Fibonacci five is going to be Fibonacci of three plus Fibonacci of four. So, generally, Fibonacci of n is going to be Fibonacci of n minus 1 plus Fibonacci of n minus 2. So, we could say that here. Instead of factorial, we call it Fibonacci. And we'll notice that like they're almost the same thing. I'll say fib. If n minus if n is greater than 1, we return

这些……这个相当于元素的编号。就是一些数字，索引，如果你愿意这么叫的话。所以 Fibonacci(2) 等于 Fibonacci(0) 加 Fibonacci(1)。那么 Fibonacci(5) 就等于 Fibonacci(3) 加 Fibonacci(4)。一般来说，Fibonacci(n) 就等于 Fibonacci(n 减 1) 加 Fibonacci(n 减 2)。我们可以在这里这样写。把 factorial 换成 Fibonacci。你会发现它们几乎是一样的东西。我就写成 fib。如果 n 减……如果 n 大于 1，我们就返回

6:44

Fibonacci of n minus 1.

Fibonacci(n 减 1)。

注意讲者此处口误：代码应返回 fib(n−1) + fib(n−2) 两项之和。这种朴素递归实现的时间复杂度是指数级的，实际工程中会用迭代或缓存优化。

indices /ˈɪndɪsiːz/ *n.*
index 的复数：索引、下标

if you will *phr.*
姑且这么说吧（缓和措辞的口头插入语）

03 递归转移网络生成英语句子

6:54

And this gives us the Fibonacci numbers, actually. so, it makes sense to you guys? So, on the in the handout, there are some example outputs of both of these, and you can see that's what happens. So, who has their copy of Gödel, Escher, Bach today? So, if we will look on page 132 of Gödel, Escher, Bach. Oh, no, no, I can't look at it. Oh, well. 132 of Gödel, Escher, Bach has these two diagrams that are recursive tran– transition networks. They define a grammar, like sort of like English. It's not in English, it's not complete. It's a simplified version of English, but it he communicates the essence of the notion of a grammar, a recursive grammar.

这实际上就给出了斐波那契数。嗯，大家听明白了吗？在讲义上有这两个程序的一些示例输出，你们可以看到就是这样的结果。那么，今天谁带了《哥德尔、埃舍尔、巴赫》这本书？我们来看《哥德尔、埃舍尔、巴赫》第 132 页。哦，不不，我看不了。算了。《哥德尔、埃舍尔、巴赫》第 132 页有两幅图，是递归转移网络。它们定义了一种语法，有点像英语。不是英语，也不完整，是英语的简化版本，但他传达出了语法——递归语法——这个概念的精髓。

递归转移网络（RTN）出自 GEB 第 5 章第 131–132 页：用有向图定义一个简化英语语法，节点可再调用其他网络甚至自身，从而生成嵌套句。

transition networks *phr.*
转移网络（RTN，用有向图表示语法的形式化工具）

essence /ˈɛsəns/ *n.*
本质、精髓（communicate the essence of 传达…的精髓）

notion /ˈnoʊʃən/ *n.*
概念、观念（比 idea 更正式）

7:50

So, you'll notice that It's hard to do it in my head. Fancy noun, one of the nodes calls fancy noun again. It loops back out on itself. So, this is where the recursion is. So, what I did is I took this diagram and wrote a little computer program that whenever there's a choice of the transitions, it chooses one of those transitions at random. And this is the program I wrote. I think it's on page five of my handout. So, if you look at that, Yeah, I wish I had the projector. It's unfortunate.

你会注意到，嗯……在脑子里想这个有点难。fancy noun（花式名词），其中一个节点又调用了 fancy noun 自己。它绕回到自身。这就是递归所在。于是我做的事情就是，我把这幅图拿过来，写了一个小程序。每当有多个转移可以选择时，它就随机选一条转移路径。嗯，这就是我写的程序。我想它在讲义第五页。如果你看那个，嗯……唉，真希望我有投影仪，太可惜了。

「fancy noun」节点调用自身，是语法递归性的关键：名词短语内部可以再嵌套名词短语，理论上可以无限嵌套下去。讲者把这张图翻译成了随机生成句子的程序。

at random *phr.*
随机地、任意地

8:32

Actually, can I look at the diagram? So, if you look at fancy noun, we begin and it calls ordinate noun. And then if you look at the program in the handout, you find fancy noun. It's sort of halfway down, it says the RTN for fancy noun. Fancy noun equals And this is a function call. Well, first of all, fancy noun equals When you put curly braces around something, it makes it a function, pretty much. So, fancy noun equals and it copies pretty much directly from the diagram. Ordinate noun, which is also a function call, which is defined above, plus I'm still I'm back in fancy noun.

嗯，我能看一下那幅图吗？如果你看 fancy noun，我们从起点开始，它调用 ordinate noun。然后你看讲义里的程序，嗯，你会找到 fancy noun。大概在中间偏下的位置，写着「fancy noun 的 RTN」。fancy noun 等于……这是一个函数调用。首先，fancy noun 等于……当你在某个东西外面加上花括号，基本上就把它变成了一个函数。所以 fancy noun 等于……呃，它基本上是直接从图上抄下来的。ordinate noun，这也是一个函数调用，在上面已经定义过了，加上……我还在讲 fancy noun。

curly braces *phr.*
花括号 { }（编程术语）

9:29

So, ordinate noun plus pick from preposition or relative pronoun or nothing. And if you look at the diagram in Gödel, Escher, Bach, the arrows coming out of ordinate noun point to relative pronoun, nothing, the end, and preposition. So, you can make this into a computer program, which is which is what I did. So, if you look at my program for a little while, you'll notice that all the arrows in the diagrams correspond to function calls in the in the program. And they're recursive because they eventually loop back on themselves.

所以是 ordinate noun 加上从 preposition（介词）、relative pronoun（关系代词）或者「什么都没有」里面选一个。如果你看《哥德尔、埃舍尔、巴赫》里的那幅图，从 ordinate noun 出来的箭头指向 relative pronoun、「什么都没有」、结束，还有 preposition。所以你可以把它写成一个计算机程序，我做的就是这件事。如果你花点时间看我的程序，你会发现图里所有的箭头都对应着程序里的函数调用。在程序里对应函数调用。而它们是递归的，因为它们最终会绕回到自身。

要点：图中的每条箭头对应程序里的一个函数调用。「语法图」与「程序」在结构上同构——不同表示形式共享同一抽象结构，正是 GEB 反复强调的主题。

preposition /ˌprɛpəˈzɪʃən/ *n.*
介词（语法术语）

relative pronoun *phr.*
关系代词（如 which、that，引导定语从句）

loop back on themselves *phr.*
绕回自身、回到自身（描述递归结构的常用表达）

10:10

So, here Paul Stetson is trying to communicate the fact that languages themselves are defined by recursive grammars. This is why we can nest sentences inside of each other. It's recursive. So, recursion leads to nesting and sometimes infinite nesting. And that's what That's where fractals come from. So, right after this program in the handout, there's a sample output. So, just read through some of those sample outputs. They're pretty funny. I have an old version. Can I look at someone's handout just to read? Just a handout.

所以在这里，Paul Stetson 想传达的是：语言本身就是由递归语法定义的。这就是为什么我们可以把句子嵌套在句子里面。这是递归的。所以递归会导致嵌套，有时候是无限嵌套。分形就是从这里来的。讲义里这个程序后面就有一段示例输出。大家读一读那些示例输出。挺挺有意思的。我手上是旧版本。我能看一下谁的讲义读一下吗？就借一下讲义。

「Paul Stetson」应为口误，所指为 GEB 作者 Douglas Hofstadter（侯世达）。自然语言的从句嵌套能力是乔姆斯基生成语法理论的核心论据：有限规则生成无限句子。

nest /nɛst/ *v.*
嵌套（一层套一层地放入）；名词 nesting 嵌套结构

04 递归画树：分枝与深度

10:54

So, small bagel inside the strange cow. it's sort of It makes sense as an English sentence and it was generated by this computer program. So, I think that's just fascinating. But of course, some of them don't make sense like large small bagel that runs large small large horn. Like, it just doesn't make any sense. So, next example. let's see. So, the next example on page five, I think, is a tree. We're going to make a tree picture using recursive functions. So, I'm going to

「奇怪的牛肚子里的小小百吉饼。」呃，它作为一个英语句子还算说得通，而且是这个计算机程序生成的。我觉得这真是太迷人了。当然，有些就说不通了，比如「跑动着的大的小的百吉饼大的小的大的号角」。就是完全讲不通。那么，下一个例子。呃，我看看。第五页上的下一个例子，我想是一棵树。我们要用递归函数画一幅树的图。那么，我要，呃……

随机语法生成器产出「语法通顺但语义荒诞」的句子，直观演示了语法与语义可以分离——机器可以掌握形式规则而不理解意义。

bagel /ˈbeɪgəl/ *n.*
百吉饼（环形硬面包）

11:41

I'm going to write some pseudo code. It's not real code. It's sort of pseudo code to communicate the idea of what this program is doing.

我要写一些伪代码。不是真正的代码，是伪代码，用来传达这个程序在做什么的思路。

pseudo code /ˈsuːdoʊ koʊd/ *n.*
伪代码（只表达思路、不能直接运行的代码）；pseudo– 假的

12:04

So, we have a function that grows a tree. It starts from a single branch. And I'll do it at that one here. This is like the starting point. This is the entry point. So, it says all right, class, all this stuff, tree parenthesis parentheses. That gets called when the program starts. So, this function call, grow tree, 0.50, trunk height, all this stuff is this first one. This is what initiates the process. And then what the function does is pretty much if the depth is greater than zero.

我们有一个函数，用来长出一棵树。它从一根树枝开始。我在这里画一下。这就像是起点，是入口点。所以它说，好，class，所有这些东西，tree tree括号括号。程序启动的时候就会调用它。所以这个函数调用 grow tree、0.50、trunkheight，所有这些东西，就是第一根。这就是启动整个过程的东西。然后这个函数做的事情，嗯，基本上就是如果深度大于零。嗯……

13:16

So, this tree function calls itself twice, once for each branch. So, the first time the program calls this function it makes this. It draws it on the screen. So, notice here it adds the actual line. If you look at the code that it says add new JV line X1 X2 that stuff. That actually adds a line to the screen. I wish I could you know, show you the actual code running, but I can't cuz So, this is the first time. And then depth is the number of times it's going to branch out. And so, depth is 11. It's going to make a big tree.

这个 tree 函数呃，会调用自己两次，每根分枝调用一次。所以程序第一次调用这个函数时，画出的是这个。它把它画在屏幕上。注意，嗯，这里它加上了实际的线条。如果你看那段代码，写着 add new JV line X1 X2 之类的。那实际上就是往屏幕上加一条线。往屏幕上加。真希望我能给你们演示代码实际运行的样子，但我做不到，因为……所以这是第一次。然后 depth 就是它要分枝的次数。depth 是 11，那就会长成一棵很大的树。

画树程序的结构：从树干（trunk）这一入口点开始，函数每层调用自身两次生成左右分枝，depth 参数控制递归深度。

entry point *phr.*
入口点（程序开始执行的地方）

trunk /trʌŋk/ *n.*
树干

14:09

What it's going to do is make two sub branches. Says grow tree This one corre– correlates to this one here. And this one corresponds to this one here. And if you look at the code in the handout, it says grow tree X2 Y2. X2 Y2 is the end point of the previous branch. Root length times size factor. is a factor by which it's going to scale them. So, this would be like maybe half the size or 0.7 of size factor is 0.58, so it's about half the size. And root angle plus angle factor, root angle minus angle factor. These are in the two grow tree function calls.

嗯，它要做的是长出两根子枝。写的是 grow tree……这一个对应这里这一根。这一个对应这里这一根。呃，如果你看讲义里的代码，写的是 grow tree X2 Y2。X2 Y2 是前一根树枝的终点。root length 乘以 size factor。这是缩放的比例因子。所以这根大概是一半的大小，或者 0.7…… size factor 是 0.58，所以差不多是一半大小。还有 root angle 加 angle factor、root angle 减 angle factor。这些在那两个 grow tree 函数调用里。

每层子枝长度乘以缩放因子 0.58——这正是自相似性的来源：每个子结构都是整体按比例缩小的副本。

scale /skeɪl/ *v.*
按比例缩放；n. 尺度

15:02

Angle factor, which is point well, pi over four. It's exactly 45°. So, each time it branches, the two branches are going to branch out at that angle. And so, here say we do depth of four when you call it the first time. The depth here is going to be four. And then you pass into the function depth minus one. So, here inside the function that's generating this, depth is going to be three. And so, it's going to keep going down until depth is zero. So, here depth is well, depth is four.

angle factor 是 0……嗯，是 π 除以 4。正好是 45 度。所以每次分枝的时候，那两根枝就会按那个角度岔开。那么在这里，假设我们第一次调用的时候 depth 是 4。这里的 depth 就是 4。然后你传进函数的是 depth 减 1。所以在生成这个的函数内部，depth 就变成 3。它会一直往下走，直到 depth 等于零。所以这里 depth 是……嗯，depth 是 4。

分枝角固定为 π/4（45°），depth 每层减 1、到 0 停止。depth 归零即递归的「基准情形」（base case），与后文 bottom out 的讨论呼应。

branch out *phr. v.*
分叉、岔开；引申义为拓展新领域

15:45

Three. Two. One. And it does it on this side, too.

3。2。1。这一边也是一样。

05 科赫曲线与无限周长悖论

16:04

So, it just keeps going like this. This is This is a fractal. And you could imagine if you were to continue this infinitely like instead of saying depth of 10 or 11, just say depth of infinity. Say theoretically, if we could do this shape could be zoomed in on infinitely, forever. And this is the notion of a fractal. And it would look the same as it does on the on the large scale. So, any questions about the tree? So, next we're going to do a the Koch curve. Koch curve is sort of similar to the tree except that the rules are different.

所以它就这样一直进行下去。这就是一个分形。你可以想象，如果你无限地继续下去，不说深度 10 或 11，而是说深度无穷大。假设理论上，如果我们能做到，这个形状可以被无限地放大，永远放大下去。这就是分形的概念。放大后它看起来会和大尺度上一模一样。那么，关于树的部分还有什么问题吗？接下来我们要做的是科赫曲线。科赫曲线和树有点类似，只不过规则不一样。

若 depth 为无穷大，树可无限放大且局部与整体相同——这是分形的理论定义。实际程序只能有限逼近，这个「理论 vs 现实」的张力贯穿本讲。

16:50

So, first con– conceptually, this is what a Koch curve is. You start with a line or in some cases a triangle to make the whole thing. And you divide it into three parts. And then you make an equilateral triangle. I mean, the sides are all the same. Out of the thing in the middle. And get rid of this line. So, this is the rule. Go from a line to this thing. And then this rule is applied to each one of these segments. So, you go like this. And so on. Like to each of these smaller segments. So, the fact that like the simple rule is being applied to something over and over again.

首先从概念上讲，科赫曲线是这样的。你从一条线段开始，有时候是从一个三角形开始，来构成整个图形。然后你把它分成三段。接着你做一个等边三角形。我是说，各边长度都相同。以中间那一段为基础做出来。然后把这条线去掉。所以这就是规则：从一条线段变成这个东西。然后这个规则会被应用到这些线段中的每一段上。所以你会得到这样。以此类推，对每一段更小的线段都这样做。所以，正是因为这个简单的规则被一遍又一遍地应用到某个东西上，

科赫曲线由瑞典数学家 Helge von Koch 于 1904 年提出：把线段三等分，以中段为底向外作等边三角形并去掉底边，再对每一小段重复此规则。

equilateral /ˌiːkwəˈlætərəl/ *adj.* 等边的（equilateral triangle 等边三角形）

segments /ˈsɛgmənts/ *n.* 线段；段、部分（segment）

17:35

And the thing it's being applied to is the result of the previous execution of the rule makes it recursive. So, you could keep drawing it. Let's look at the actual code, the program. It says Koch. I think it's on page seven. So, let's see let's see, create curve.

而且被应用的对象是上一次执行规则的结果，这就使它具有递归性。所以你可以一直画下去。我们来看看实际的代码，这个程序。嗯，上面写着 Koch。我想是在第七页。那我们看看，create curve（创建曲线）。

递归的精确刻画：规则作用于上一次规则执行的结果。这与阶乘「n! 用 (n−1)! 来定义」在结构上完全相同，只是对象从数字换成了图形。

execution /ˌɛksəˈkjuːʃən/ *n.* （程序、规则的）执行

18:12

What create curve does essentially is call itself four times.

create curve 本质上做的事，就是调用它自己四次。

create curve 调用自身四次，对应每步生成的四条子线段——递归调用的次数等于自相似部件的个数，这是读分形代码的通用线索。

18:26

And each of those four times corresponds to this one, this one, this one, and this one.

而这四次分别对应这一段、这一段、这一段，还有这一段。

18:39

So, yeah, let's look at the actual code. You see these four function calls to create curve in the middle there? X1 Y1, AX, AY, CX, CY, BY and all this stuff. If you look at the little red diagram this point is X1 Y1. And you know, it's in the diagram what the points are. So, the first function call pretty much says apply the rule to this segment. The second function call starts from A, which is this point here. Which says apply the rule to this segment. The third one starts at C, which is the top point.

嗯，好，我们来看看实际的代码。你们看到中间那四个对 create curve 的函数调用了吗？X1 Y1、AX、AY、CX、CY、BY 等等这些东西。如果你看那个红色的小示意图，这个点就是 X1 Y1。然后你知道，图里标出了这些点分别是什么。所以第一个函数调用基本上就是说：把规则应用到这一段上。第二个函数调用从 A 开始，也就是这里这个点。意思是把规则应用到这一段上。第三个从 C 开始，也就是顶点。

19:25

And says apply the rule to this segment. And the fourth one applies rule to this. So, it's another beautiful recursive fractal. So, what happens So, this is the Koch curve. If you generalize this and add a randomness to it say you know, say these points are A you know, it doesn't matter what they're called. If you move this one up and down a little bit randomly. So, like you start from this, instead of making these points exact, make them like a little bit off. And then connect the lines and make a line a triangle there.

意思是把规则应用到这一段上。第四个则把规则应用到这一段上。所以这又是一个漂亮的递归分形。那么会发生什么呢，嗯，这就是科赫曲线。如果你把它推广一下，加入一些随机性，比如说这些点是 A，你知道，叫什么名字并不重要。如果你把这个点随机地上下移动一点点。就是说，你从这个开始，不要让这些点的位置那么精确，让它们稍微偏一点。然后把线连起来，在那里做出一条线、一个三角形。

generalize /ˈdʒɛnrəlaɪz/ *v.*
推广、一般化（把特例扩展为一般规则）

randomness /ˈrændəmnəs/ *n.*
随机性

20:10

And this is also, you know, a little bit off. And then if you keep doing this, adding a little bit of randomization each time, you actually get lines that look just like coastlines around you know, pieces of land. And if you generalize this into three dimensions and do this randomness, it actually generates 3D mountains, like 3D virtual 3D surfaces that look exactly like real mountains. So, it's sort of strange like these recursive structures are definitely in nature. What does it mean for this one page on page six? The Koch snowflake has finite area but infinite Oh, yeah. Yeah, I forgot to mention that. That's a really cool thing. So, the Koch snowflake is when you start with a triangle and you apply the rule to each sides of the triangle.

这个也稍微偏一点。然后如果你一直这样做下去，每次都加入一点随机化，你实际上会得到看起来就像陆地边缘海岸线的线条。如果你把这个推广到三维并加入这种随机性，它实际上能生成三维的山脉，就像三维的虚拟三维表面，看起来跟真实的山一模一样。所以这有点奇妙，这些递归结构确实存在于自然界中。第六页上这一页说的是什么意思？科赫雪花的面积是有限的，但周长是无限的——哦，对，对，我忘了提这个。这是个非常酷的事情。科赫雪花就是你从一个三角形开始，把规则应用到三角形的每一条边上。

给科赫规则加入随机扰动即可生成逼真海岸线；推广到三维可生成虚拟山地。电影与游戏中的程序化地形生成正是基于这类分形算法。

coastlines /ˈkoʊstlaɪnz/ *n.*
海岸线（coastline）

randomization /ˌrændəmɪˈzeɪʃən/ *n.*
随机化（人为引入随机扰动）

21:04

And you get you get the you get this curve on all these sides. So, it's been mathematically proven or you know, extrapolated that if you if you were to do this rule an infinite number of times, which you can do in math cuz it's all theoretical, the volume inside of this object would be finite. It's a definite amount. But the surface area is infinite. Oh, not surface area. The perimeter. Perimeter is infinite. That's because every time you apply the rule, you actually increase the perimeter.

然后你在所有这些边上都得到这种曲线。所以数学上已经证明了，或者说推导出来了，如果你把这个规则执行无限多次——在数学里你可以这么做，因为都是理论上的——呃，这个图形内部的体积会是有限的。它是一个确定的量。但表面积是无限的。哦，不是表面积，是周长。周长是无限的。这是因为每次你应用这个规则，实际上都会增加周长。

科赫雪花的核心悖论：面积有限（能被一个圆包住）而周长无限——每次迭代周长变为原来的 4/3 倍，无限次后发散。

extrapolated /ɪkˈstræpəleɪtɪd/ *v.*
外推、推断（extrapolate：由已知趋势推出未知）

perimeter /pəˈrɪmɪtər/ *n.*
周长；（营地等的）外围

finite /ˈfaɪnaɪt/ *adj.*
有限的（反义词 infinite 无限的）

21:40

So, this has a certain length. And then once you apply the rule to it, if you do this, then this new curve, the total length is longer than this one. So, you can imagine if you do it infinitely, like it's just going to be infinitely long. So, it's sort of mind-boggling. And this goes, you know, it's a If you like go near a certain number of iterations, it's going to get smaller and smaller. Yeah, it gets smaller and smaller, definitely. But if you do it theoretically an infinite number of times, like it'll still exist. If you look at the picture, on page six, right next to the title, the Koch snowflake, that curve there, you see it?

所以这个呃有一定的长度。然后一旦你对它应用规则，如果你这样做，那么这条新曲线的总长度就比原来这条更长。所以你可以想象，如果你无限次地做下去，它就会变得无限长。所以这挺让人费解的。而且这个，你知道，如果你做到一定的迭代次数，它会变得越来越小。是的，它确实会变得越来越小。但如果理论上你做无限多次，它仍然会存在。如果你看那张图，嗯，第六页上，就在标题旁边，科赫雪花，那里那条曲线，你看到了吗？

每次迭代新增的长度确实越来越小，但总和仍然发散——与调和级数发散类似的反直觉现象：无穷多个越来越小的量加起来可以是无穷大。

mind-boggling /ˈmaɪndˌbɑːglɪŋ/ *adj.*
令人难以置信的、烧脑的

iterations /ˌɪtəˈreɪʃənz/ *n.*
迭代（次数）；iteration 一次重复执行

22:34 学生与讲者（对话）

That's sort of what it would look like even if you did it an infinite number of times. So, if I could have like break it maybe like cut it in half and then like spread it, it go on infinitely. Say again? If I could like maybe have like a kind of like a snowflake like that. And I like cut like it in half like I just cut it like So, like you just cut it in half? Yeah, not in half but just cut like one part of it. I don't want like the segments to go from like one part to the other but just cut it like you would like cut like a string in a circle.

就算你重复无限多次，大概也就是这个样子。所以，如果我，我可以把它掰开比如从中间剪断，然后把它拉开，它就能无限延伸下去。再说一遍？如果我有一个类似那样的雪花图形，然后我把它从中间剪开我就这么剪一下——所以，你是说你把它从中间剪断？对，不是正中间，就是剪掉它的某一部分。我不是想让那些线段从一边连到另一边，而是就这么剪一下，就像你剪断一根围成圆圈的绳子那样。

23:10

I don't really understand. You can draw it on the board if you want. On the board?

我不太明白。你想的话可以到黑板上画出来。到黑板上？

23:21

I have like a circle. I just cut like this part over here and then if I even if this is like finite area, you're saying that if I cut it like that and spread it, it can go on infinitely but that doesn't make I see. So, it's like having a finite area. I see what you mean. So, say you have this Koch curve, this shape, and it were a string and you would cut it so the string would now be loose. And if you pulled it, it would go on forever. Yeah. It would. Yeah. That's what it means to have infinite perimeter, which is why it's just so fascinating.

比如我有一个圆。我就在这个位置剪一下，然后，就算这个东西的面积是有限的，你是说如果我这样剪断然后把它拉开，它可以无限延伸下去，但这说不通——哦，我懂了。所以就是说它的面积是有限的。我明白你的意思了。假设你有这条科赫曲线，这个图形，假如它是一根绳子，你把它剪断，绳子就松开了。然后你一拉，它就会无限地延伸下去。对。确实会。对。这就是周长无限的含义，所以才这么让人着迷。

学生的「剪断绳子拉直」思想实验很有价值：把「周长无限」翻译成可操作的图像——这根围成有限面积的「绳子」剪开拉直后将无限长。

06 英国海岸线的测量难题

24:06 Curran Kelleher

Like it's crazy. People tried to measure the coast of Britain. How long is the coast of Britain, right? Have you heard about this problem? If you look at it on a larger on a large scale, like from a satellite image of the whole country, you could just draw a line around it and say like, "Oh, yeah, it's this length. This is the how the length of the coast of Britain." But if you zoom in on it, you'll find that those big lines that you drew are actually like really wrong. Like they don't actually line up. So, if you make it more precise to this new zooming angle, this zooming, it gets longer.

简直太不可思议了。有人试图测量英国的海岸线。英国的海岸线到底有多长？你们听说过这个问题吗？如果你从很大的尺度上看，比如从整个国家的卫星图像上看，你可以沿着它画一条线，然后说：“哦，就是这么长，这就是英国海岸线的长度。”但如果你放大来看，就会发现你画的那些大线条其实非常不准确，根本对不上。所以，如果你按照放大后的这个尺度画得更精确，长度就会变长。

「英国海岸线有多长」出自 Mandelbrot 1967 年发表于 Science 的著名论文，被视为分形几何学的开端：测量结果依赖于所用标尺的尺度。

satellite image *phr.*
卫星图像

24:39 学生与讲者（对话）

And so actually the more that you zoom in on the coast of Britain, the longer the perimeter gets. And those should be getting smaller cuz you're like just adding little pieces. Yeah, just adding little pieces. So, say the actual coast of Britain is like that. And but you look at it like at this huge distance away, you say like, "Oh, yeah, this is approximately that line." But if you look at it more detail and refine it, you say, "Oh, it's not actually that line. It's maybe like these lines."

所以实际上，你把英国海岸线放大得越厉害，周长就越长。可这些增量应该会越来越小才对，因为你只是在加一些小碎片。是啊，只是加一些小碎片。所以，假设英国海岸线实际长这样。但你从很远的距离看，你会说：“哦，差不多就是这条线。”可你看得更细致、更精确一些，你就会说：“哦，其实不是那条线，可能是这几条线。”

refine /rɪˈfaɪn/ *v.*
使更精确、细化、改进

25:10

But the new lines are actually the whole thing is longer. And if you do it even more and more precisely, you just get longer and longer and longer. So, nobody's been able to really figure out how long the coast of Britain is. It's a fractal. Yeah, but shouldn't you do like Is it even possible like in this world like have like something that's infinitely precise? Like we would need a fraction. Well, not really. Cuz I mean, it's the So, he asks like, is it possible in this world to have something that's really infinite like that?

但这些新的线加起来整体反而更长。而如果你做得越来越精确，它就会越来越长、越来越长。所以，从来没有人真正搞清楚英国的海岸线到底有多长。这是个分形。是啊，但是难道不该……在这个现实世界里，真的有可能存在无限精确的东西吗？我们大概需要一个分数。嗯，其实不太可能。因为我是说，这个——他问的是，在这个世界上有可能存在那种真正无限的东西吗？

结论：海岸线没有唯一的「长度」，只有依赖测量尺度的数值，且尺度越细数值越大、不收敛。分形维数概念正是为刻画这类对象而引入的。

25:44

No, it's not. Cuz I mean, there's only finite space on the earth. Yeah, but even within that, I mean, you just say that even if it has finite area or maybe volume, it can still have like infinite perimeter. Yeah, so the Koch curve theoretically, you know, in math language, if you do it mathematically, it has infinite perimeter but finite area. But keep in mind this is a theoretical creation. It's just in the world of math. And it can't really exist in the universe. But things come pretty close to it in nature.

呃，不可能。因为地球上的空间是有限的。对，但即便在这个范围内，我是说，你刚刚说了，即使它的面积、或者体积是有限的，它仍然可以有无限的周长。是的，所以科赫曲线在理论上，用数学的语言来说，如果你从数学上去做，它有无限的周长但有限的面积。但要记住，这是一个理论上的构造。它只存在于数学的世界里。它在宇宙中并不能真正存在。但自然界里有些东西相当接近它。

关键澄清：无限周长只存在于数学理想世界；物理世界有原子尺度的下限，自然物只是「接近」分形。区分理论构造与物理实在是本段的思想要点。

07 混沌游戏画谢尔宾斯基三角形

26:21 Curran Kelleher

It's not actually infinite. The coastline of Britain is not actually infinite. But it's it really resembles this sort of shape. So, let's see. Yeah. So, any questions about the Koch curve? It's really interesting. So, the next example is the Sierpinski triangle, page eight. So, the Sierpinski triangle is very interesting. Well, You take a big triangle and you add a smaller triangle inside of it like this. So, this is the rule for the Sierpinski triangle. And you know, this is let's call it that or something.

它并不是真的无限。英国的海岸线并不是真的无限长。但它确实非常像这种形状。那么，我们来看看。好。关于科赫曲线还有什么问题吗？真的很有意思。那么下一个例子是谢尔宾斯基三角形，第八页。谢尔宾斯基三角形非常有意思。嗯，你取一个大三角形，然后在里面加上一个像这样的小三角形。这就是谢尔宾斯基三角形的规则。然后，嗯，我们就把这个叫做这个之类的吧。

27:22

So, this is the rule and what you get is three new triangles. And then you apply the same rule to these new triangles. So, that's recursion. When you apply a rule to something that you already applied the rule to, so you apply it again and you get these even smaller things, these smaller triangles, and it goes down infinitely if you do it infinitely. So, imagine this. This So, this is one way of computing it, one way of doing it, one way of looking at the rule. But what I did in the If you look at the lecture notes, I talk about iterated function system.

这就是规则，而你得到的是三个新的三角形。然后你对这些新的三角形应用同样的规则。这就是递归——你对一个已经应用过某条规则的东西再次应用这条规则，于是你再来一次，就得到了这些更小的东西，这些更小的三角形，如果你无限做下去，它就会无限地分下去。所以想象一下。这就是计算它的一种方式，做出它的一种方式，理解这条规则的一种方式。不过呃，我在……如果你看一下讲义笔记，我在里面讲了迭代函数系统。

> 谢尔宾斯基三角形由波兰数学家 Wacław Sierpiński 于 1915 年提出：不断在三角形中挖去中央的倒三角形，留下三个自相似的小三角形。

28:08

So, that means Well, I'll just do it and you'll see what it means. So, let's say we start with a point. say this point here. It could be any point. And we have three possible choices. This is called the chaos game. We have three possible choices of something to do with this point. We either bring it halfway to this point, halfway to this point, or halfway to this point. So, let's say we bring it halfway to this point. We go right here, right in the middle. And what we do in the iterated function system, we do this over and over and over again, picking randomly which one of the three points we go halfway towards.

所以，那意思是呃……好吧，我直接做一遍你就明白了。假设我们从一个点开始。呃，比如这里这个点。它可以是任意一个点。呃，然后我们有三种可能的选择。这个叫做混沌游戏。对这个点，我们有三种可能的操作选择。我们要么把它移到通往这个点的一半处，通往这个点的一半处，或者通往这个点的一半处。那么，假设我们把它移到通往这个点的一半处。我们就走到这里，正中间。而在迭代函数系统里我们要做的，就是把这个过程一遍又一遍地重复，随机挑选这三个点中的哪一个作为我们要走一半路程的目标。

> 「混沌游戏」（chaos game）：从任意起点出发，每步随机选一个顶点并向它移动一半距离，标出落点。规则极简，却能生成精确的分形。

28:48

So, let's say we go to this one next. We go here, halfway. Then we go halfway to this one. Halfway to this one again. Halfway to this one again. And then halfway to this one. then halfway to this one. Halfway to this one. You know, and halfway to this one, halfway to this one, halfway to this one. So, we just plot these points. And the program that I wrote, which is in the handout, does this. It executes this. It just keeps going. It keeps plotting the points. And eventually what you get is the Sierpinski triangle.

那么，假设我们接下来走向这个点。我们走到这里，一半处。然后我们走向这个点的一半处。再走向这个点的一半处。再一次走向这个点的一半处。然后走向这个点的一半处。呃，然后走向这个点的一半处。走向这个点的一半处。你知道的，再走向这个点的一半处，这个点的一半处，这个点的一半处。然后我们就把这些点画出来。呃，我写的那个程序，就在讲义里，做的就是这件事。它执行的就是这个过程。它就一直做下去。它不停地画出这些点。最终你得到的就是谢尔宾斯基三角形（Sierpinski triangle）。

29:26 学生与讲者（对话）

It's crazy. So, page eight is the picture of this is on page eight, right? Yeah. So, when you do it randomly, it doesn't matter which one you choose. No matter what you choose, whenever you do it, it's always going to look the same. So, So, if I like started the program again, but now this time it since it's random, it's going to be like Yeah. Exactly. So, he's getting at So, you run the program again, it's going to choose differently. Maybe it'll choose instead of going to this one first, it'll go to this one first. Say it does it 10 times or 100 times to this one.

太不可思议了。所以第八页是这个的图，图在第八页，对吧？对。所以，当你随机地做这件事时，你选哪一个都无所谓。不管你怎么选，不管你什么时候做，结果看起来总是一样的。所以，所以如果我比如说重新启动这个程序，但这次因为它是随机的，它就会像是……对。没错。所以他想问的是……你重新运行这个程序，它会做出不同的选择。也许它不会先走向这个点，而是先走向那个点。比如说它连续 10 次或 100 次都走向这个点。

30:08 Curran Kelleher

But it the program chooses randomly which one to go to. Going to go to this one and it just chooses randomly. So, he asked, even though it's random, will it still generate the same picture? And the answer is yes, it will. it's crazy. It's just fascinating. And we'll understand this better after we do the next example, which is making a fern, a fractal fern. Which is really cool. So, let's just look at the code quickly. and think about this, is the code for the Sierpinski triangle recursive?

呃，但程序是随机选择走向哪一个的。要走向这个点，它就是随机选的。所以他问，即使它是随机的，它还是会生成同样的图像吗？答案是：是的，会的。呃，太疯狂了。真的非常迷人。等我们做完下一个例子之后，你们会更好地理解这一点——下一个例子是画一棵蕨，一棵分形蕨。那个真的很酷。那我们快速看一下代码。呃，然后想一想，画谢尔宾斯基三角形的这段代码是递归的吗？

本讲最反直觉的结论之一：完全随机的过程生成完全确定的图形。无论随机序列如何变化，点云总收敛到同一个吸引子——谢尔宾斯基三角形。

getting at *phr. v.*
（be getting at）想说的是、意指

30:47

So, what it does is see def draw Sierpinski, that's the thing that draws the triangle. A def A, B, and C are these three points. So, this is like A, B, and C. Def points equals a list of A, B, and C. Current point is just a you know, say start at point A. While this statement while true means execute this code repeatedly. Just keep doing it an infinite number of times until you stop the program. So, it says def next point equals pick from points. And pick from is a function defined above, which pretty much picks at random out of the list that you give it.

它做的事情是这样的，看 def draw Sierpinski，这就是画三角形的那个东西。def 里的 A、B、C 就是这三个点。所以这就相当于 A、B 和 C。def points 等于一个由 A、B、C 组成的列表。current point 就是一个……你懂的，比如说从点 A 开始。while 这个语句，while true 意思是重复执行这段代码。就一直做下去，做无限多次，直到你把程序停掉。所以它写着 def next point 等于 pick from points。而 pick from 是上面定义的一个函数，它基本上就是从你给它的列表里随机挑一个出来。

> 注意代码结构：while true 无限循环 + 随机挑顶点 + 坐标取平均，全程没有函数调用自身——为后文「代码不递归、输出却递归」的讨论埋下伏笔。

31:39

So, that pick from is the thing that picks randomly which one of these to go to. So, it says okay, next point pick from points. So, that gives you the point. And then current point X equals current point X plus next point X divided by two. So, what you're doing is averaging the X coordinates of the current point and the point that you're going to go to next. And if you do that for X and Y, what you do is go that's going halfway between the current point and the next point. That's what that does. Then image.fill pixel that point. So, that's what plots it on the screen.

所以那个 pick from 就是随机决定要走向这几个点中哪一个的东西。所以它说，好，next point 等于 pick from points。这样你就得到了那个点。然后 current point 的 X 等于 current point 的 X 加上 next point 的 X 再除以 2。所以你做的事情就是把当前点和你接下来要走向的那个点的 X 坐标取平均值。如果你对 X 和 Y 都这么做，你所做的就是走到当前点和下一个点的正中间。那句代码做的就是这个。然后 image.fill pixel 那个点。所以那就是把它画到屏幕上的部分。

> **averaging** /ˈævərɪdʒɪŋ/ *v.*
> 取平均值（average 作动词）

> **coordinates** /koʊˈɔːrdənəts/ *n.*
> 坐标

32:16

And the picture here is actually was actually generated by this very program. So, it just keeps going, keeps plotting it. So, here's the question, is this recursive? Is this thing actually recursive? Cuz we said a recursive function is a function that calls itself. let's hold off on that answer until after we do the next one. Any questions so far? So, the next thing we're going to do is the fern. Yeah. So, recursion is generally recursion is something which is defined in terms of itself.

而且这里这幅图实际上就是由这个程序生成的。呃，所以它就一直运行，一直画。那么问题来了，这是递归的吗？这个东西到底是不是递归的？因为我们说过，递归函数是一个调用自身的函数。呃，这个答案我们先放一放，等做完下一个例子再说。到目前为止有什么问题吗？那么我们接下来要做的就是那棵蕨。请讲。所以，递归一般来说是用它自身来定义的东西。

讲者在此抛出全讲的核心问题：不含自调用的代码算不算递归？答案悬置到蕨类例子之后（第 71、76 段）才揭晓，这是有意的教学设计。

hold off on *phr. v.*
暂缓、推迟（做某事）

32:59 学生

Something that loops back on itself.

一种回到自身的循环。

33:06 学生与讲者（对话）

Again? If you apply which one? The recursive rule? It'll become the same thing that it started with? no, not necessarily because each time it applies the rule, there's a slight change. With this factorial, it's not just saying n factorial equals n factorial, it's saying n factorial equals n minus 1 factorial. The factorial part is recursive, but it doesn't loop back on itself exactly. There's some change. And with recursive functions at least, so, the function is defined, it calls itself. Maybe this I don't know if this really answers your question, but say if we didn't have these parts, if the function just was this, if def Fibonacci return Fibonacci of n minus 1 n minus 2, what would happen is it would call itself and just keep going infinitely.

再说一遍？如果你应用哪一个？那个递归规则？它会变成和它开始时一样的东西吗？呃，不，不一定，因为它每次应用规则时，都会有一点小小的变化。以这个阶乘为例，它并不是说 n 的阶乘等于 n 的阶乘，而是说 n 的阶乘等于 n 减 1 的阶乘。阶乘那部分是递归的，但它并不是原封不动地绕回自己。中间是有变化的。至少对递归函数来说，嗯，就是说，函数定义好之后，它会调用自己。也许这个……我不知道这是不是真的回答了你的问题，不过假设我们没有这几部分，假设这个函数就只有这一句，如果 defFibonacci 返回 Fibonacci(n–1) 加 Fibonacci(n–2)，那结果就是它会不断调用自己，一直无限进行下去。

重要澄清：递归不是原样重复。每次调用都有变化（n 变成 n–1），若定义完全绕回自身（n! = n!），就成了无信息的循环，永远算不出结果。

not necessarily *phr.*
不一定、未必（部分否定）

34:11

Which is a problem, so all recursive functions in order to do anything have to bottom out at some point. They have to stop, and that's what this is. Only do this if x if n is greater than 1. If n is less than or equal to 1, return. So, this stops it. So, I mean d– does that answer your question? I don't know. You can ask it again if you want.

这就有问题了，所以所有递归函数要想真正做点什么，都必须在某个点上触底。它们必须停下来，而这就是这部分的作用。只有当 x……当 n 大于 1 时才这么做。如果 n 小于或等于 1，就返回。所以，这就让它停下来了。所以，我是说，这有没有回答你的问题？我也不确定。如果你想的话，可以再问一次。

递归必须有终止条件（bottom out），否则无限自调用、栈溢出。这与数学归纳法必须有「基例」的道理一一对应。

bottom out *phr. v.*
触底；（递归）到达终止条件而停止

34:42

It is. Yeah. Yeah.

是的。嗯。嗯。

34:51

It only functions, computer functions that are that are recursive need to bottom out. There are other kinds of recursion. well, they do. Like natural recursions don't bottom out. Well, they do have to bottom out eventually. Like take for example a tree in nature. It branches and branches and branches, but eventually it just gets to a leaf. And it stops branching. There's some sort of program that's executing inside this tree that's recursive. And there's a certain signal that this program gets when the branch gets to a certain size, I think or something, that signals it to generate a leaf. So, I mean those kind of recursive processes do bottom out.

它只是……函数，计算机里那些递归的函数需要触底。还有别的种类的递归。呃，其实它们也是的。比如自然界中的递归就不会触底。其实它们最终也得触底。比如就拿自然界里的一棵树来说。它不断分叉、分叉、分叉，但最终总会长到一片叶子。然后就不再分叉了。这棵树内部有某种正在执行的程序，而它是递归的。而且当树枝长到某个尺寸的时候，这个程序会收到某种信号，我想大概是这样，这个信号让它去长出一片叶子。所以，我是说，那类递归过程确实是会触底的。

35:35 Curran Kelleher

But this one is actually recursive. We'll see why in a minute, but it doesn't bottom out because it just keeps going forever.

但这一个其实是递归的。过一会儿我们就会看到原因，但它不会触底，因为它就这样永远进行下去。

自然界的递归同样会触底：树反复分枝最终长出叶子。讲者推测树枝达到某尺寸时有信号触发「停止分枝」——把程序的终止条件类比到生物发育。

36:08 学生与讲者（对话）

Yeah, exactly. Exactly. Yeah, that was that was very well put. So, what he's basically said was if you want to get any real manifestation of recursion, it has to bottom out in order to return you the result. But yeah, you can imagine theoretical worlds where it doesn't bottom out. Go on and on forever. Actually, Douglas Hofstadter in the dialogue before this chapter does that. A genie has to ask a meta genie has to ask a meta genie. Did anybody read that? Yeah, I but it is the thing with time it goes in it gets smaller and smaller and Yeah, the time is smaller and smaller.

对，没错。完全正确。嗯，说得非常好。他刚才基本上说的是，如果你想得到递归的任何真实体现，它就必须触底，才能把结果返回给你。不过没错，你可以设想一些理论上的世界，在那里它不触底。一直不停地进行下去。其实道格拉斯·侯世达在这一章前面的对话里就写了这个。一个精灵要去问一个元精灵，元精灵又要去问元元精灵。有人读过那段吗？嗯，我……不过那个和时间有关的地方，它会越来越小，越来越小。对，时间越来越小。

指 GEB 第 5 章前的对话《小和声迷宫》：精灵之上有元精灵、元元精灵层层上推，但每层耗时递减，无限层可在有限时间内完成——类似芝诺级数收敛。

manifestation /ˌmænəfəˈsteɪʃən/ *n.*
体现、表现形式

genie /ˈdʒiːni/ *n.*
（阿拉伯神话中的）精灵，如神灯精灵

well put *phr.*
表述得好（That was well put. 这话说得好）

08 分形蕨与迭代函数系统

36:43

So, like eventually you know, you just repeat an infinite number of intensely small Yeah. Yeah, isn't that crazy? Yeah, it's really something to think about. Yeah. But keep it's like theoretical, but it is very interesting to think about. So, we'll see how this is recursive after we do the fern. So, the fractal fern is next, page nine. Yeah, this is really cool.

所以，最后你知道的，你就在无限多个极短的时间里重复……对。对，这是不是很疯狂？是啊，真的很值得琢磨。对。不过要记住，这算是理论上的，但确实非常有意思，值得去想一想。那么，等我们做完蕨类之后，就会看到这个是怎么递归的。所以接下来是分形蕨，第九页。嗯，这个真的非常非常酷。嗯

37:26 Curran Kelleher

So, the fractal fern it looks beautiful, doesn't it? So, we have this triangle. this isn't the picture in the handout. There's an outside triangle that's black. And then there's an inside triangle that's blue and some other triangles that are the leaves. So, first of all, I want to talk about this notion of a coordinate transformation. You have Say we had a coordinate transformation between this rectangle and this rectangle. What would happen it like it's a function on a point, a two-dimensional point.

那么这个分形蕨，呃，它看着很漂亮，是不是？所以我们有这个这个三角形。嗯，这个……这不是讲义里的那张图。外面有一个三角形是黑色的。然后里面有一个三角形是蓝色的，还有另外一些三角形，就是那些叶子。那么首先，我想讲一讲坐标变换这个概念。你有……假设我们在这个矩形和这个矩形之间有一个坐标变换。会发生什么呢，它就像是作用在一个点上的函数，一个二维的点。

坐标变换：把一个矩形内任意点映到另一（缩小的）矩形内对应位置的函数。它是迭代函数系统（IFS）的基本构件。

coordinate transformation
phr.
坐标变换（把一个坐标系中的点映到另一坐标系的函数）

38:09

So, say you had the point right here, this point. You apply this transformation to the point, and what you get is this point here. so, you have this point here and you apply the transformation to that point, you'll get this point here. It's just like a little copy of this. Okay? So, this is a function, this is the function part of iterated function systems. Iteration is the fact that you just keep doing it. And a system is the fact that there's more than one of them. It's like a bunch. In this case, it's three.

那么，假设你有这里这个点，这个点。你把这个变换应用到这个点上，得到的就是这里这个点。嗯，所以你有这里这个点，你把变换应用到那个点上，就会得到这里这个点。它就相当于这个的一个小副本。好吗？所以，这是一个函数，这就是迭代函数系统里的函数那部分。迭代指的就是你不停地重复做这件事。而系统指的是这样的函数不止一个。是一堆。在这个例子里是三个。

IFS 三要素在此拆解：函数（坐标变换）、迭代（反复应用）、系统（多个变换随机选用）。谢尔宾斯基三角形是三个变换，蕨类是四个。

38:50

I'll talk about it in a minute. So, in and with the fractal fern, what we have is a rectangle like this.

嗯，我等一下会讲到。那么，在分形蕨里，我们有的是，呃，像这样一个矩形。

39:14

So, this the fractal fern is an iterated function system that has four possible functions. This one has three. Each one of those points is one is a function. You know, going halfway to one of those points is a function. But in the fern one, there are four functions. This is one of them. and each function is a coordinate transformation. This function is a coordinate transformation from this outer one, this outer rectangle to this inner rectangle. So, if we had this point in this rectangle and we applied this transformation, we would get this point here.

所以这个……分形蕨是一个迭代函数系统，它有四个可能的函数。这一个有三个。那些点里的每一个都是一个函数。你知道，走到那些点中某一个的一半路程，这就是一个函数。但在蕨类那个例子里，有四个函数。这是其中之一。嗯，每个函数都是一次坐标变换。这个函数是从外面这个、这个外层矩形到里面这个矩形的坐标变换。所以，如果我们在这个矩形里取这个点，然后应用这个变换，我们就会得到这里的这个点。

39:53

So, say we have the middle point of this outer rectangle. And we apply this transformation once. We would get the middle point of this inner rectangle, which is about right here. But here, what we do is we say, "Okay, now this new point is actually in the outer rectangle." And we'll apply the transformation again on that. So, this point in the outer rectangle maps to about this point in the smaller rectangle. And then you say, "Okay, this point is now a point in this one." And you apply it again.

嗯，所以，比如说我们取这个外层矩形的中点。然后我们应用一次这个变换。我们会得到这个内层矩形的中点，大概就在这里。但在这里，我们要做的是说："好，现在这个新的点其实是在外层矩形里。"然后我们再对它应用一次这个变换。那个点。所以，外层矩形里的这个点会映射到较小矩形里大概这个点。然后你说："好，这个点现在是这个矩形里的一个点。"然后你再应用一次。

这个分形蕨即著名的 Barnsley 蕨：数学家 Michael Barnsley 于 1988 年提出，用仅四个仿射变换的 IFS 逼真再现蕨类植物形态。

40:39

And you get this point here and this point here, this point here. And like all these little points. And eventually, they just get smaller and smaller. Because the corner points of these two rectangles are the same point. I think. Probably, yeah. They're the same point. So, let's look at another one. Let's see.

于是你得到这里这个点、这里这个点、还有这里这个点。以及所有这些小点。最后，它们会变得越来越小。因为这两个矩形的角点是同一个点。我想是吧。大概是的，是的。它们是同一个点。那我们再来看另一个。嗯，我看看。

41:16

So, this is the thing that's going to map to. It's It's so It's a line. But think of it as a rectangle. It's a coordinate transformation. So, we go from any point in this outer rectangle to a point on here. So, say if we have this point here in the outer rectangle, it's going to go to the top point of this line. If we have this point here, it's going to go to the bottom point of this line. If we have the center point, it'll go to the center of this line. So, let's imagine an iterated function system with only these two functions.

所以，这就是那个要映射过去的东西。它是……它是……所以它是一条线。但你可以把它想成一个矩形。它是……它是一个坐标变换。所以我们从这个外层矩形里的任意一点，映射到这上面的一个点。所以，比如我们在外层矩形里取这里这个点，它会映射到这条线的最上面那个点。如果我们取这里这个点，它会映射到这条线的最下面那个点。如果我们取中心点，它会映射到这条线的中心。那么，我们来想象一个只有这两个函数的迭代函数系统。

41:55

What's going to happen is and let's say each one is chosen Well, let's say each one is chosen about half the time randomly. Picks between each one. Or no, let's say that it applies this mapping about 90% of the time. So, say we start with this point here, it'll map to this point here, this point here. And it'll just go up and up into here as long as this transformation is being applied. But say it does that like 100 times and then one time it goes down to here. So, say it's all the way up here and we map to this one, it's going to start here.

接下来会发生的是……我们假设每个函数被选中的概率……嗯，我们假设每个函数大约有一半的概率被随机选中。在两者之间随机挑选。不对，我们假设它大约85%……大约90%的时候应用这个映射。那么，假设我们从这里这个点开始，它会映射到这里这个点，再到这里这个点。只要一直在应用这个变换，它就会不断往上、往上，一直到这里。但假设它这样做了大概100次，然后有一次它跑到了这里下面。所以，假设它一路上到了这里，然后我们映射到这一个，它就会从这里开始。

单变换行为演示：若 90% 的概率应用「向上映射」，点会沿茎向上爬升，偶尔被其他变换拉回底部重新开始——概率决定图形各部分的密度。

mapping /ˈmæpɪŋ/ *n.*
映射（数学：点到点的对应规则）

42:38

And it's going to map here, here, here. It's going to go up. And say we stop at this point, the computer decides to map to here. It's about, you know, 3/4 up. It's going to go to 3/4 up here. So, because it's random, if you do this just forever, it's going to eventually fill in pretty much every point on this line. Which is very interesting to think about. Any questions so far? So, let's have this rectangle. All four of the transformations are from the outer rectangle, this big one, to one of the ones inside, one of the smaller ones inside.

然后它会映射到这里、这里、这里。它会往上走。假设我们在这一点停下，计算机决定映射到这里。它大概在，你知道，四分之三的位置。那它就会映射到这里四分之三的位置。所以，因为这是随机的，如果你一直这样做下去，最终它几乎会填满这条线上的每一个点。这想起来非常有意思。到目前为止有什么问题吗？那么，我们来看这个矩形。这四个变换都是从外层矩形、就是这个大的，映射到里面某一个、里面某个较小的矩形。

随机性在此起「遍历」作用：无限次迭代后，点几乎会落到吸引子上的每一点。这是 IFS 收敛定理的直观版本。

43:24

So, say we have this point up here and we apply this transformation to this one. It's going to go to this point here. Make sense? And if we have this one, it's going to go to this point here. So, the transformation is pretty much going like this. So, say our IFS, iterated function system, has these three. Say we go about halfway up here and we choose to apply this mapping. It's going to go about here, the middle of this one. And then we apply this mapping like a bunch of times. Just keep going up and up and up.

所以，假设我们在上面这里有个点，我们对它应用这个变换，映射到这一个。它会跑到这里这个点。说得通吗？如果我们取这一个，它会跑到这里这个点。所以，这个变换大概是这样走的。那么，假设我们的IFS，也就是迭代函数系统，有这三个函数。假设我们走到这里大概一半高的位置，然后我们选择应用这个映射。它会跑到大概这里，也就是这个的中间。这一个的中间。然后我们把这个映射应用很多次。就一直往上、往上、往上。

44:03

We apply this mapping again. But it's so closer to the top this time, so it's going to be out here, closer to this the top of this rectangle. And then we apply it again and again and again. And maybe it only goes up one. So, it'll be down here. So, eventually, it's going to fill in this rectangle with this pattern. It's going to look like this. Okay? And this, since that's, you know, every point in here is probably going to get applied to this mapping a bunch of times. So, what we're going to get is this fern structure.

我们再应用一次这个映射。但这次它离顶部更近了，所以它会跑到这外面来，更靠近这个矩形的顶部。然后我们再一次又一次地应用它。也许它只往上走了一格。那它就会在这下面。所以，最终，它会用这个图案填满这个矩形。看起来会是这样。好吗？而且这个，因为，你知道，这里面的每一个点大概都会被这个映射作用很多次。所以我们得到的就是这个蕨类结构。

44:52 学生与讲者（对话）

Okay. Isn't that really cool? You're saying that all of the points in those smaller in that little small triangle the points that you have in there applied to like the biggest one to like different parts of the biggest one or do you like Are you Are you talking about the Sierpinski triangle or the fern? The fern. Okay. Like you have like a bunch of points in that little like Do those Do all those like little points that correspond like one of the either one either like parts of the fern Yeah. It's like they're repeated, but only smaller and each one has the same amount of points that they have. Yeah, exactly.

好。是不是特别酷？你是说，那些小的……那个小三角形里的所有点，呃，你在里面取的那些点，被映射到最大的那个上面，映射到最大那个的不同部分？还是说你……你是在说谢尔宾斯基三角形还是蕨类？蕨类。好的。就是说你在那个小的里面有一堆点……那些……所有那些小点是不是都对应……对应蕨类的某一部分？是的。就好像它们是重复的，只是更小，而且每一个都有相同数量的点。是的，没错。

45:32

I mean, if you like get smaller and smaller, how can you get into the same amount of stuff? Right, you can't actually. And this is Okay, I'll go through an example. Say we have, instead of just applying it to one point, we apply it to a bunch of points. Say all the points on this line or almost all the points on there. Say we apply this mapping a bunch of times till we get this one, this one, this one, this one, this one. And say we apply this mapping to all of these points. It's going to map to here.

我是说，如果你越变越小，怎么可能装进同样多的东西呢？对，实际上不能。这就是……好，我举个例子。假设我们不只是把它应用到一个点上，而是应用到一堆点上。比如这条线上的所有点，或者说几乎所有的点。假设我们把这个映射应用很多次，直到我们得到这一个、这一个、这一个、这一个、这一个。然后假设我们把这个映射应用到所有这些点上。它会映射到这里。

46:13 Curran Kelleher

And then we do that again and again and again. And say and then what we have, say all those points eventually are going to get mapped to this one. So, you have a whole little copy of this thing in here. So, you have these little branches. And then that is going to be mapped up here. And you have the branches, branches, branches. And then since you have the smaller branches here, you have two levels down. And this whole thing gets mapped back onto here, including the new branches. And it gets filled in as it goes on.

然后我们再一次又一次地这样做。然后假设……我们得到的是，所有那些点最终都会被映射到这一个上。所以你在这里面就有了这整个东西的一个小副本。于是你就有了这些小的，嗯，分枝。然后那个又会被映射到上面这里。你就有了分枝、分枝、分枝。然后因为你这里有更小的分枝，你就往下多了两层。而这整个东西又被映射回这里，包括那些新的分枝。随着过程继续，它就被填满了。

自相似的生成机制说透了：整体（连同其全部细节）被映射进自身的子区域，因此每根分枝都是整棵蕨的缩小副本，副本里又含副本。

46:54

But it doesn't just keep going up to here and just like say we were to apply this mapping 100% of the time. It would just go up here and go nowhere else. But say we apply this mapping like 7% of the time, it'll go up different lengths and then it'll map back down there and go up, map back down. And then if we add this transformation going from this one to this one, we have this. And it's recursive. The reason why it's recursive is because the mappings map onto themselves eventually. If we look at the code, it's really similar to the Sierpinski triangle code.

但它不会只是一直往上到这里，就好像……假设我们100%的时候都应用这个映射。那它就会一直往上走到这里，哪儿也不去了。但假设我们大概7%的时候应用这个映射，它就会往上走不同的长度，然后再映射回下面那里，再往上，再映射回下面。然后如果我们再加上从这一个到这一个的这个变换，我们就得到了这个。而且它是递归的。它之所以是递归的，是因为这些映射最终会映射到它们自己身上。如果我们看代码，它跟谢尔宾斯基三角形的代码非常相似。

全讲点题句：递归性在于「映射最终映回自身」，而不在于代码调用自身。递归可以存在于数学结构层面，与其代码载体分离。

47:41

The code itself is not recursive, but the mappings are. And that's what makes this whole thing recursive. So, yeah, let's just look at the code a little bit. class fern. So, draw fern. While true, so that means execute this. Just keep doing it over and over and over. Pick from these list of transformations. So, the first one says maps to the stem. That means it goes from here to here, to the stem. The second one maps to the left branch, meaning it goes from here to this one. The third one maps to the right branch, it goes down here.

代码本身不是递归的，但映射是递归的。这才是让整个东西变得递归的原因。嗯，那么，好，我们稍微看一下代码。呃，class fern（蕨类类）。那么，draw fern（画蕨类）。while true，也就是说执行这个。就一遍又一遍地一直做下去。从这个变换列表里挑一个。所以，第一个写的是映射到茎。意思是它从这里到这里，到茎。第二个映射到左边的分枝，意思是它从这里到这一个。第三个映射到右边的分枝，它跑到下面这里。

48:25

And the fourth one maps from here to here. It just goes up like that. And following that, it says pick from a list of functions and then a list of probabilities. It says.01,.07,.07,.85. So, these are the probabilities at which these mapping functions are going to be chosen. If you choose them all equally, then the chances are very low that this mapping is going to occur more than like five times or something. So, it just go up here and get mapped. So, it'd be a very sparse fractal. it would be most of the points would be like down here.

第四个是从这里映射到这里。它就是像这样往上走。接下来它说，从一个函数列表和一个概率列表里挑选。上面写着 .01、.07、.07、.85。所以，这些就是这些映射函数被选中的概率。如果你让它们的概率都相等，那么这个映射连续出现超过大概五次的概率就非常低。所以，它就会往上走一点然后被映射走。这样得到的分形就会非常稀疏。呃，大部分的点都会在下面这里。

四个变换的概率 .01/.07/.07/.85 是精心调校的：概率决定各区域落点密度。若四者均分，向上盘旋难以连续发生，图形会稀疏难看。

probabilities /ˌprɑːbəˈbɪlətiz/ *n.*
概率（probability）

sparse /spɑːrs/ *adj.*
稀疏的、零落的

49:06

It just didn't look very good. I tried it. I wish I could like code it and show you. So, the.01 means that the mapping to the stem happens 1% of the time. The mapping to each of the branches happens 7% of the time. And the mapping up and spiraling happens 85% of time. So, I'm going to take a break now for 10 minutes. Do you have any questions? So, I guess we'll start up again. Before we leave these iterated function systems, I want to point out how the Sierpinski triangle is pretty much the same thing as the ferns in terms of mappings.

看起来就很不好看。我试过。真希望我能现场写代码演示给你们看。嗯，所以 .01 表示映射到茎的情况有1%的概率发生。映射到每个分枝的概率是7%。而向上盘旋的那个映射有85%的概率。那么，我现在要休息10分钟。你们有什么问题吗？好，我想我们再开始吧。在我们离开迭代函数系统这个话题之前，我想指出，从映射的角度看，谢尔宾斯基三角形跟蕨类基本上是同一回事。

讲者总结：谢尔宾斯基三角形与蕨在映射意义上是同一类对象（IFS），区别只在变换的个数、形状与选取概率。

spiraling /ˈspaɪrəlɪŋ/ *v.*
盘旋（上升）；spiral 螺旋（运动）

50:05

With the Sierpinski triangle, we have these three mappings. one of them maps from this triangle to this triangle here. One of the maps from the big triangle to this triangle. And this the other one maps from this triangle to this triangle. And each one of them are applied with equal probability. so, if you think about it, going halfway from any of these points to one of the other points is essentially mapping it down. So, say you have this one this line. You apply the function to all these points.

对于谢尔宾斯基三角形，我们有这三个映射。嗯，其中一个是从这个三角形映射到这里这个三角形。其中一个映射是从大三角形映射到这个三角形，另一个是从这个三角形映射到那个三角形。而且每一个映射被应用的概率都相等。嗯，所以，如果你想一想，从这些点中的任意一个走到另一个点的一半距离，本质上就是把它映射下去。比如说，你有这条线。你对所有这些点应用这个函数。

09 代码不递归，递归在映射里

50:46 学生与讲者（对话）

What it does is maps it down to that triangle. So, these you could see these this these triangles map onto themselves recursively. And that's why it actually it there is a recursion going on here, but it's not in the code itself. The code itself is just repetitive, but what it's repeating is these recursive mappings onto themselves. Yeah. So, there's always more than one way to like represent an algorithm? There's always more than one way to represent an algorithm. Yeah, it's really fascinating.

它所做的就是把它映射到那个三角形上。所以，你可以看到这些三角形是递归地映射到它们自身的。这就是为什么这里其实确实有递归在发生，但递归不在代码本身里。代码本身只是重复性的，但它重复的正是这些映射到自身的递归映射。是啊。所以，表示一个算法总是有不止一种方式？表示一个算法总是有不止一种方式。呃，是的，这真的很迷人。

51:23

Especially with these fractals and things, it seems like there's always another way to do it, to achieve the same result. And that's one thing especially Sierpinski's triangles are really amazing. So, what if in the new code you don't have like a In my what? In when you like do the code, you don't have like some maybe you don't have a recursive function. So, how can you like figure out like without running the code that your output is so kind of being recursive? Ah, so how can you figure out if the output is going to be recursive without running the code?

尤其是在这些分形之类的东西上，似乎总有另一种做法能达到同样的结果。而这一点尤其是谢尔宾斯基三角形真的很神奇。那么，如果在新代码里你没有那种——在我的什么？在你写代码的时候，你可能没有递归函数。那么，你怎么能在不运行代码的情况下，判断出你的输出是某种递归的呢？啊，所以，怎么才能在不运行代码的情况下判断输出会是递归的？

核心论断：代码本身只是重复（迭代），递归发生在「图形被映回自身」这一数学层面。同一算法可有多种表示，递归性属于结构而非某种写法。

repetitive /rɪˈpɛtətɪv/ *adj.*
重复性的（与 recursive 递归的相对，本讲关键对比）

51:57

You would now you have the code, but it has no recursive function. Right. Yeah, you have the code no recursive function. If you can't run it, how can you find out that the output is going to be recursive without actually running it? So, he said since the code itself is not recursive, how do you know that the output is going to be recursive? Even if you don't have any recursive function. Even without any recursive functions. Right. I mean, the thing is when you write the code, if you realize that what the code is doing is mapping these two-dimensional mappings you can sort of think in your head, okay.

现在你有了代码，但它里面没有递归函数。对。是的，你有代码，没有递归函数。如果你不能运行它，你怎么能在不实际运行的情况下发现输出会是递归的？所以，他说，呃，既然代码本身不是递归的，你怎么知道输出会是递归的？即使你没有任何递归函数。即使没有任何递归函数。对。我的意思是，问题在于，当你写代码时，如果你意识到代码在做的事情是映射这些这些二维映射，你可以在脑子里大致想象，好的。

学生的追问触及可计算性问题：能否不运行程序就判定其输出具有递归性？这个问题把讨论从编程引向元层面。

52:38

These mappings are going to be applied with equal probability. Oops. So, just by mentally extrapolating in your head, so you're sort of stepping out of the system. You're saying, okay, what's the thing actually doing? But are you going to like do that? I mean, can you can't you just like do while you're like in the system? While you're in the system? Well, being in the system in this case is actually running the code and like being a computer program running it. Stepping out of the system, but what I mean by that is like taking a higher level view of what's going on.

这些映射将以相等的概率被应用。哎呀。所以，只要在脑子里做心理外推，你就相当于跳出了这个系统。你在说，好的，这东西实际上在做什么？但你会那样做吗？我是说，你就不能在你身处系统之内的时候做吗？在系统之内的时候？嗯，在这个情况下身处系统之内其实就是运行代码，就像是一个正在运行它的计算机程序。跳出系统，我的意思是对正在发生的事情采取一个更高层次的视角。

「跳出系统」（stepping out of the system）是 GEB 的招牌概念：不在系统内部按规则执行，而是上升一层审视系统整体在做什么。

stepping out of the system
phr.
跳出系统（GEB 核心概念：从更高层次审视系统）

53:15

So, when you abstract from all of these like different what is doing the same thing, you get like this higher level which actually does your recursion. Yeah, exactly. Exactly. Yes, that's it. So, he said, when you abstract away from all the details of what's going on with the actual the functions you begin to perceive this higher level thing which is itself recursion. Yeah, and it has recursion in it. So, if you if you abstract your thought process significantly enough you'll be able to logically tell that it's going to create this recursive shape.

所以，当你从所有这些不同的、在做同样事情的东西中抽象出来，你就得到了这个更高的层次，而它实际上做的就是你的递归。是的，正是如此。完全正确。对，就是这样。所以，他说，当你从实际函数中正在发生的所有细节里抽象出来，你就开始感知到这个更高层次的东西，而它本身就是递归。是的，而且它里面有递归。所以，如果你把你的思维过程抽象到足够高的程度，你就能在逻辑上判断出它会生成这个递归的形状。

abstract /æbˈstrækt/ *v.*
抽象、提取共性（动词重音在第二音节，区别于形容词）

perceive /pərˈsiːv/ *v.*
感知、察觉、领悟

53:56

Yeah. But there's like no like simple algorithm or like any program you have like see if it's like recursive or not. Yeah, so you mean like a test? Like Yeah, like test. No, you have to think about it. That's what humans can do. Like there's no strict test that could be applied to some computer program that would tell you whether or not it's going to create a recursive result. Cuz in this case, there's no recursive function. So, the computer can't know like, okay, hm well, this Can I abstract into Only humans can do that. Yeah, so far.

是的。但是好像没有什么简单的算法或者程序，能让你看出它是不是递归的。是的，所以你是说像一种测试？比如是啊，像测试。不，你必须自己去思考。这是人类才能做的事。就是说，没有一个严格的测试可以应用到某个计算机程序上，来告诉你它会不会产生一个递归的结果。因为在这种情况下，没有递归函数。所以计算机没法知道，好吧，嗯，这个……我能不能抽象成只有人类能做到这一点。是的，目前为止是这样。

讲者断言：不存在判定「输出是否递归」的机械测试，只有人能通过抽象洞察做到——这一立场与停机问题、哥德尔不完备定理的精神一脉相承。

54:29 Curran Kelleher

And it's not coded up as a system of mappings. It's just these simple functions. I don't know. So, yeah. So, it's really cool. So, yeah, if you just think about mapping and you map it over here, all the stuff that was in here that goes infinitely down is now like in like this little sub section of this little triangle. And you just yeah. It's a recursive structure. So, let's go on to the next example, which is very interesting, the Mandelbrot set. First, now that we have this up and running, I just want to run the programs that are in the handout to give you a sense of what's going on.

而且它并没有被编写成一个映射系统。它只是这些简单的函数。我不知道。所以，是的。所以，这真的很酷。所以，是的，如果你就这样想着映射、映射、映射，然后你把它映射到这边，原本在这里面无限延伸下去的所有东西，现在就像是在这个小三角形的这一小块子区域里了。然后你就……是的。这是一个递归结构。那么，我们进入下一个例子，非常有意思，曼德博集合。嗯，首先，既然我们现在把这个跑起来了，我想运行一下讲义里的那些程序，让大家感受一下是怎么回事。

55:17

So, the recursive transition network. Let's run it. See what happens. It's going to generate 100 sentences. Cuz if you look at the code there's a print statement that goes 100 times. yeah, page five. It says 0 to 100.each print line call the function fancy noun. So, this is just fancy notation in the Groovy programming language to Oh, crap. It's not working. Oh, well. I'll just try to get this to work as I'm talking. I can't do the examples. So, Yeah, I've been having problems with this stay all day.

那么，递归转移网络。我们来运行一下。看看会发生什么。它会生成 100 个句子。因为如果你看代码，呃，那里有一个打印语句执行 100 次。呃，是的，第五页。上面写着 0 到 100，each，print line，调用函数 fancy noun。所以，这只是 Groovy 编程语言里的花哨写法，用来呃，糟糕。它没跑起来。唉，算了。我一边讲一边试着把它弄好。我做不了这些例子。所以，嗯是啊，我一整天都在被这东西困扰。

10 曼德博集合与逃逸迭代

56:07

So, the next example and the last example is the Mandelbrot set. You guys probably have heard about the Mandelbrot set. It's very famous, probably one of the most famous fractals of all. It's called Mandelbrot because this guy Mandelbrot really loved fractals and tried to communicate the notion of fractals to the world and this is really a great fractal. So, it'll get a little bit mathy, but that's okay. so, we'll have it's in the complex plane. So, that means that we have this plane where these are the real numbers and these are the imaginary numbers. I think this is where it goes.

那么，下一个例子，也是最后一个例子，是曼德博集合。嗯，你们大概都听说过曼德博集合。它非常有名，可能是所有分形里最有名的之一。嗯，它叫曼德博是因为曼德博这个人呃真的很喜欢分形，并且试图把分形的概念传达给全世界，而这确实是一个很棒的分形。所以，接下来会有一点数学味，但没关系。嗯，所以，我们会在复平面上讨论。也就是说我们有这样一个平面，这边是实数，这边是虚数。我想是这么放的。

曼德博集合得名于 Benoit Mandelbrot（1924–2010），IBM 研究员、「分形几何之父」，1980 年首次绘出该集合图像，是最著名的分形。

mathy /ˈmæθi/ *adj.*
（口语）数学味重的、偏数学的

complex plane *phr.*
复平面（横轴为实数、纵轴为虚数的平面）

imaginary numbers *phr.*
虚数（含 i = √−1 的数）

56:55

So, the Mandelbrot set is defined by the function Z = Z ^ 2 + C. And when you square a complex number well, first of all, first of all, a complex number is a point in this plane. So, say this point is you know, this is B, say, and this is A. This point is represented by A + B I. And I is the square root of −1, which is not really possible. That's why it's called imaginary. So, if you multiply two complex numbers together it's not really that simple. You have to do like the actual multiplication out. When you get is A AC + you know, I don't know if I should do it all out.

嗯，所以，曼德博集合由函数 $Z = Z^2 + C$ 定义。嗯，当你对一个复数取平方时，呃，首先，首先，一个复数就是这个平面上的一个点。比如说，这个点，你知道，这是 B，比如说，这是 A。这个点用，嗯，A + Bi 表示。而 i 是 −1 的平方根，这实际上是不可能的。所以它才被叫做虚数。所以，嗯，如果你把两个复数相乘，嗯，其实并没有那么简单。你得把乘法实际展开算出来。你会得到 AC + 之类的，我不知道该不该全部算出来。

复数 a+bi 对应平面上一点，i 是 −1 的平方根。复数乘法兼有旋转与缩放效果，正是这种「不那么线性」的运算造就了图形的复杂性。

square root *phr.*
平方根

58:16

It could it'll take a little bit of time. But you get this sort of a mapping function essentially that's not really linear. It's not really that predictable what it's going to do. So, you have this point here. And Okay, so the way the algorithm works, you take a point on the complex plane. That gets C gets the value of that point. And then Z starts off at zero. Just not Just zero. And you apply this function a bunch of times. So, say we apply this function once. Z = Z ^ 2 + C. So, 0 ^ 2 is nothing + C. So, the first iteration, Z gets the value C. So, Z is going to be there. Then you apply the function again.

呃，这可能会花一点时间。但你会得到这样一种映射函数，本质上它不是线性的。它做出来的结果并不那么容易预测。所以，你在这里有这个点。那么，好的，算法是这样工作的：你在复平面上取一个点。那个点的值就赋给 C。然后 Z 从零开始。就只是，就只是零，零。然后你把这个函数应用很多次。比如说我们应用一次这个函数。呃，$Z = Z^2 + C$。所以，0^2 是零，加上 C。所以第一次迭代，Z 得到值 C。所以 Z 会在那里。然后你再应用一次函数。

59:09

With and this Z is now the new Z that we just got. So, Z = Z ^ 2 + C. So, Z ^ you apply the squaring function. And you add C. You add the real and imaginary components of C. So, it just maps this point to some other point over here, say. and then you then you apply this again and again and again and again. So, apply it here. Here. So, I don't know if I'm going too fast, but like we get this point after applying the function to this. So, now this is the new Z. And then this the new Z loops back around.

用的这个 Z 就是我们刚刚得到的新的 Z。所以，$Z = Z^2 + C$。所以，Z 的平方——你应用平方函数。然后你加上 C。你加上 C 的实部和虚部。所以，它就把这个点映射到别的某个点，比如说这边。嗯，然后你再一遍又一遍地应用这个。所以，在这里应用。这里，这里。所以，我不知道我讲得是不是太快了，不过，对这个点应用函数之后我们得到了这个点。所以现在这就是新的 Z。然后这个新的 Z 又循环回去。

算法框架：像素对应的复数作为常数 C，Z 从 0 出发反复计算 $Z \leftarrow Z^2 + C$，观察轨道是留在原点附近还是飞向无穷。

linear /ˈlɪniər/ *adj.*
线性的；not linear 指变化不成简单比例、难预测

59:53 学生与讲者（对话）

And so, we apply it now Z squared plus C. C is always going to be the initial point. But Z will be changing. And it maps to here, for example. And now the new point is Z. And you do Z squared plus C again. And that maps over here. So, I wrote a little applet that demonstrates the this mapping and what it actually looks like, yeah. And the C can be anywhere you like? Z can be anywhere? the starting point. The starting point could be anywhere. Yeah. But when you actually run the algorithm, you only go from –2 to 2.

于是我们现在做 Z 平方加 C。C 永远是初始的那个点。但是 Z 会不断变化。然后它映射到这里，举个例子。现在新的点就是 Z。然后你再做一次 Z 平方加 C。它映射到这边。所以，呃，我写了一个小程序来演示这个映射，以及它实际看起来是什么样子，是的。那 C 可以是任意位置吗？Z 可以在任何地方？起始点。起始点可以在任何地方。是的。但当你实际运行这个算法时，你呃只从 –2 到 2。

60:39 Curran Kelleher

–2 2. Because that's the only region in which interesting things happen with this fractal. So, I didn't finish explaining the algorithm. So, say a point in here, a point that's very close to the to the origin, 0 0. What it tends to do is spiral inward as you iterate it. But a point out here, what it tends to do or definitely a point outside of two, what it tends to do when you apply this function is to get mapped out here and to get mapped like over there and just go like really far away. So, what we do to render the Mandelbrot set is apply the function a number of times.

呃，–2 到 2。因为那是这个分形唯一会发生有趣现象的区域。所以，我还没讲完这个算法。那么，比如说这里面的一个点，一个非常靠近原点 0,0 的点。它往往会随着迭代向内螺旋。但是外面这里的一个点，它往往会——或者说肯定是二以外的点，当你应用这个函数时，它往往会被映射到这边，被映射到那边，然后就跑到非常远的地方去。所以，我们渲染曼德博集合的做法，就是把这个函数迭代应用很多次。

只需考察 –2 到 2 的区域：可以证明轨道一旦跑出以原点为圆心、半径 2 的圆，就必然发散到无穷，不会再回来。

origin /ˈɔːrədʒɪn/ *n.*
（坐标系的）原点；起源

spiral inward *phr.*
向内盘旋、螺旋式收敛

61:28

And if the point eventually goes outside this circle at the origin of radius two, sorry, it's a bad circle. If the point eventually goes outside of this circle, we stop performing the function because we know that after it gets outside, it's going to keep going forever out. It has escaped. It's called the escape iterations. The number of iterations it takes to escape outside of this circle. These are escape iterations. So, what we do is we color the point depending on its escape iterations. So, when we actually And the other part of it is if it never escapes, we color it black.

如果这个点最终跑出了这个以原点为圆心、半径为 2 的圆，抱歉，这个圆画得不太好。如果这个点最终跑出了这个圆，我们就停止继续迭代，因为我们知道它一旦跑出去之后，就会一直往外跑，永远不回来了。它逃逸了。这叫做逃逸迭代次数。也就是它跑出这个圆所需要的迭代次数。这些就是逃逸迭代次数。所以我们的做法是，根据一个点的逃逸迭代次数来给它上色。所以，当我们实际上——另一部分是，如果它永远不逃逸，我们就把它涂成黑色。

「逃逸迭代次数」：轨道跑出半径 2 的圆所需的步数，是着色依据。曼德博集合图像外围著名的彩色光晕正是由逃逸速度不同造成的。

radius /ˈreɪdiəs/ *n.*
半径

escape iterations *phr.*
逃逸迭代次数（曼德博集合渲染术语：轨道逃出圆所需步数）

62:10

To actually test that, we just iterate a certain number of times, say like 500 times or like 70 times. And if it hasn't escaped after that many iterations, we sort of assume that it's never going to. It's not valid. So, it's an approximation to the actual set, but we have to do it to render the actual thing. Then we color it black. So, when we actually render it, what we do is go through each pixel on the screen and apply this algorithm. So, for this point, this point becomes C. And we iterate, test, iterate, test, iterate, test. And what if the test what we're testing is if it's outside of the circle. If it's outside of the circle, we assign it a color based on the number of iterations it took.

要真正判断这一点，我们就迭代固定的次数，比如说 500 次，或者 70 次。如果迭代了这么多次之后它还没逃逸，我们就大致认为它永远不会逃逸了。这不严谨。所以这只是对真实集合的一个近似，但为了把这东西渲染出来，我们不得不这么做。然后我们就把它涂成黑色。所以，实际渲染的时候，我们的做法是遍历屏幕上的每一个像素，对它应用这个算法。所以，对于这个点，这个点就成了 C。然后我们迭代、检测，迭代、检测，迭代、检测。我们检测的是它是不是跑到了圆外面。如果它在圆外面，我们就根据它所用的迭代次数给它分配一个颜色。

永不逃逸的点才属于集合本身（涂黑）；程序只能迭代有限次（如 500 次）作近似——集合的精确边界在计算上永远无法完全判定。

approximation /əˌprɑːksəˈmeɪʃən/ *n.*
近似、近似值

render /ˈrɛndər/ *v.*
渲染、绘制（图形学术语）；亦有「使成为」义

pixel /ˈpɪksəl/ *n.*
像素

62:54

And we do that for each one. So, we keep going and say this one, for example, is going to spiral in. It's not a continuous line. It just applies the function. And it spirals in. So, that's never going to escape, so we're going to color it black. And this applet is excellent showing this. So, I implemented this algorithm in Java. And it's going to output the point and it's going to draw the lines between each point. So, say it starts here, it's going to draw a line here and here.

每一个点我们都这么做。我们继续往下走，比如说这个点，它会往里螺旋收敛。这个——它不是一条连续的线。它只是在不断应用这个函数。然后它就螺旋收敛进去了。所以这个点永远不会逃逸，那我们就把它涂成黑色。还有这个小程序，把这一点展示得非常好。我用 Java 实现了这个算法。它会输出这些点，并且在每两个点之间画出连线。比如说它从这里开始，它会画一条线到这里，再到这里、这里、这里。

applet /ˈæplət/ *n.*
（尤指 Java 的）小应用程序

implemented /ˈɪmpləmɛntɪd/ *v.*
实现、落实（implement：把设计变成可运行的代码）

63:42

So, what we're going to see is really cool. So, this is this is what we get in the end. And as I move my mouse around, it does these tests. So, here we can see that there's only one iteration. The one iteration gets the color light blue. And then here there are two iterations. And they're listed up there, one two. So, you can see all the way wherever it's this color, there are only two iterations before the point gets outside of this circle. Two, I think. So, as we go deeper in, there it takes three iterations, four or five, six, seven.

所以我们将会看到非常酷的东西。这就是我们最终得到的结果。当我移动鼠标的时候，它就在做这些检测。在这里我们可以看到只有一次迭代。一次迭代得到的颜色是浅蓝色。然后在这里是两次迭代。上面也列出来了，一、二。嗯，所以你可以看到，凡是这个颜色的地方，都是只经过两次迭代这个点就跑出了这个圆。二、二、二，我想是这样。所以，随着我们越往里走，这里需要三次迭代，四次、五次、六次、七次。

64:22

A lot of iterations. So, all the points that are colored, they do eventually escape. But all the points that are black don't escape. And so, the patterns that are inside are really interesting. See these in the middle, they just sort of spiral in. Keep going. the ones here, there's some looping pattern. See it? The function loops back on itself. So, it's sort of recursive. Like I don't really understand how it's recursive, but it sort of is. So, if you go out to this other nub, the second nub out, it gets this it's really cool shape.

迭代次数很多。所以所有有颜色的点，它们最终都会逃逸。但所有黑色的点都不会逃逸。而里面这些图案真的很有意思。看中间这些，它们就是一直往里螺旋。一直转下去。嗯，这边这些点，有一种循环的模式。看到了吗？这个函数绕回到了它自己。所以它有点递归的意思。我其实不太说得清它到底怎么个递归法，但它确实有那么点意思。如果你跑到旁边这个凸起上，也就是往外数第二个凸起，就会得到这个——形状真的很酷。

64:59

So, what we could do is zoom in actually on this. It redraws it. So, it's calculating this algorithm for every single point as we speak. It's doing it really fast. So, we can see all these really cool shapes that generate this fractal. And it's just crazy.

我们可以做的是，实际放大这里看看。它会重新绘制。所以就在我们说话的这会儿，它正在对每一个点计算这个算法。速度非常快。嗯，所以我们能看到所有这些生成这个分形的超酷图形。简直太疯狂了。

集合边界附近的轨道呈现螺旋、循环等复杂动力学行为；边界本身是无限复杂的分形曲线，可无限放大并不断出现新细节乃至整体的小副本。

nub /nʌb/ *n.*
小凸起、疙瘩；(the nub of) 要点

65:31

So, we can sort of see recursion here. Like it looks like there's definitely some recursive thing going on. And the recursive part of this is the fact that Z gets applied to itself. Z of it like this is the nth Z is equal to the Z that came before it squared plus C. And then this Z is reframed as the old Z. And this reframing and reapplying of the function is what makes it recursive. So, yeah, it sort of makes sense. Let's just go through the code quickly. And then I want to show some really cool things.

所以我们在这里多少能看出递归。看起来这里面肯定有某种递归的东西在发生。而这里面递归的部分在于，Z 被应用到了它自己身上。这个 Z——比如说第 n 个 Z 等于它前面那个 Z 的平方加上 C。然后这个 Z 又被重新当作旧的 Z。这种把结果重新代入、再次应用函数的做法，正是它递归的地方。所以，是的，这多少讲得通。我们快速过一遍代码吧。然后我想给大家看一些非常酷的东西。

曼德博集合的递归性同样在于自指：Z 的输出被重新代入为下一轮输入。「重新框定再应用」与 IFS 的映射映回自身是同一种结构。

reframed /ˌriːˈfreɪmd/ *v.*
重新框定、重新定义
（reframe：换一种框架看待）

66:20

Oh, where did it go? So, yeah, in the black part, it just recurses infinitely. And the code itself is not recursive, but the mapping function, again, is what's recursive. So, yeah, the black parts are in the set. So, the real actual Mandelbrot set is just the black points. We just color the other points so it looks pretty. So, let's look at the code. def draw Mandelbrot. See that there? window.set –2 2. So, it just sets the window, the plotting range to be in this range. For every X pixel in the height of the window or the width of the window and every Y pixel in the height, C.real and C.imaginary get the X and Y values on the screen.

哦，跑哪去了？所以是的，在黑色的部分，它就是无限地递归下去。代码本身并不是递归的，但那个映射函数，再强调一次，才是递归的那部分。所以，是的，黑色的部分属于这个集合。所以真正的曼德博集合其实只有那些黑色的点。我们给其他点上色只是为了好看。那我们来看代码。嗯，def draw Mandelbrot（绘制曼德博集合）。看到了吗？嗯，window.set –2 2。这只是设置窗口，把绘图范围设成这个区间。对于窗口宽度中的每一个 X 像素，以及高度中的每一个 Y 像素，C.real 和 C.imaginary 取的就是屏幕上的 X 和 Y 值。

67:17

So, what we're doing there is saying basically the pixel gets this actual point in the space that we're working the complex plane. def iterations equals calculate iterations C. And we pass it C. Keep in mind, this is C. The starting point is C. So, let's just stay inside this function for now. def color equals calculate color iterations. So, we're calculating the color based on the number of iterations it took to escape. And then fill the pixel with the color. That plots it on the screen.

我们在那里做的事情，基本上就是说：这个像素对应到我们所处的这个空间——也就是复平面——里的这个实际的点。def iterations = calculate iterations C（迭代次数 = 计算迭代次数(C)）。我们把 C 传进去。记住，这就是 C。起始点就是 C。那我们先待在这个函数里面看。def color = calculate color iterations（颜色 = 根据迭代次数计算颜色）。所以我们是根据逃逸所用的迭代次数来计算颜色。然后用这个颜色填充这个像素。这就把它画到屏幕上了。

67:56

So, let's look at the calculate iterations function. int calculate iterations. This notation means that it will return an integer, a number. It takes a complex number C. So, Z.real, Z.imaginary start at zero. Start there. And then iteration starts off at zero. So, this is we're going to count how many iterations it takes. So, while the circle contains Z, that's another function that just tests if it's inside. and the number of iterations is less than max iterations. Max iterations is the number of iterations after which we assume it's just going to spiral inward just stay inside.

integer /ˈɪntɪdʒər/ *n.*
整数

notation /noʊˈteɪʃən/ *n.*
记法、符号体系

那我们来看 calculate iterations 这个函数。int calculate iterations。这个写法表示它会返回一个整数，一个数字。它接收一个复数 C。然后 Z.real、Z.imaginary 都从零开始。就从那里开始。然后迭代计数从零开始。所以我们要数一数它需要多少次迭代。所以，当这个圆还包含着 Z 时——那是另一个函数，就是判断它在不在圆里面。并且迭代次数小于最大迭代次数。最大迭代次数就是超过这个次数之后我们就假定它会一直往里螺旋、一直待在里面的那个次数。

68:39

So, while that stuff is true, Z equals Z squared plus C. Z equals Z times Z plus C, which is Z squared. So, we can do that because I overloaded the operators, the multiplication and addition for complex number. So, it behaves correctly. It does this strange multiplication thing. And so, we do that, iterations plus. That means we increment the number of iterations by one. And calculate color just it calculates the color based on number of iterations. So, we have 20 minutes left. I want to blow you away with some music and fractals.

嗯，所以在这些条件都成立时，Z = Z 的平方加 C。Z = Z 乘 Z 加 C，也就是 Z 的平方。我们能这么写，是因为我重载了运算符，重载了复数的乘法和加法。所以它的行为是正确的。它会执行那种奇特的乘法运算。然后我们这么做，iterations++。意思是把迭代次数加一。而 calculate color 就只是根据迭代次数算出颜色。我们还剩 20 分钟。我想用一些音乐和分形把大家震撼一下。

运算符重载（operator overloading）：为自定义的复数类型重新定义 * 和 +，使复数运算代码写起来像普通算式。iterations++ 即计数加一。

overloaded /ˌoʊvərˈloʊdɪd/ *v.*
重载（运算符重载：为自定义类型重新定义 +、* 等运算）

increment /ˈɪŋkrəmɛnt/ *v.*
使递增、加一；n. 增量

blow you away *phr.*
使你大为震撼、令你叹为观止

11 巴赫音乐里的嵌套语法

69:28

So, I'm going to play a piece three pieces of Bach. And yeah, they're just great. So, the first one is actually not written by Bach. Oh, no. What's going on here? it's not written by Bach, but it's the little harmonic labyrinth, which is the song that the dialogue in is based on. Carl Philipp Emanuel Bach. Sort of you know, English language has a grammar. We could nest sentences inside which we can nest more sentences. We could we could nest more sentences. And that itself was a nested sentence.

我要弹一首——三首巴赫的曲子。是的，它们真的非常棒。第一首其实不是巴赫写的。哦不，这是怎么回事？呃，它不是巴赫写的，而是《小和声迷宫》，这首曲子正是那段对话所依据的原型。卡尔·菲利普·埃马努埃尔·巴赫。就好比说，英语是有语法的。我们可以在句子里嵌套句子，在里面还能再嵌套更多句子。我们还能再嵌套更多句子。而这句话本身就是一个嵌套句。

《小和声迷宫》是 GEB 第 5 章前对话的原型乐曲。讲者称其并非 J.S. 巴赫所作并提到 C.P.E. 巴赫（巴赫次子）；该曲（BWV 591）的作者归属确有争议。

labyrinth /ˈlæbərɪnθ/ *n.*
迷宫（比 maze 更文学化，源自希腊神话）

70:10

So, music sort of like English has sort of a grammar to it. These chord progressions and melodies and you know, harmonies particularly have a sort of a language in which they can move between keys and whatnot. And so, that's one way in which Bach embeds recursion inside of music. He has these nested harmonic structures of chord progressions. So, the chord it's yeah, in chapter five of Carl Philipp Emanuel Bach, he talks about it. And the dialogue is modeled after that first song that we heard, little harmonic labyrinth. And melodies also can sort of have recursion in that they can have patterns.

所以音乐有点像英语，英语有它的语法。这些和弦进行、旋律，还有和声，尤其是和声，都有一套自己的语言，让它们能够在不同调性之间转换等等。所以，这就是巴赫在音乐中嵌入递归的一种方式。他用的是这些嵌套的和声结构、和弦进行。所以那个和弦——是的，在《卡尔·菲利普·埃马努埃尔·巴赫》的第五章里，他谈到了这一点。而那段对话正是仿照我们刚才听到的第一首曲子《小和声迷宫》写的。旋律也可以带有某种递归性，因为它们可以有各种模式。

音乐的「语法」：和声进行有转调规则。巴赫在乐曲中嵌套调性层次——离开主调进入属调、再层层返回，结构上与语言的从句嵌套同构。

chord progressions *phr.*
和弦进行（音乐术语：和弦的先后序列）

whatnot /ˈwʌtnɑːt/ *n.*
（and whatnot）诸如此类、等等（口语）

embeds /ɪmˈbɛdz/ *v.*
嵌入、植入（embed A inside B）

70:57

Like this theme for example. This theme like Bach does this a lot. What he does is he takes themes and he restates them with maybe different harmonies. And they embeds them inside smaller and larger scales. So, for example, I don't know if he did this did this like maybe on a larger scale the actual harmony might actually follow this sort of theme. And inside those, you know, like for a long period of time be this key and long period of time be this key. And then when you're in that key he plays this theme.

嗯，比如说这个主题。这个主题——巴赫经常这么干。他会拿一个主题，然后用不同的和声把它重新陈述一遍。然后他把这些主题嵌套在更小和更大的尺度里。比如说，我不知道他有没有在更大的尺度上这么做过，嗯，整体的和声走向说不定真的会遵循这个主题的形状。而在这些内部，比如说很长一段时间处在这个调上，再很长一段时间处在这个调上。然后当你处在那个调里的时候，他就会演奏这个主题。

巴赫的主题手法：同一主题以不同和声、不同时间尺度重现（扩大、缩小），在大尺度的调性走向与小尺度的旋律间形成音乐的自相似。

restates /ˌriːˈsteɪts/ *v.*
重新陈述；（音乐）再现（主题）

71:41

So, that's an example of nesting. So, recursion per se is not there, but the result of a recursion. This is the distinction between recursive processes and recursive structures. Recursive structures are basically any structure that has nesting to it. So, English language is recur– is a recursive structure because it comes from a recursive grammar. It has nesting. And music also it's a recursive structure, not a recursive process. So, recursive process is like almost like a function that defines how you can get the recursive structure.

所以这就是嵌套的一个例子。所以严格来说递归本身并不在那里，在那里的是递归的结果。这就是它们的区别所在递归过程和递归结构之间的区别。递归结构基本上就是任何带有嵌套的结构。所以，英语是递归的——是一种递归结构，因为它来自递归语法。它有嵌套。而音乐也是一种递归结构，不是递归过程。所以递归过程差不多就像一个函数，它定义了你怎么得到这个递归结构。

全讲最重要的概念区分：递归过程（生成规则，如函数）vs 递归结构（带嵌套的产物，如英语句子、巴赫乐曲）。后者是前者留下的痕迹。

per se /ˌpɜːr ˈseɪ/ *adv.*
本身、就其本身而言（拉丁语借词，常用于否定句）

distinction /dɪˈstɪŋkʃən/ *n.*
区别、区分（draw a distinction between 加以区分）

72:15 学生与讲者（对话）

Exactly. That's it. He said so, a recursive process is sort of like a function or something that defines how you end up with a recursive structure. That's exactly it. Yeah. So, so he's just playing like overall there's this whole theme and he chooses some parts of that theme that he also plays. And then he plays like that and then he gets out of that part of it and then he goes to another one which is also part of the Yeah. Yeah. But it's hard to hear cuz it's Yeah, it's hard to hear. It takes a really fine musical ear to hear this.

没错，就是这样。他说，递归过程有点像一个函数或者别的什么东西，它定义了你最终怎么得到一个递归结构。完全正确。对。所以，所以他就是在弹，整体上有这么一个主题，然后他挑出这个主题的某些部分，那些部分他也会弹。然后他那样弹一段，接着从那部分里出来，再进到另一段，而那一段同样是……的一部分。对。对。但是很难听出来，因为——对，很难听出来。要听出这个得有非常敏锐的音乐耳朵。

fine musical ear *phr.*
敏锐的音乐听觉（a fine ear for… 对…的敏锐辨别力）

12 调用栈：压入与弹出

72:49 Curran Kelleher

And so, let's see how much time. 3 minutes. So, We talk about pushing and popping of stacks. So, a function, a computer function say F foo actually. Foo is a function. Or better yet, well, no, forget it. And inside this foo, it calls the function bar. And here's a function bar. It does some stuff and it calls maybe G. It does some other stuff and here's G. G maybe does just one thing and I don't know ends. So, what you have is a stack a call stack. Like it happens in any grammar actually that's sort of recursive flow. So, in computer programs and also in English, you have this notion of a stack. So, here you do some stuff, you do some stuff.

呃，我们看看还有多少时间。3分钟。那么，嗯，我们讲讲栈的压入和弹出。所以，一个函数，一个计算机函数，比如说 F，其实叫 foo 吧。foo 是一个函数。或者更好的是——嗯，不，算了算了。然后在这个 foo 里面，它调用了函数 bar。这里是函数 bar。它做一些事情，然后它可能调用 G。它再做一些别的事情，这里是 G。G 可能只做一件事，然后我不知道，就结束了。所以你得到的是一个栈，一个调用栈。其实在任何语法里都会发生，只要是那种递归式的流程。所以在计算机程序里，还有在英语里，你都有栈这个概念。所以在这里你做一些事，你做一些事。

调用栈（call stack）：函数调用时把返回位置压栈，返回时弹栈取回。GEB 用压栈/弹栈类比人在故事套故事、梦中梦里的层次进出。

call stack *phr.*
调用栈（记录函数调用返回位置的后进先出结构）

74:05 学生与讲者（对话）

And when you call bar your focus actually changes to over here, bar. But you retain sort of in your mind or in the in the memory of the computer where you were, where you left from. So, you yeah. Is this like when you do when you're like trying to do something even like maybe have several songs and so on and so on? Calls? When have goals. Goals. Yeah, exactly. So, that's why computer programming is tough because like you have a high level goal, but then to get to that goal just like you said, you have other goals that you need to meet and each of those goals requires some other goals. So, it's actually recursive. Like It's crazy. So, when you when this func– when this when this program executes, it does this stuff.

然后当你调用 bar 的时候，你的注意力其实转移到了这边，bar 这里。但你在脑子里，或者在计算机的内存里，保留着你刚才在哪儿、你是从哪儿离开的。所以你——对。这是不是就像你在做某件事的时候，比如你可能有好几首歌之类的，一层套一层？调用？当你有目标的时候。目标。对，正是这样。所以，这就是为什么计算机编程很难，因为你有一个高层的目标，但要达到那个目标，就像你说的，你还有别的目标要完成，而那些目标里的每一个又需要一些别的目标。所以它其实是递归的。就像——太疯狂了。所以当这个函数——当这个程序执行的时候，它做这些事。

学生把栈类比到「目标套子目标」，讲者认同：编程之难正在于目标的递归分解——每个目标都需要先完成一串子目标。问题求解本身是递归的。

retain /rɪˈteɪn/ *v.*
保留、保持、记住

74:51 Curran Kelleher

It puts this position A, let's say, on the stack. So, this is the stack. A. And then it goes into here, bar. Blah. Does all this stuff. Then it calls G. But to remember where this is, let's call this position B. It puts B on the stack. And calls G and then does all this stuff. And then after this it returns. And when it returns, it asks, "Where was I last time?" And this is what the stack is for. So, "Where was I last time?" I was at B. So, we pop. Putting something on the stack is pushing. Taking off is called pop.

它把这个位置，就叫 A 吧，放到栈上。所以这就是栈。A。然后它进到这里，bar。啦啦啦。做完所有这些事。然后它调用 G。但为了记住这里是哪儿，我们把这个位置叫 B。它把 B 放到栈上。然后调用 G，接着做完所有这些事。然后在这之后它返回。当它返回的时候，它就问：“我上次在哪儿来着？”这就是栈的用处。所以，“我上次在哪儿来着？”我在 B。所以我们弹出。把东西放到栈上叫压入（push）。取下来叫弹出（pop）。

pop /pɑːp/ *v. / n.*
弹栈、弹出（与 push 压栈相对，栈操作术语）

75:26 学生与讲者（对话）

We pop it and say, "Okay, here's B." This is B is where we were. We go back to there. And we finish this. Once we finish this, we say, "Oh, where were we last time?" So, it gets A. And then go back to A and finish. Yeah. So, so you can like make it so that you have those things, right? Those are like themes maybe like parts of the music, right? Yeah, sure. Yeah, I mean yeah, exactly. other things that thing sort of like reflects itself. Yeah. Yeah. So, the dialogue which reflects little harmonic labyrinth has this sort of structure. It goes inside something inside yet another thing and back out and then back up. So, so it's 5:00. I'm finished.

我们把它弹出来，说：“好，这是 B。”这个 B 就是我们刚才所在的地方。我们回到那儿。然后我们把这部分做完。做完之后，我们说：“哦，我们上次在哪儿来着？”于是它取到 A。然后回到 A，把它做完。对。所以，所以你可以把它做成这样，让你有那些东西，对吧？那些就像是主题，可能是音乐的某些部分，对吧？对，当然。对，我是说，对，正是这样。别的东西，那个东西有点像是在反射自身。对。对。所以那段呼应《小和声迷宫》的对话就有这种结构。它进到某个东西里面，再进到另一个东西里面，然后出来，再回到上一层。所以，5点了。我讲完了。

> 收束呼应：《小和声迷宫》对话的叙事结构本身就是压栈与弹栈——进入故事中的故事再逐层退出。形式与内容的同构是 GEB 的典型手法。

第二部分　核心句型 · 值得模仿的表达

1. What makes X … is the fact that …

“So, what makes this function recursive is the fact that it calls itself.”

用「使 X 具有某性质的正是…」点出本质原因。主语从句 + is the fact that 从句，强调因果关系的核心，适合下定义或点题。

2. if you were to do …, … would …

“If you were to do this rule an infinite number of times … the volume inside of this object would be finite”

were to 虚拟条件句，描述纯理论、不可能实际执行的假设（如无限次迭代）。比 if you do 更突出「思想实验」的语气。

3. the more …, the longer/更 …

“The more that you zoom in on the coast of Britain, the longer the perimeter gets”

「越…越…」双比较级结构，描述两个量的联动关系。科学解释中表达单调关系的首选句式，可仿写 the deeper…, the harder…。

4. Say we … / Let's say …

“So, say we start with this point here, it'll map to this point here.”

口语中引入假设或示例的高频开头，相当于 suppose。讲解算法、举数字例子时非常自然，比 assume 轻松得多。

5. That's what it means to + v.

“That's what it means to have infinite perimeter, which is why it's just so fascinating.”

「这就是…的含义」，用于把抽象概念落到刚讲完的具体图景上，收束解释。后接 which is why 顺势给出结果，衔接流畅。

6. A is pretty much the same thing as B in terms of …

“The Sierpinski triangle is pretty much the same thing as the ferns in terms of mappings”

in terms of 限定比较的维度：「就…而言两者基本是一回事」。做类比、归纳共性时必备，避免绝对化。

7. It's been mathematically proven that …

“It's been mathematically proven or you know, extrapolated that … the volume inside of this object would be finite”

被动完成时引出已确立的结论，隐去证明者、突出结论权威性。学术写作与讲解中陈述定理、共识的标准句式。

8. But keep in mind (that) …

“But keep in mind this is a theoretical creation. It's just in the world of math.”

提醒听者注意前提或限制条件，常用于防止误解（这里区分数学理想与物理现实）。演讲中转折补充的常用信号语。

9. ... correspond to ...

“All the arrows in the diagrams correspond to function calls in the program”

表示两个系统间的一一对应关系，是描述同构、映射时的核心动词，可仿写 each node corresponds to a state。

10. Let's hold off on ... until after ...

“Let's hold off on that answer until after we do the next one.”

「先把…放一放，等…之后再说」。组织讲解节奏、设置悬念的实用表达，hold off on 后接名词或动名词。

第三部分 生词总表 · 复习用

词汇	音标	词性	释义	出现
abstract	/æbˈstrækt/	v.	抽象、提取共性（动词重音在第二音节，区别于形容词）	53:15
applet	/ˈæplət/	n.	（尤指 Java 的）小应用程序	62:54
approximation	/əˌprɑːksəˈmeɪʃən/	n.	近似、近似值	62:10
argument	/ˈɑːrgjəmənt/	n.	（编程）参数、实参；注意不是「争论」义	2:40
at random	—	phr.	随机地、任意地	7:50
averaging	/ˈævərɪdʒɪŋ/	v.	取平均值（average 作动词）	31:39
bagel	/ˈbeɪgəl/	n.	百吉饼（环形硬面包）	10:54
blow you away	—	phr.	使你大为震撼、令你叹为观止	68:39
bottom out	—	phr. v.	触底；（递归）到达终止条件而停止	34:11
branch out	—	phr. v.	分叉、岔开；引申义为拓展新领域	15:02
call stack	—	phr.	调用栈（记录函数调用返回位置的后进先出结构）	72:49
chord progressions	—	phr.	和弦进行（音乐术语：和弦的先后序列）	70:10
coastlines	/ˈkoʊstlaɪnz/	n.	海岸线（coastline）	20:10
complex plane	—	phr.	复平面（横轴为实数、纵轴为虚数的平面）	56:07
coordinate transformation	—	phr.	坐标变换（把一个坐标系中的点映到另一坐标系的函数）	37:26
coordinates	/koʊˈɔːrdənəts/	n.	坐标	31:39
curly braces	—	phr.	花括号 { }（编程术语）	8:32
distinction	/dɪˈstɪŋkʃən/	n.	区别、区分（draw a distinction between 加以区分）	71:41
embeds	/ɪmˈbɛdz/	v.	嵌入、植入（embed A inside B）	70:10
entry point	—	phr.	入口点（程序开始执行的地方）	12:04
equilateral	/ˌiːkwəˈlætərəl/	adj.	等边的（equilateral triangle 等边三角形）	16:50
escape iterations	—	phr.	逃逸迭代次数（曼德博集合渲染术语：轨道逃出圆所需步数）	61:28
essence	/ˈɛsəns/	n.	本质、精髓（communicate the essence of 传达…的精髓）	6:54
exclamation point	—	phr.	感叹号（美式说法；英式为 exclamation mark）	0:40

execution	/ˌɛksəˈkjuːʃən/	n.	（程序、规则的）执行	17:35
extrapolated	/ɪkˈstræpəleɪtɪd/	v.	外推、推断（extrapolate：由已知趋势推出未知）	21:04
factorial	/fækˈtɔːriəl/	n.	阶乘（n! = n×(n−1)×…×1）	0:40
fill in	—	phr. v.	代课；临时顶替（fill in for sb.）	0:00
fine musical ear	—	phr.	敏锐的音乐听觉（a fine ear for… 对…的敏锐辨别力）	72:15
finite	/ˈfaɪnaɪt/	adj.	有限的（反义词 infinite 无限的）	21:04
generalize	/ˈdʒɛnrəlaɪz/	v.	推广、一般化（把特例扩展为一般规则）	19:25
genie	/ˈdʒiːni/	n.	（阿拉伯神话中的）精灵，如神灯精灵	36:08
getting at	—	phr. v.	（be getting at）想说的是、意指	29:26
handout	/ˈhændaʊt/	n.	（课堂发的）讲义、资料	1:24
hold off on	—	phr. v.	暂缓、推迟（做某事）	32:16
if you will	—	phr.	姑且这么说吧（缓和措辞的口头插入语）	5:53
imaginary numbers	—	phr.	虚数（含 i = √−1 的数）	56:07
implemented	/ˈɪmpləmɛntɪd/	v.	实现、落实（implement：把设计变成可运行的代码）	62:54
increment	/ˈɪŋkrəmɛnt/	v.	使递增、加一；n. 增量	68:39
indices	/ˈɪndɪsiːz/	n.	index 的复数：索引、下标	5:53
integer	/ˈɪntɪdʒər/	n.	整数	67:56
iterations	/ˌɪtəˈreɪʃənz/	n.	迭代（次数）；iteration 一次重复执行	21:40
labyrinth	/ˈlæbərɪnθ/	n.	迷宫（比 maze 更文学化，源自希腊神话）	69:28
linear	/ˈlɪniər/	adj.	线性的；not linear 指变化不成简单比例、难预测	58:16
loop back on themselves	—	phr.	绕回自身、回到自身（描述递归结构的常用表达）	9:29
manifestation	/ˌmænəfəˈsteɪʃən/	n.	体现、表现形式	36:08
mapping	/ˈmæpɪŋ/	n.	映射（数学：点到点的对应规则）	41:55
mathy	/ˈmæθi/	adj.	（口语）数学味重的、偏数学的	56:07
mind-boggling	/ˈmaɪndˌbɑːglɪŋ/	adj.	令人难以置信的、烧脑的	21:40
nest	/nɛst/	v.	嵌套（一层套一层地放入）；名词 nesting 嵌套结构	10:10
not necessarily	—	phr.	不一定、未必（部分否定）	33:06

notation	/noʊˈteɪʃən/	n.	记法、符号体系	67:56
notion	/ˈnoʊʃən/	n.	概念、观念（比 idea 更正式）	6:54
nub	/nʌb/	n.	小凸起、疙瘩；(the nub of) 要点	64:22
origin	/ˈɔːrədʒɪn/	n.	（坐标系的）原点；起源	60:39
overloaded	/ˌoʊvərˈloʊdɪd/	v.	重载（运算符重载：为自定义类型重新定义 +、* 等运算）	68:39
per se	/ˌpɜːr ˈseɪ/	adv.	本身、就其本身而言（拉丁语借词，常用于否定句）	71:41
perceive	/pərˈsiːv/	v.	感知、察觉、领悟	53:15
perimeter	/pəˈrɪmɪtər/	n.	周长；（营地等的）外围	21:04
pixel	/ˈpɪksəl/	n.	像素	62:10
pop	/pɑːp/	v. / n.	弹栈、弹出（与 push 压栈相对，栈操作术语）	74:51
preposition	/ˌprɛpəˈzɪʃən/	n.	介词（语法术语）	9:29
probabilities	/ˌprɑːbəˈbɪlətiz/	n.	概率（probability）	48:25
pseudo code	/ˈsuːdoʊ koʊd/	n.	伪代码（只表达思路、不能直接运行的代码）；pseudo– 假的	11:41
radius	/ˈreɪdiəs/	n.	半径	61:28
randomization	/ˌrændəmɪˈzeɪʃən/	n.	随机化（人为引入随机扰动）	20:10
randomness	/ˈrændəmnəs/	n.	随机性	19:25
recursive	/rɪˈkɜːrsɪv/	adj.	递归的；本讲核心术语，指自我调用或自我指涉的	0:00
refine	/rɪˈfaɪn/	v.	使更精确、细化、改进	24:39
reframed	/ˌriːˈfreɪmd/	v.	重新框定、重新定义（reframe：换一种框架看待）	65:31
relative pronoun	—	phr.	关系代词（如 which、that，引导定语从句）	9:29
render	/ˈrɛndər/	v.	渲染、绘制（图形学术语）；亦有「使成为」义	62:10
repetitive	/rɪˈpɛtətɪv/	adj.	重复性的（与 recursive 递归的相对，本讲关键对比）	50:46
restates	/ˌriːˈsteɪts/	v.	重新陈述；（音乐）再现（主题）	70:57
retain	/rɪˈteɪn/	v.	保留、保持、记住	74:05
satellite image	—	phr.	卫星图像	24:06
scale	/skeɪl/	v.	按比例缩放；n. 尺度	14:09
segments	/ˈsɛgmənts/	n.	线段；段、部分（segment）	16:50
self-similar	/ˌsɛlfˈsɪmələr/	adj.	自相似的（局部与整体形状相同），分形的定义性特征	0:40

sparse	/spɑːrs/	adj.	稀疏的、零落的	48:25
spiral inward	—	phr.	向内盘旋、螺旋式收敛	60:39
spiraling	/ˈspaɪrəlɪŋ/	v.	盘旋（上升）；spiral 螺旋（运动）	49:06
square root	—	phr.	平方根	56:55
stepping out of the system	—	phr.	跳出系统（GEB 核心概念：从更高层次审视系统）	52:38
transition networks	—	phr.	转移网络（RTN，用有向图表示语法的形式化工具）	6:54
trunk	/trʌŋk/	n.	树干	12:04
well put	—	phr.	表述得好（That was well put. 这话说得好）	36:08
whatnot	/ˈwʌtnɑːt/	n.	（and whatnot）诸如此类、等等（口语）	70:10
zoom in on	—	phr. v.	放大观察；聚焦于	0:40

第四部分　理解自测 · 8 题

1. 递归函数为什么必须「触底」（bottom out）？阶乘函数靠什么终止？
2. 科赫雪花的著名悖论是什么？为什么会这样？
3. 为什么没有人能确定英国海岸线的长度？
4. 混沌游戏每步随机选顶点，为什么最终图形却总是同一个谢尔宾斯基三角形？
5. 谢尔宾斯基三角形和蕨类的程序代码里没有自调用函数，讲者认为递归体现在哪里？
6. 曼德博集合图像中黑色区域和彩色区域各代表什么？
7. 讲者如何区分「递归过程」与「递归结构」？各举了什么例子？
8. 蕨类 IFS 的四个映射为何用 .01/.07/.07/.85 的不等概率？

参考答案

1. 否则函数会无限调用自身，永远无法返回结果。阶乘靠条件「n 大于 1 才递归、否则返回 1」终止（第 3–4、50–51 段）。
2. 面积有限而周长无限。每次对边应用「三等分加凸起」规则都会让总长度变长，无限次迭代后周长发散，而图形始终被包在有限区域内（第 31–32 段）。
3. 海岸线近似分形：测量尺度越细，画出的折线越贴合细节，总长度就越长且不收敛，因此长度依赖于测量尺度（第 36–38 段）。
4. 因为三个「走一半」操作构成的映射把整体递归地映回自身，随机迭代的落点必然收敛并填满同一个吸引子结构，与具体随机序列无关（第 44–45、75 段）。
5. 递归在映射层面：各坐标变换把图形不断映回自身内部，代码只是重复执行这些映射。递归属于数学结构，不属于代码写法（第 71、76 段）。
6. 黑色点是迭代 $Z=Z^2+C$ 后始终不逃出半径 2 圆的点，才是真正的曼德博集合；彩色只表示各点逃逸所需的迭代次数，是为了好看（第 90–91、97 段）。
7. 递归过程是生成规则（如递归函数、递归语法），递归结构是带嵌套的产物。英语句子和巴赫音乐是递归结构，由递归过程生成（第 104 段）。
8. 概率决定各区域落点密度。若概率均等，「向上盘旋」映射难以连续发生，点大多堆在底部，图形稀疏难看；85% 的权重让主茎方向被充分填充（第 73–74 段）。