复杂问题和（有限的）马虎

**C H A P T E R 11**

■ ■ ■

**Hard Problems and (Limited)**

**Sloppiness 复杂**

*The best is the enemy of the good.* 要求过高，反难成功。

—Voltaire

This book is clearly about algorithmic problem solving. Until now, the focus has been on basic principles for algorithm design, as well as examples of important algorithms in many problem domains. Now, I’ll give you a peek at the flip side of algorithmics: hardness. Although it is certainly possible to find efficient algorithms for many important and interesting problems, the sad truth is that most problems are really hard. In fact, most are so hard that there’s little point in even trying to solve them. It then becomes important to recognize hardness, to show that a problem is intractable (or at least very likely so), and to know what alternatives there are to simply throwing your hands up.

这本书显然是关于算法解决问题。到目前为止，重点一直在算法设计的基本原则，以及在许多问题领域重要的算法的例子。现在，我让你窥探一下算法的另一面：困难度。虽然对于很多重要而又有趣的问题当然是有可能找到有效的算法，可悲的事实是，大多数问题是真的很难。事实上，大多数问题都很难，以至于试图解决他们都是没有意义的。所以重要的是要认识到他们的困难度，表明一个问题是棘手的（或至少是非常有可能的是棘手的），并知道什么替代品有简单高举双手，顺利解决问题。

There are three parts to this chapter. First, I’m going to explain the underlying ideas of one of the greatest unanswered questions in the world—and how it applies to you. Second, I’m going to build on these ideas and show you a bunch of monstrously difficult problems that you may very well encounter in one form or another. Finally, I’ll show you how following the wisdom of Voltaire, and relaxing your requirements a bit, can get you closer to your goals than might seem possible, given the rather depressing news in the first two parts of the chapter.

本章有三个部分。首先，我要解释世界最大的悬而未决的问题之一的基本思想 ，以及这些思想如何适用于你。其次，建立在这些思想，我会告诉你一堆骇人听闻困难的问题，你很可能以某种形式或另一种形式遇到。最后，我将告诉你如何遵循伏尔泰的智慧，松弛一些你的需求，使得你对于本章的前两部分提及的那些困难的问题，能够有可能更接近你的目标。

As you read the following, you may wonder where all the code has gone. Just to be clear, most of the chapter is about the kind of problems that are simply *too hard*. It is also about how you uncover that hardness for a given problem. This is important, because it explores the outer boundaries of what our programs can realistically *do*, but it doesn’t really lead to any programming. Only in the last third of the chapter will I focus on (and give some code for) approximations and heuristics. These approaches will allow you to find usable solutions to problems that are too hard to solve optimally, efficiently, and in all generality. They achieve this by exploiting a loophole—the fact that in real life we may be content with a solution that is “good enough” along some or all of these three axes.

当你阅读以下内容，你可能会有疑问代码去哪儿了。我们需要清楚，本章节大部分内容是关于很困难的问题。这也关于到对你怎么去界定一个给定的问题的困难度。这一点很重要，因为它探讨的是什么我们的程序到底可以实现些什么的边界，但它并没有真正需要任何编程。只有在本章的最后三部分，我将专注于（并给出一些代码）近似和启发式算法。这些方法可以让你普遍上对于那些太难以找到有效的最优解的问题，找到可用的解决方案。他们通过利用一个漏洞解决问题，事实上，在现实生活中，我们可能会满足于一个“足够好”解决方案，沿着这三个轴的部分或者全部。

* **11-1Reduction Redux 重提规约**

From Chapter 4, I’ve been discussing reductions every now and then. Mostly, I’ve been talking about reducing to a problem you know how to solve—either smaller instances of the problem you’re working on or a different problem entirely. That way, you’ve got a solution to this new, unknown problem as well, in effect proving that it’s easy (or, at least, that you can solve it). Near the end of Chapter 4, though, I introduced a different idea: reducing in the other direction to prove *hardness*. In Chapter 6, I used this idea to give a lower bound on the worst-case running time of any algorithm solving the convex hull problem. Now we’ve finally arrived at the point where this technique is completely at home. Defining complexity classes (and problem hardness) is, in fact, what reductions are normally used for in most textbooks. Before getting into that, though, I’d like to really hammer home how this kind of hardness proof works, at the fundamental level. The concept is pretty simple (although the proofs themselves certainly need not be), but for some reason, many people (myself included) keep getting it backward. Maybe—just maybe—the following little story can help you when you try to remember how it works.

从第4章起，我就一直在讨论规约。大多数情况下，我一直在讨论将未知问题规约到你已知如何解决的一个问题上，无论是你正在处理的问题的一小部分或者一个完全不同的难题。这样一来，对于这个新的，未知的问题，你也有了解决方法。实际上证明它很容易（或者，至少，你可以解决它）。在第4章的结尾附近，虽然我介绍了一个不同的想法：通过规约的方式来证明困难度。在第六章中，我用这个想法，证明了任何算法求解凸包问题的最坏情况运行时间的下界。现在，我们终于要具体讨论这种技术了。其实，规约一般在大部分教科书都是用来定义复杂类（和问题的复杂度）。在具体讨论之前，我想强调，作为基础如何证明困难度。这个概念很简单（虽然证明本身的确不需要），但由于某些原因，很多人（包括我自己）保持向后得到它。也许，只是也许，下面的小故事可以帮助你，当你试着去记住它是如何工作的。

Let’s say you’ve come to a small town where one of the main attractions is a pair of twin mountain peaks. The locals have affectionately called the two Castor and Pollux, after the twin brothers from Greek and Roman mythology. It is rumored that there’s a long-forgotten gold mine on the top of Pollux, but many an adventurer has been lost to the treacherous mountain. In fact, so many unsuccessful attempts have been made to reach the gold mine that the locals have come to believe it can’t be done. You decide to go for a walk and take a look for yourself.

比方说，你已经到了一个小县城里，那里的主要景点之一，是一对孪生的山峰。当地人都亲切地称这两座山峰Castor和Pollux，取名来从希腊和罗马神话中的孪生兄弟。有传言说，有一个长期被遗忘的金矿在Pollux的上方，但许多冒险家已经输给了奸诈的高山。其实，这么多找金矿的不成功的尝试，已经让当地人都不相信有人可以找到金矿了。你决定去亲自去看看。

After stocking up on donuts and coffee at a local roadhouse, you set off. After a relatively short walk, you get to a vantage point where you can see the mountains relatively clearly. From where you’re standing, you can see that Pollux looks like a really hellish climb—steep faces, deep ravines, and thorny brush all around it. Castor, on the other hand, looks like a climber’s dream. The sides slope gently, and it seems there are lots of handholds all the way to the top. You can’t be sure, but it seems like it might be a nice climb. Too bad the gold mine isn’t up there.

当你在当地的客栈吃饱甜甜圈和咖啡后，你就出发了。经过一个相对短的步行路程，你到达一个有利位置，你可以看到比较清楚看到山。从你站的地方，可以看到，Pollux看起来像一个真正地狱般的陡峭盘旋上升，深深的沟壑，周围荆棘丛生。在另一方面，Castor看起来像一个登山者的梦想。两侧坡度平缓，似乎有很多把手一路向顶。你不能确定，但是看起来它可能会是一个很好的攀登。太糟糕了金矿的是，金矿不在那里。

You decide to take a closer look, and pull out your binoculars. That’s when you spot something odd.

There seems to be a small tower on top of Castor, with a zip line down to the peak of Pollux.

Immediately, you give up any plans you had to climb Castor. Why? (If you don’t immediately see it, it might be worth pondering for a bit.)[[1]](#footnote-1)

你决定仔细看看，并拿出你的望远镜。你发现了一些奇怪的事情。 在Castor,的顶部似乎有一个小塔，有一条飞索连接到Pollux的山峰。你马上放弃了你必须爬上Castor任何计划。为什么呢？ （如果您没有立即看到它，也许还值得琢磨了一下。）

Of course, we’ve seen the exact situation before, in the discussions of hardness in Chapters 4 and 6. The zip line makes it easy to get from Castor to Pollux, so if Castor were easy, someone would have found the gold mine already.[[2]](#footnote-2) It’s a simple contrapositive: if Castor were easy, Pollux would be too; Pollux is *not* easy, so Castor can’t be either. This is exactly what we do when we want to prove that a problem (Castor) is hard. We take something we *know* is hard (Pollux) and show that it’s easy to solve this hard problem using our new, unknown one (we uncover a zip line from Castor to Pollux).

当然，我们这样遇到过同样的状况，当我们在第4章和第6章讨论问题困难度。飞索可以很容易从Castor去Pollux，因此，如果Castor是容易的，那么肯定已经会有人发现金矿了。这是一个简单的逆反命题：如果Castor很容易，那么Pollux也是容易的，如果Pollux不容易，那么Castor也是不容易的。这正是当我们想证明一个问题（Castor）是很难的时候所做的。我们把我们知道很难的东西（Pollux），并表明很容易使用新的，未知的（我们发现的从Castor到Pollux有一条飞索）来解决这个难题。

As I’ve mentioned before, this isn’t so confusing in itself. It can be easy to confuse things when we start talking about it in terms of reductions, though. For example, is it obvious to you that we’re reducing Pollux to Castor here? The reduction is the zip line, which lets us use a solution to Castor as if it were a solution to Pollux. In other words, if you want to prove that problem X is hard, find some hard problem Y and reduce it to X.

正如我之前提到的，这本身并不混乱。但当我们开始用归约的方式讨论他的时候，它很容易混淆，。举例来说，很明显，在这里我们将Pollux归约到 Castor？归约的方法就是飞索，它让我们可以找到方法到达Castor就像他是到达Pollux的方法一样。换句话说，如果你想证明问题X是很难，那么找到一个难题Y，并和将其归约到X。

注意：飞索是朝着规约的相反方向。这是很重要，你没有混淆，或者整体思路错误。术语规约在这里的意思基本上是“哦，那很容易，你只需要……”换句话说，如果你将A规约到B，你就在说：“你想解决A？这很简单，你只要解决B.“或者在这种情况下，说：”你要去Pollux？这很简单，只要去Castor（并使用飞索）。“换句话说，我们将去Pollux规约到去Castor（而不是相反的一条路）。

很多事情事情在这里值得注意。首先，假设该飞索很容易使用。如果它不是一个飞索，而是一条水平线，你必须保持平衡跨越？这将是非常辛苦，因此这不会给我们的任何信息。对于我们所知，人们可能很容易到达Castor的山顶;他们可能无论如何都无法达到Pollux的金矿，所以我们知道了什么？另一方面则是从另一个方向规约也将不会给我们任何帮助。一条从Pollux到Castor的飞索不会影响我们对Castor的任何估计。所以，如果你可以从Castor到Pollux？你无论如何都无法到达Pollux的山顶！

我们看图11-1。图中节点表示的问题，线代表容易的规约（即，它们并不重要，渐近）。底部的粗线代表“地”，在某种意义上说，没有解决的问题是“在空中”，而解决这些问题，相当于将他们规约到没有，或者将他们接近“地面”。第一图像示出了一个未知的问题u被规约到一个已知的容易的问题e的情况。 e是容易问题的事实，通过有一个简单的规约从e到地面表明。 u连接到e，因此，让我们有了从u到地面的解决方案。

现在看一下第二个图。这里，公知的，困难问题规约到未知的问题u。我们是否有从u到地面的边（比如图中的虚线边）？这将使我们有条从h到地面的道路，但这样的路径不存在，否则h不会是困难的！

下面，我将使用这个基本理念，不仅表明问题是很难，同时定义一些困难程度的标记。正如你可能（也可能不会）注意到，这里困难这个词有些含糊不清。它基本上可能有两种不同的含义：

\*问题是棘手的，任何算法求解它必须是指数级的。

\*我们不知道这个问题是否是棘手的，但从来没有人能够找到一个多项式算法去解决他。

第一点表示这些问题是难通过一台计算机来解决，而第二点，它对于人（或许计算机同样）很难解决。再看一下图11-1中右边的图像。“困难”的两个定义在这里是什么含义？让我们看第一种情况：我们知道h是棘手。不可能有效地解决它。解决u的方法（即，规约到地面）将意味着h的解决方法，所以没有这样的解决方法存在。因此，u也必须是棘手的。

第二种情况是有点不同，这里困难涉及到知识的缺乏。我们不知道问题h是否棘手，尽管我们知道，似乎很难找到一个解决方案。其核心观点仍然是，如果我们将h规约到U，则U至少是和h一样困难的。如果h是棘手的，那么u也是。另外，事实上，很多人都试图找到h的解决方法使得它看起来不太可能成功解决，这也意味着u不可能是容易的。越多的努力去解决h，更惊人它会如果u是容易的（因为这样h也会是）。这是，事实上，正是对实践的重要问题整体转换的情况：我们不知道他们是否棘手，但大多数人还是非常相信他们是。让我们来仔细看看这些淘气的问题。

通过子问题规约

通过规约表现困难可能有点抽象和奇怪，有一个特殊的情况（或者，在某些方面，是一个1不同的角度）可以很容易理解：如果你的问题有一个困难的子问题，那么问题作为一个整体（显然）是困难的。换句话说，如果解决您的问题意味着你还必须解决众所周知很难的一个问题，你基本上是倒霉了。例如，如果你的老板要求你创建一个反重力悬浮滑板，你很可能做了很多工作，如手工艺制作滑板或画上一个漂亮的图案。然而，实际上解决规避重力的问题使得一切努力都从头开始。

那么，这是怎么是规约？这是一个规约，因为你仍然可以用你的问题来解决的困难的子问题。换句话说，如果你能建立一个反重力悬浮滑板，那么你的解决方案可以（再次，很明显）用来规避重力。困难的问题甚至没有真正转化，最多规约了;它只是嵌入（有些无关）的上下文中。或考虑对于一般排序的运行时间最坏情况的对数线性下界。如果你写一个程序，操作了一系列对象，对他们进行一些操作，然后输出有序的输出这些对象的信息，在最坏的情况下，你可能无法做的比对数线性更好。

但是，为什么是“可能”？因为这取决于是否有一个真正的规约在那里。你的程序是否是令人信服地作为“排序机”？有没有可能对我来说，如果我可以使用你的程序，给它的对象，可以对任何实数进行排序？如果是的话，那么界限将保持。如果没有，那么也许它没有。例如，可能排序是基于整数，那么可以用计数排序？或者，也许你实际上创建了自己的排序键，以便对象可以在你按照你的要求以任何顺序输出？问题是你否是能够将问题表述的充分，是否可以表达一般的排序问题。这是，事实上是这一章的重要见解之一：问题的困难度关乎是否能够将他描述清楚的困难度。

* [**11-2 Not in Kansas Anymore？ 再也不在堪萨斯州了？**](https://github.com/owlman/PythonAlgorithms/wiki/Not%20in%20Kansas%20Anymore)

当我在写这个的时候，一篇科学论文在网上公布后，互联网上激烈的开始讨论，声称证明了已经解决了这个所谓的P与NP问题，得出的结论是P不等于NP。虽然正在形成的共识是，这个证明是有缺陷的，这篇论文引起了极大的兴趣，至少在计算机科学界。此外，更不可信的有相似内容的论文（或相反，即P等于NP）会定期的出现。从20世纪70年代开始，计算机科学家和数学家一直在设法解决这个问题，甚至解决这个问题还有还有一百万美元奖金。虽然在了解这个问题有了很大的进展，似乎没有真正的解决办法即将到来。这是为什么这么难？为什么它如此重要？究竟什么是P与NP？

问题是，我们真的不知道我们正生活的世界是怎么样的。使用绿野仙踪作为比喻，我们可能会认为我们生活在堪萨斯州，但如果有人证明了P =NP，我们就绝对不再在堪萨斯了。相反，我们会在某种仙境看Oz，一个Russel Impagliazzo已命名为Algorithmica的世界。Algorithmica有什么重要的，你说呢？在Algorithmica，引用一个著名的歌曲，“你永远不会更换你的袜子，酒精在岩石间涓涓流下来。”更重要的是，生活会少了很多问题。如果你能说出一个数学问题，你也可以自动解决这个问题。事实上，程序员不再需要告诉计算机做什么，只需要清晰描述所需的输出。几乎任何类型的优化将是微不足道的。在另一方面，加密将是非常困难的，因为破解密码会非常，非常简单。

P与NP看似非常不同的东西，尽管他们都是两类问题。实际上，它们是两类决策问题，问题可以通过是否来回答。这可能是这样一个问题，如“有从s到t的权重最多是w路径的妈？”或者“是否有放将东西放到背包的方式，使我至少有v值？”第一类P被定义为包括这些≈我们可以在多项式时间内解决（在最坏的情况下）的问题。换句话说，如果你能差不多把我们看到的问题转化成一个决策问题，其结果必然是属于P的.

NP似乎有一个更不严格定义：它包括能够在多项式时间内由一个“神奇的电脑”被称为非确定性图灵机，或NTM解决任何决策问题。这就是NP中的N的来源-NP表示“不确定地多项式。”据我们所知，这些不确定性的机器超级强大。基本上，在任何他们需要做出选择的时候，他们可以猜测，通过魔法，他们会永远猜对。听起来很真棒，对不对？

举个例子，考虑在图中找从s到t的最短路径。你已经知道了很多关于如何使用非魔法的算法做到这一点。但是，如果你有一台NTM？你只要从S开始，看邻居。你要走哪条路？谁知道，那就采取一种猜测。机器，因为你使用，你永远是对的，所以你会神奇地走没有弯路的最短路径。对于像有向无环图中找最短路径这样的问题，这可能似乎不是一个巨大的胜利。当然是一个可爱的花招，但运行时间仍然是线性的。

不过，考虑第1章第一个问题：参观瑞典的所有城镇只有一次，尽可能高效。还记得我怎么说，用最先进的技术这花了大约85 CPU-年来解决这个问题在几年前？如果你有一个NTM，你只需要每个城镇计算一次。即使你的机器是机械用手摇，它应该在几秒钟内完成计算。这似乎很厉害吧？而且神奇？

描述的NP（或者，非确定性计算机）的另一种方法是看解决问题和检验解决方法之间的差异。我们已经知道解决问题方法的意思。如果我们要检验解决一个决策问题，我们比“是”或“否”需要的更多 - 我们还需要某种证明或者证书（与此证书必须是多项式的大小）。例如，如果我们想知道是否有从S到T的路径，证书可能是实际路径。换句话说，如果你解决了这个问题，发现答案是“是”，你可以使用该证书来说服我，这是真的。换种方式说，如果你成功地证明了一些数学表达式，你的证明就是是这个证书。

对于一个属于NP的问题，对于任何“是”的回答，我都必须能够在在多项式时间内检查该证书。非确定性图灵机可以解决此类问题，通过简单地猜测证书。神奇吧？

好吧，也许……你看就是这样。我们知道，P是不是神奇的，它包含了我们非常清楚地知道如何解决的问题。 NP看起来像一个巨大的类的问题，任何能够解决所有这些问题的机器超越了这个世界。事情是，在Algorithmica，有个叫NTM的冬天洗。或者说，我们很普通，单调的电脑（确定性图灵机）会变成得一样强大。他们始终拥有魔法！如果P = NP，我们可以解决任何（决策）的问题，通过实际的（可验证）的解决方案。

* [**11-3 Meanwhile, Back in Kansas … 与此同时，回到堪萨斯州**](https://github.com/owlman/PythonAlgorithms/wiki/Meanwhile,%20Back%20in%20Kansas%20%E2%80%A6)

好吧，Algorithmica是一个神奇的世界，如果我们原来是住在这将是非常棒的，但事实是，我们不是。在所有的可能性，有找到一个证明和检验它之间有很大的区别，解决一个问题和每一次猜测正确的解决方案之间也有很大的区别。所以，如果我们仍然在堪萨斯州，我们为什么要关心这一切？

因为它给我们关于困难非常有用的概念。你看，我们有一大堆的卑鄙的小动物形成一类名为NPC。这代表“NP完全问题，”这些是所有NP问题中最困难的问题。更精确地说，在NPC中每个问题是至少和在NP每个问题一样困难。我们不知道他们是否棘手，但如果你是要解决这些和钉子一样坚韧问题中的任何一个，你就自动地将我们所有人带到Algorithmica了！虽然世界人口可能会因为不必再更换袜子而欢喜，这不是一个很可能的情况（我希望在上一节强调过）。这将是完全惊人的，但似乎完全不可行的。

这不仅会是令人震惊的怪异…考虑到巨大的上升空间，以及打败这些小动物中其中任何一个所付出的巨大的努力，四十年的失败（到目前为止）似乎增强了我们的信心，下注你不会成为一个成功的人。至少不会很快。换言之，对NP完全问题可能是棘手的（对于计算机很困难），但迄今他们对于人类肯定是困难的。

但这一切是如何工作的？为什么杀死一个NPC怪物就能把把所有NP的轰然倒塌到P并把我们带到Algorithmica？让我们回到我们规约图。看看图11-2。假设，现在，所有的节点代表了NP问题（即在此刻，我们将NP看为“问题的整个世界”）。左边的图像展示了完全的概念。这类问题中，一个问题c是完全的，如果在类中的所有问题都可以“轻松地”规约到c。在这种情况下，我们谈论的类是NP，并且规约是“容易的”，如果他们多项式。换句话说，存在的问题c是NP完全的如果（1）C本身是在NP，和（2）中的NP的每一个问题可以在多项式时间规约到c。

事实上，每一个问题（在NP），可以规约到这些困难的问题，意味着他们是硬核，如果你能解决这些问题，那么你可以解决任何NP问题（突然，我们就不在堪萨斯了）。图片应该有助于明确这一点：解决c表示从C到地面添加一条实线箭头（规约到没有），这立即让NP中所有问题通过c都有了一条到地面的实线箭头。

现在，我们已经使用规约去定义NP中最棘手的问题，但我们可以稍微拓展这个想法。如11-2右图，说明了我们如何利用传递地规约，对于困难度的证明，如那些我们一直在讨论之前（比如图11-1的右边的图）。我们知道，c是困难的，所以它规约到u，证明u是困难的。我们已经知道这是如何工作的，但该图稍微更多的给出技术理由说明为什么它是在这种情况下，这是对的。通过将c规约到U，我们现在已经将U放在了原来和c一样的位置。我们已经知道，在NP中的每一个问题可以规约到c（这意味着它是NP-完全）。现在我们也知道，每一个问题可通过C规约到u。换句话说，U也满足NP-完全的定义，和，如图所示，如果我们能够在多项式时间内解决它，我们将建立的P = NP。

现在，到目前为止我只谈到决策问题。主要的原因是，它使得在正式推理很多东西（其中大部分我在这里将不包括）更容易一点。即便如此，这些想法对于其他各种问题都是相关的，比如许多优化问题，我们已经在这本书中提到（将在在本章的后面部分）。

Figure 11-2

考虑，例如，发现瑞典的最短游的问题。因为它不是一个决策问题，它不是在NP。即便如此，这是一个非常棘手的问题（意味着“对于人类很难解决”和“最有可能棘手的”），就好像在NP中的任何东西，它会突然很容易，如果我们发现自己在Algorithmica。让我们分别考虑这两点。

术语完全保留了一类问题中最困难的问题，所以NP完全问题是NP中的恶霸。虽然对于可能属于类外的其他问题，我们也可以使用相同的困难指标。也就是说，任何问题至少与（是否能做多项式时间规约决定）中的NP任何问题一样困难，但它本身不一定是NP。这样的问题被称为NP-难。这意味着，NPC问题的另一个定义，NP-完全问题，是它包含的NP中所有的NP-困难问题。在图中找到最短线路（如通过瑞典镇）是一个NP-难问题称为旅行商（或销售代表）问题，通常被称为是TSP。我之后会回到这个问题。

关于另外一点：如果P = NP，为什么会优化等问题会很容易？对于如何用证书查找实际的路线有一些技术手段，但我们只专注于NP问题的是非的本质，和我们正在寻找在TSP问题的数值长度之间的差异。为了简单起见，我们假设所有的边权重均为整数。同时，由于P = NP，就可以在多项式时间解决决策问题的是非（参见附文“不对称，CO-NP，和Algorithmica的奇迹”）。一种方式是将决策问题作为一个黑盒子，执行二分查找去找到最佳答案。

例如，我们可以累加所有的边权重，我们得到一个对TSP的旅游成本的上限C，以0作为下限。然后，我们初步猜测的最小值是C/ 2和解决决策问题，“有长度至多C / 2的线路吗？”我们在多项式时间内得到是或非，然后可以保持平分上界或者下界的范围。练习11-1要求你证明，所产生的算法是多项式的。

提示 用一个黑盒子来二等分可以用在其他情况下，即使在复杂类之外。如果你有一个算法，让你确定一个参数是否足够大，你可以平分找到合适/最佳值，在对数因子的成本下。真的挺便宜的。

换句话说，即使多复杂性理论集中于决策问题，优化问题并不是完全不同。在很多情况下，你可能会听到人们用这个词NP-完全时，其实他们真正的意思是NP-难。当然，你应该小心把事情做对，但你是否呈现的问题是NP-难或者NP-完全对于讨论他的困难程度并不是至关重要的。 （只要确保你的规约是正确的方向！）

不对称，CO-NP，和Algorithmica的奇迹

类NP的定义是不对称的。它包括所有的决策问题的是的情况可以通过NTM在多项式时间解决。但是请注意，我们没有说任何关于否的情况。因此，举例来说，这是相当清楚的，是否有一个参观瑞典每个镇经过一次的旅行，一个NTM将在合理时间内回答“是”。如果答案是“不”，那么，它可能需要很多时间。

这种不对称背后的直觉是非常接近的。我们的想法是，为了回答“是”,NTM只需要（用“魔法”）找到一组选择导致计算的答案。为了回答“不”，但是，它需要确定没有这样的计算存在。虽然这似乎很不一样，我们真的不知道，如果是这样。你看，我们这里遇到了很多“与问题”的复杂性理论问题之一：NP与CO-NP。

co-NP是NP问题的补充。对于每一个“是”的回答，我们现在想“否”，反之亦然。如果NP是真正不对称的，那么这两个类是不同的，尽管它们之间存在重叠。例如，所有的P在于他们的交集，因为无论是在P的是和否的情况都可以在多项式时间通过NTM来解决（由确定性图灵机，对于这个问题）。

现在考虑如果一个NP-完全问题FOO发现是在NP和反NP的交集会发生什么。首先，在NP所有问题都可以规约到NPC，所以这将意味着所有NP会在CO-NP里面（因为我们现在可以通过FOO处理它们的补）。还有仍在NP问题之外的在CO-NP中的问题吗？考虑这样一个假设的问题，BAR。它补，co-BAR，将是NP，对不对？但由于NP是在co-NP内，co-BAR也将是在co-NP内。这意味着，它的补，BAR，将在NP内。但是，但是，但是...我们认为它是NP外，-矛盾！

换句话说，如果我们发现一个在NP和co-NP的交集中的一个NP-完全问题，那么我们将证明，NP= CO-NP，那么不对称已经消失了。如前所述，P均是在此交集中，因此，如果P = NP，我们也将有NP= CO-NP。这意味着，在Algorithmica，NP是惊喜对称的。

注意，这个结论经常被用来说明在NP和co-NP交集中的问题很可能不是NP-完全的，因为我们（强烈）相信NP和反NP是不同的。例如，没有人找到一个多项式时间内分解数字的方法，而这构成了密码学的基础。然而，问题是，这个问题同时在NP和co-NP中，所以大多数计算机科学家认为，这不是NP-完全问题

* [**11-4 But Where Do You Start? And Where Do You Go from There? 但是，从哪里开始？以及从那里到哪去？**](https://github.com/owlman/PythonAlgorithms/wiki/But%20Where%20Do%20You%20Start?%20And%20Where%20Do%20You%20Go%20from%20There?)

我希望现在基本思路是相当清晰的：NP由所有可以在多项式时间验证的回答“是”的决策问题组成。NPC包括NP中最难的问题;NP所有问题可在多项式时间规约到NPC问题。 P是一系列在NP中的问题，我们可以在多项式时间内解决。因为不同分类的定义，如果有P与NPC之间有一点重叠，我们将有P = NP= NPC。我们还建立了，如果我们在多项式时间从一个NP-完全问题规约到NP其他的一些问题，那第二个问题也必须是NP-完全问题。 （当然，所有NP-完全问题可在多项式时间规约到彼此;见练习11-2）。

这给了我们关于困难度的一个有用的概念，但到目前为止，我们还没有确定，存在一个像NP-完全的问题，更别说发现之一。我们将如何做呢？去救援Cook和Levin！

在70年代初期，Steven Cook证明，确实是有这样的问题，后来不久，LEenoid Levin独立证明了同样的事情。他们都表明，称为布尔可满足性的一个问题，或SAT，是NP-完全问题。这一结果已由他们两个命名，现在被称为Cook - Levin定理。这个定理，给了我们的出发点，是相当先进的，在这里我不能给你一个完整的证明，但我会尽力说明主要的思路。 （完整的证明是由Garey和Johnson给出;请参阅“参考”一节）。

SAR问题需要逻辑公式，如（A或非B）和（B或C），并询问是否有办法使它为真（也就是满足它）。当然，在这种情况下，是存在的。例如，我们可以设置A = B =真。为了证明这是NP-完全，考虑一个任意的问题FOO在NP中，你如何将其规约到SAT。我们的想法是先构造一个NTM，将在多项式时间内解决FOO。通过定义这是可能的（因为FOO在NP中）。然后，对于FOO（即，对于给定输入到机器）的bar的给定实例，你构造（在多项式时间内）的逻辑表达式（多项式大小的）表达如下：

\*机器的输入是bar

\*机器正确的执行他的任务

\*机器停止，回答“是”

有技巧的部分是你如何使用布尔代数表达，但一旦你这样做，在NTM很清楚，实际上，通过这种逻辑公式给出了SAT问题的模拟。如果公式是可满足的，也就是说，如果（且仅当），我们可以通过分配真值给不同变量使得结果为真（表示，除其他事外，由机器做的神奇选择），然后原始问题的答案应该是“是”。

总括来说，Cook - Levin定理说，SAT是NP-完全的，证明基本上给你一种模拟了一种NTM解决SAT问题的方式。这适用于基本的SAT问题及其相近的问题，电路-SAT，是我们使用逻辑（数字）电路，而不是用逻辑表达式。

这里有一个重要的想法是，所有的逻辑公式可以写成合取范式（CNF），也就是说，作为一个合取（一系列的与）连接的子句，其中每个子句是或连接的序列。变量的每次出现可以是A或它的否定的，非A.公式可能一开始不是以CNF的形式，但它们可以被自动（并有效）转换。考虑，例如，在式A与（B或（C与D））。它是完全等效与此其它公式，也就是CNF的形式：A与（B或C）与（B或D）。

因为任何公式可以有效地改写到一个（不是太大）CNF形式，不应该是一个惊喜，CNF-SAT是一个NP-完全问题。有趣的是，即使我们限制每个子句变量的数目为k，并得到所谓的K-CNF-SAT（或简称K-SAT）的问题，我们仍然可以证明是NP-完全，只要K>2。你会看到许多NP-完全性证明是基于一个事实，即3-SAT是NP-完全问题。

2-SAT是NP-完全问题吗？谁知道呢…

当处理复杂性问题的时候，你需要知道特例。例如，背包问题（或是求子集和，你会遇到）的变化被用于加密。问题是，背包问题很多情况下是很容易解决的。事实上，如果该背包容量有一个多项式界（如有东西计数的函数），那么问题是在P中（见练习11-3）。如果在构建问题实例时不小心，加密可以很容易解开。

在K-SAT时我们有类似的情况。对于k≥3中，这个问题是NP-完全问题。对于k =2，尽管它可在多项式时间内解决。或考虑最长路径问题。一般情况下这是NP-难的，但如果你碰巧知道你的图是一个有向无环图，你可以在线性时间内解决它。即使最短路径问题是，事实上，在一般情况下NP-难的。这里的解决方案是假设没有负环的。

如果你做加密相关的工作，这种现象是个好消息。这意味着，即使你遇到了问题，其一般形式是NP-完全问题，你需要处理的特殊具体情况是在P中的.你可以称之为困难度的不稳定性的一个例子。稍微调整你的问题的要求可以导致巨大的差异，使一个棘手的问题容易处理，或不可判定的问题（如停机问题）可判定。这就是为什么近似算法（稍后讨论）是如此有用的原因。

这是否意味着2-SAT不是NP-完全问题？当然没有。得到这个结论是一个容易掉进去的陷阱。这将是真的，只有当P≠NP，因为否则在P中的所有问题都是NP-完全问题。换句话说，我们的NP-完全性证明2-SAT失败，我们可以证明它在P中，但我们不知道他不在NPC中。

现在我们有一个开始的地方：SAT和他相近的朋友，电路SAT和3-SAT.目前还有很多问题需要研究，虽然和复制Cook和Levin的壮举似乎有点令人生畏。怎么样，比如，你会表明，每一个NP问题可以通过找到一条通过一列城镇的旅行线路而解决？

这是我们（终于）开始规约的工作。让我们来看看的相对简单的NP完全问题之一，即找到一个汉密尔顿环。在第5章我已经触及这个问题（在“Island-Hopping in Kaliningrad”的侧边栏）。问题是要确定有n个节点的图是否具有长度为n的环，也就是，您是否可以沿着图中的边访问每个节点恰好一次，并返回到您的出发点？

这并不立即看起来像SAT问题表达那么明确，我们将接触命题逻辑的所有语言，NTM看起来有点像编码。正如你所看到的，它并不是。汉密尔顿环问题可以表达为SAT问题。我的意思是，SAT可以在多项式时间内规约到汉密尔顿环问题。换句话说，我们可以使用的解决哈密尔顿环问题的机器上创建一个SAT解决机！

我将向您介绍细节，但在此之前我做的，我想请你在你的脑海里有一个大的概念：我们正在做的总体思路是，我们正在处理的一个问题作为一种机器，而我们差不多是编程那台机器去解决不同问题。考虑到这一点，让我们看看我们如何能够将布尔公式编码为图，让汉密尔顿环可满足…

为了简单起见，我们假设我们要满足公式为CNF形式。我们甚至可以假设是3-SAT（虽然这不是真的有必要）。这意味着我们有一系列子句要可满足，并且每个，我们需要满足的至少一个元素，其可以是变量（如A）或它们的否定（非A）。真值需要通过路径和环来表示，所以我们说，我们将每个变量的真值编码为路径的方向。

这个想法由图11-3显示。每个变量由一行节点表示，并且这些节点由反向的边链接在一起，以便我们将节点能够从左边移动到右边，或从右到左。一个方向（例如，左到右）表示该变量被设置为真，而另一个方向是指假。节点的数量是不重要的，只要有足够。

在我们开始尝试真正对公式进行编码之前，我们想迫使我们的机器每个变量设置为两种可能的逻辑值中的一个。也就是说，我们要确保任何汉密尔顿环会通过每一行（通过给我们的真值的方向）。我们还必须确保当从一行到另一行的时候环可以自由的切换方向，因此变量可以彼此独立的分配。我们可以通过连接每一行的任一端（在图11-3突出显示）连接的每一行到下一行的两个边缘，锚点，如图11-4所示。

如果我们只有一组连接的行，如图11-4，那么在图中将没有汉密尔顿环。我们只能从一行到另一行，没有办法再回来。最后修改行的基本结构，就是添加一个源节点s的顶部（与边缘到第一行的左，右锚）和一个宿节点t在底部（与边缘和最后一排的右锚），然后加入一边从t到s

在继续之前，您应该说服自己，这个结构确实是我们想要的。对于k变量，我们迄今建造的图形都会有2^K个不同的汉密尔顿环，每个可能的赋值给变量的真值，通过在给定行中的环是向左或者向右的来表示。

现在，我们已经将分配一系列逻辑变量真值的想法编码到我们汉密尔顿机器中，我们只需要编码包含这些变量实际公式的一种方式。我们可以通过对每个子句引入单个节点来做到这一点。一个汉密尔顿环将要访问每个点一次。诀窍是将这些子句的节点挂钩到我们现有的行上，要利用的是这些行已经编码的真值。我们设置了这些，这样的环可以通过子句的节点绕过，但只有当它朝右边。因此，举例来说，如果我们有子句（A或非B），我们将对A行添加一个弯路，使得环从左到右，我们添加另一个弯路（通过相同子句的节点）到B行，但此时由右至左（因为是非）。我们需要注意的唯一的事情是，没有两个弯路可以链接到在行同样的地方，这就是为什么每一行我们需要有多个节点，所以对于所有子句有足够的。在图11-5的例子中，你可以看到这是如何工作的。

以这种方式对子句编码后，每个子句是可满足的，只要其中至少一个变量有向右的真值，让它通过子句节点绕道而行。由于汉密尔顿环必须访问每个节点（包括每个子句的节点），公式与的部分是可满足。换言之，逻辑变大式是可满足的，当且仅当我们构建的图中有一个汉密尔顿环。这就是说，我们已经成功地将SAT（或者更具体地，CNF-SAT）规约到的汉密尔顿环问题，从而证明后者是NP-完全的！现在，还有这么难吗？

好了，只是有些困难。至少你自己像这样思考一些事将是非常具有挑战性的。幸运的是，大量的NP-完全问题很多比SAT更像汉密尔顿环问题，您将在下面看到的。

* [**11-5 未完成**](https://github.com/owlman/PythonAlgorithms/wiki/11-5)
* [**11-6**](https://github.com/owlman/PythonAlgorithms/wiki/11-6)**When the Going Gets Tough, the Smart Get Sloppy当事情变得艰难时，聪明人开始犯错**

As promised, after showing you that a lot of rather innocent-looking problems are actually unimaginably hard, I’m going to show you a way out: sloppiness. I mentioned earlier the idea of “the instability of hardness,” that even small tweaks to the problem requirements can take you from utterly horrible to pretty nice. There are many kinds of tweaks you can do—I’m only going to cover two. In this section, I’ll show you what happens if you allow a certain percentage of sloppiness in your search for optimality; in the next section, I’ll have a look at the “fingers crossed” school of algorithm design.

以上为您展示了许多问题，它们在第一眼看起来非常简单，但仔细思考过后，会发现要解决它们实际上无法想象地困难。不要紧，我之前就向您保证过，会给您指一条解决之路：“粗心”。我之前提过“困难度的不稳定性”，也就是说，有的时候，只需要对问题进行小小的调整，就可以把它从几乎无解的境地变成柳暗花明。有很多种调整问题的方式，在这里我只会提及两种。本节中，我将向您展示，在寻求优化时，如果问题条件允许一定程度的修改，会发生什么。在下一节中，我们来看一下算法设计中的“祈祷派”。

Let me first clarify the idea of approximation. Basically, we’ll be allowing the algorithm to find a solution that may not be optimal, but whose value is at most a given percentage off. More commonly, this percentage is given as a factor, or approximation ratio. For example, for a ratio of 2, a minimization algorithm would guarantee us a solution at most twice the optimum, while a maximization problem would give us one at least half the optimum. Let’s see how this works, by returning to a promise I made back in Chapter 7.

首先我来阐述一下近似法的概念。基本上，我们要允许我们的算法会找到这样一个解决方案：这个解决方案可能不是最优的，但这个解决方案是在某种程度上较好的。更一般化地，这个“程度”是以某个因素或系数的形式给出的。例如，如果系数为2的话，一个最小限度的算法应该保证解决方案最多为最优解的两倍。而最大限度的算法给出的答案至少会是最优解的一半。如同我在第七章承诺您的，我们来看一下它是如何工作的。

What I said was that the unbounded integer knapsack problem can be approximated to within a factor of two using greed. As for exact greedy algorithms, designing the solution here is trivial (just use the same greedy approach as for fractional knapsack); the problem is showing that it’s correct. How can it be that, if we keep adding the item type with the highest unit value (that is, value-to-weight ratio), we’re guaranteed to achieve at least half the optimum value? How on Earth can we know this when we have no idea what the optimum value is?

首先，关于整数型无界限背包问题，使用贪心算法思想可以把问题近似为因子为2的问题。在这里，并没有必要（与分数背包问题一样）使用准确的贪心算法设计解决方案。问题显示它是正确的。那么，如果我们每次都向背包中放入最大值的单位（也就是权重最大的单位），为什么就能保证我们至少会达到最优解的一半？我们实际上并不知道最优解是多少，我们又到底怎么得出这个结论的呢？

This is the crucial point of approximation algorithms. We don’t know the exact ratio of the approximation to the optimum—we only give a guarantee for how bad it can get. This means that if we get an estimate on how good the optimum can get, we can work with that instead of the actual optimum, and our answer will still be valid. Let’s consider the maximization case. If we know that the optimum will never be greater than A, and we know our approximation will never be smaller than B, we can be certain that the ratio of the two will never be greater than A/B.

这便是近似算法的关键所在。我们并不知道最优解的精确值——我们只给出解答最坏情况的保证。这意味着如果我们估算最优解的值，我们就可以基于这个估算来工作，而不是实际的最优解，而我们的答案却仍然是有效的。让我们考虑一下最大限度情况。如果我们知道最优解总是不会比A大，而我们的近似解答总是不会比B小，我们就能确认最优解与近似解的系数总是不会大于A/B。

For the unbounded knapsack, can you think of some upper limit to the value you can achieve? Well, we can’t get anything better than filling the knapsack to the brim with the item type with the highest unit value (sort of like an unbounded fractional solution). Such a solution might very well be impossible, but we certainly can’t do better. Let this optimistic bound be A.

对于无界限背包问题，让我们来考虑一下最优解的上限。首先，我们放入的物品的总重量无法超过背包的重量上限（有点像分数型无界限问题）。在大多数情况下，总重量等于重量上限是不能能的，但我们当然不可能比这个值更好。所以我们得出了这个最优解上限A。

Can we give a lower bound B for our approximation, or at least say something about the ratio A/B? Consider the first item you add. Let’s say it uses up more than half the capacity. This means we can’t add any more of this type, so we’re already worse off than the hypothetical A. But we did fill at least half the knapsack with the best item type, so even if we stop right now, we know that A/B is at most 2. If we manage to add more items, the situation can only improve.

我们可以给出我们近似解的下线吗？或者，至少得到一些关于系数A/B的信息？让我们来考虑一下你放入的第一个物品。让我们假设它占据了容量的一半。这意味着我们无法放入更多同样的这种物品，从而我们已经得到了一个解，这个解诚然比假设的情况A要差，但是由于我们已经填满了背包容量的一半，所以即使我们现在停止装填，我们已经知道了A/B至少高于2.如果我们能够放入更多的其他物品，我们也只是得到了更好的解，而不会影响这个结论。

What if the first item didn’t use more than half the capacity? Good news, everyone: we can add another item of the same kind! In fact, we can keep adding items of this kind until we’ve used at least half the capacity, ensuring that the bound on the approximation ratio still holds.

那么，如果第一个物品没有占满背包容量的一半呢？各位，这是一个好消息：我们可以再放入一个同样的物品了！实际上，我们可以持续放入同样的物品，直到超过容量的一半，这保证了我们刚刚得出的近似系数仍然成立。

There are tons and tons of approximation algorithms out there—with plenty of books about this topic alone. If you want to learn more about the topic, I suggest getting one of those (such as the excellent Approximation Algorithms, by Vijay V. Vazirany). I will show you one particularly pretty algorithm, though, for approximating the metric TSP problem.

世界上有成千上万的近似算法——单就关于这个话题就有很多本著作。如果您希望深入这个话题，我建议您去读一本（其中的佼佼者如Vijay V. Vazirany的《近似算法》），在这里，我只向你展示其中一个非常好的算法，用于近似TSP问题的数据。

What we’re going to do is, once again, to find some kind of invalid, optimistic solution, and then tweak that until we get a valid (but probably not optimal) solution. More specifically, we’re going to aim for something (not necessarily a valid Hamilton cycle) that has a weight of at most twice the optimum solution, and then tweak and repair that something using shortcuts (which the triangle inequality guarantees won’t make things worse), until we actually get a Hamilton cycle. That cycle will then also be at most twice the length of the optimum. Sounds like a plan, no?

再说一遍，我们要做的，是找到一种不正确的、乐观的情况，然后改进它，知道我们找到一个正确的（但可能不是最优的）解。说的再明确一些，我们的目标是找到一个解（不一定是一个哈密顿回路），这个解最多是最优解的两倍，然后通过寻找这个解中的捷径路径来改进这个解，知道我们得到一个哈密顿回路。所以这个贿赂一定仍然小于最优解的两倍。这听起来尚能称为一个计划，对吧？

What, though, would be only a few shortcuts away from a Hamilton cycle, and yet be at most twice the length of the optimum solution? We can start with something simpler: what’s guaranteed to have a weight that is no greater than the shortest Hamilton cycle? Something we know how to find? A minimum spanning tree! Just think about it. A Hamilton cycle connects all nodes, and the absolutely cheapest way of connecting all nodes is using a minimum spanning tree.

然而，怎么找到一个离哈密顿回路差不了多少，又在最优解两倍以内的近似解呢？我们可以从更简单的情况开始：怎么得到不大于最短哈密顿回路的权重呢？我们已经学到的知识中有解决方法吗？最小生成树！让我们来思考一下。哈密顿贿赂连接了所有的点，而最小的连接了所有点的算法是最小生成树。

A tree is not a cycle, though. The idea of the TSP problem is that we’re going to visit every node, walking from one to the next. We could certainly visit every node following the edges of a tree, as well. That’s exactly what Trémaux might do, if he were a salesman (see Chapter 5). In other words, we could follow the edges in a depth-first manner, backtracking to get to other nodes. This gives us a closed walk of the graph but not a cycle (because we’re revisiting nodes and edges). Consider the weight of this closed walk, though. We’re walking along each edge exactly twice, so it’s twice the weight of the spanning tree. Let this be our optimistic (yet invalid) solution.

然而，树并不是一个回路。TSP问题本身要求我们不但要访问每个节点，而且需要从一个节点到另一个节点地访问。我们当然可以沿着树的变访问所有的点：如果Tremaux是一个销售商，他就会使用这种算法（见第五章）。也就是说，我们可以用深度优先算法沿着最小生成树的边访问所有的节点，这是较好的遍历节点的方法，但它并不是一个环（因为我们总是在重复访问节点与边）。再思考一下这种路径，我们会发现我们走过的每条路径正好被走了两次，也就是说，总路径恰巧是生成树路径和的两倍。我们就把这个解（虽然它不符合条件）作为我们的优化。

The great thing about the metric case is that we can skip the backtracking and take shortcuts. Instead of going back along edges we’ve already seen, visiting nodes we’ve already passed through, we can simply make a beeline for the next unvisited node. Because of the triangular inequality, we’re guaranteed that this won’t degrade our solution, so we end up with an approximation ratio bound of two! (This algorithm is often called the “twice around the tree” algorithm, although you could argue that the name doesn’t really make that much sense, as we’re only going around the tree once.)

现在好消息来了，我们可以开始跳过回溯，使用捷径提到。比起沿着我们已经走过的边走回去，访问我们已经访问过的节点，我们可以选择直接到下一个节点的路径。由于三角形的一条边的长度总是比另两条边的长度和要小，我们可以保证这种选择不会让我们的总路径增大。通过不断重复这个过程，最终我们得到了一个结论：我们得到了最优解的两倍上界！（这个算法经常被称为“树的两倍遍历”（自己乱编的）算法，可能你会对这个名字有异议，因为它并不那么确切地描述了这种算法，因为我们我们只遍历了树一次）。

Implementing this algorithm might not seem entirely straightforward. It kinda is, actually. Once we have our spanning tree, all we need is to traverse it, and avoid visiting nodes more than once. Just reporting the nodes as they’re discovered during a DFS would actually give us the kind of solution we want. You can find an implementation of this algorithm in Listing 11-1.

这个算法的实现可能看起来没有那么直观。不过实际上它使用的就是这种思想。一旦我们得到了生成树，我们所要做的就是遍历它，然后避免访问节点多于一次。在DFS中报告所经过的节点，这实际上就是我们所需要的解决方案了。你可以在列表11-1中看到这个算法的一个实现。

Listing 11-1. The “Twice Around the Tree” Algorithm, a 2-Approximation for Metric TSP

列表11-1 “树的两倍遍历”算法，TSP最优解的参数为2的近似。

There is one way of improving this approximation algorithm that is conceptually simple but quite complicated in practice. It’s called Christofides’ algorithm, and the idea is that instead of walking the edges of the tree twice, it creates a min-cost matching among the odd-degree nodes of the spanning tree. This means that you can get a closed walk by following the edges of the tree once, and the edges of the matching once (and then fixing the solution by adding shortcuts, as before). We already know that the spanning tree is no worse than the optimum cycle. It can also be shown that the weight of the minimum matching is no greater than half the optimum cycle (Exercise 11-15), so in sum, this gives us a 1.5-approximation, the best bound known so far for this problem. The problem is that the algorithm for finding a min-cost matching is pretty convoluted (it’s certainly a lot worse than finding a min-cost bipartite matching, as discussed in Chapter 10), so I’m not going to go into details here.

这种近似算法有一种改进，这种改进在概念上非常简单，但在实现时很复杂。这种算法称为Christofides算法。它的原则是，比起遍历树的边两次，匹配生成树中度为奇数的节点花费更少。也就是说，你可以先沿着树的边行走一次，然后再沿着技术节点的匹配边一次（然后像之前那样规约捷径）。我们已经知道了生成树不会比最优回路更坏。最小匹配的权重不大于最优回路的一半也是可以得到证明的（练习11-15），所以在总和上，这个优化的近似算法给了我们一个系数为2的近似——这是这个问题迄今为止最好的结果。问题是找到最小匹配的花费是很费周章的（当然，在第10章中我们讨论过，这比找到找到最小花费的双向匹配还要糟糕得多），所以在这里，我们不会讨论这个算法的实现细节。

Given that we can find a solution for the metric TSP problem that is a factor of 1.5 away from the optimum, even though the problem is NP-hard, it may be a bit surprising that finding such an approximation algorithm—or any approximation within a fixed factor of the optimum—is itself an NPhard problem for TSP in general (even if the TSP graph is complete). This is, in fact, the case for several problems, which means that we can’t necessarily rely on approximation as a practical solution for all NP-hard optimization problems.

假设我们可以找到TSP问题对于最优解的系数为1.5的近似解，那么由于这个问题是NP-难的，那么我们找到这个近似解的算法本身也是NP难的，这个结论也就不那么令人吃惊了（即使TSP图是一个完全图）。实际上，这是很多难题的情况，也就是说，依赖近似法，把它应用于所有的NP难问题，本身是不可取的。

To see why approximating TSP is NP-hard, we do a reduction from the Hamilton cycle problem to the approximation. You have a graph, and you want to find out whether it has a Hamilton cycle. To get the complete graph for the TSP problem, we add any missing edges, but we make sure we give them huge edge weights. If our approximation ratio is k, we make sure these edge weights are greater than km, where m is the number of edges in the original graph. Then an optimum tour of the new graph would be at most m if we could find a Hamilton tour of the original, and if we included even one of the new edges, we’d have broken our approximation guarantee. That means that if (and only if) there were a Hamilton cycle in the original graph, the approximation algorithm for the new one would find it—meaning that the approximation is at least as hard (that is, NP-hard).

为了了解为什么得到TSP问题的近似解是NP难的，我们可以为哈密顿回路问题做一个近似规约。你有一个图，然后希望知道这个图是否有哈密顿贿赂。为了补全TSP图，我们增加了所有的消失的边，但是我们给了这些边很大的权重。如果我们的近似系数为k，我们要保证我们增加的每条边的权重都大于km，其中，m是原图中边的总数。所以，如果我们的哈密顿回路完全建立在原图之上，那么我们的回路的权重总数最多是m，而如果我们的哈密顿回路里只要有一条我们增加的边，我们就马上超过了我们的近似算法的权重边界。这意味着如果原图中有且仅有一条哈密顿回路，近似算法总是能够找到且仅能找到这个回路，换句话说，近似算法与原问题一样，那么近似算法本身也是NP难的。

* [**11-7**](https://github.com/owlman/PythonAlgorithms/wiki/11-7)**Desperately Seeking Solutions拼命地寻找解**

We’ve looked at one way that hardness is unstable—sometimes finding near-optimal solutions can be vastly easier than finding optimal ones. There is another way of being sloppy, though. You can create an algorithm that is basically a brute force solution, but that uses guesswork to try to avoid as much computation as possible. With a little luck, if the instance you’re trying to solve isn’t one of the really hard ones, you may actually be able to find a solution pretty quickly! In other words, the sloppiness here is not about the quality of the solution, but about the running time guarantees.

我们刚刚讨论了一种解决问题的方式，它的困难度是不定的——有的时候寻找近似最优解会比寻找最优解本身容易得多得多。而其实还有一种犯懒的方式。你可以创建一个暴力的解决方案，而解决方式中可以通过混入猜想来尽量避免计算。运气好的话，如果你解决的那个问题碰巧不是最难得几个问题之一，你就可以很快地找到一个解了！也就是说，这里犯懒的点不在于解的质量，而在于找到解的保证时间。

This is a bit like with quicksort, which has a quadratic worst-case running time, but which is loglinear in the average case, with very low constant factors. Much of the reasoning about hard problems deals with what guarantees we can give about the worst-case performance, but in practice, that may not be all we care about. In fact, even if we’re not in Russel Impagliazzo’s fantasy world, Algorithmica, we may be in one of his other worlds, which he calls Heuristica. Here, NP-hard problems are still intractable in the worst case, but they’re tractable in the average case. And even if this isn’t the case, it certainly is the case that by using heuristic methods, we can often solve problems that might seem impossible.

这有点像快速排序，它的最坏情况是指数级别的，但平均情况是对数线性的，并且常系数非常小。很多难题的解答实际上是在回答解决这个问题的最坏状况是什么，然而在实践中，最坏情况通常不是我们考虑的所有情况。实际上，即使我们不在Russel Impagliazzo的奇妙世界——算法，我们也可能在他的其他的世界里，他称为启发。在这里，NP难问题在最坏情况下仍然是痕迹搜的，但是它们在一般情况下是可解决的。而即使不是这种情况，我们也可以使用启发法，使得不可能解决的方法变成可能。

There are plenty of methods in this vein. The A\* algorithm discussed in Chapter 9, for example, can be used to search through a space of solutions in order to find a correct or optimal one. There are also such heuristic search techniques as artificial evolution and simulated annealing (see “If You’re Curious …” later in this chapter). In this section, though, I’ll show you a really cool, and actually pretty simple, idea, which has applications both for hard problems, such as those discussed in this chapter, but which can also serve as a quick-and-dirty way of solving any kind of algorithmic problem, even ones for which there are polynomial solutions. This could be useful either because you can’t think of a custom algorithm, or because your custom algorithm is too slow.

在这个领域有很多方法，例如，在第九章我们讨论过的A\*法，可以用于在一个空间中寻找数个解中正确的一个或是最优的一个。也有很多这种启发寻找技术，例如人工进化法、模拟退火法（见本章后续的“补充小结”），而在本节中，我会用一个又简单又酷的例子来做例子，其中同时使用了这两种技术，来提供一种快糙猛的解决任何算法问题的方式，包括哪些多项式解决方法。这种方法在你无法解出一个问题，或是解出算法过慢的情况下都非常有效。

The technique is called branch and bound, and is particularly well-known in the field of artificial intelligence. There’s even a special version of it (called alpha-beta pruning) used in programs playing games. (For example, if you have a chess program, chances are there’s some branch and bound going on inside it.) In fact, branch and bound is one of the main tools for solving NP-hard problems, including such general and expressive ones as integer programming. Even though this awesome technique follows a very straightforward schema, it can be hard to implement in a completely general fashion. Chances are, if you’re going to use it, you’ll have to implement a version that is customized to your problem.

这种技术被称为分支限界技术，它在人工智能界非常有名。甚至它的特殊版本（称为alpha-beta剪枝）被用于游戏设计。（例如，在棋类游戏中，你经常面临选择问题）。实际上分支衔接技术是解决NP难问题的主要工具之一，包括了普遍的而又有表现力的整数规划问题。虽然这个厉害的技术的模式非常直接，但是很难将它的算法一般化地实现出来。很可能出现的情况是，如果你要使用它，你必须为你的问题写一个这个算法的订制化实现。

Branch and bound, or B&B, is based on gradually building solutions, sort of like a lot of greedy algorithms (see Chapter 7). In fact, which new building block to consider is often chosen greedily, resulting in so-called best-first branch and bound. However, instead of fully committing to this new building block (or this way of extending the solution), all possibilities are considered. At the core, we’re dealing with brute-force solution. The thing that can make it all work, though, is that whole avenues of exploration can be pruned away, by reasoning about how promising (or, rather, unpromising) they are.

分支限界技术，也就是B&B技术，基于逐渐地建立解决方式，有点像很多贪心算法（见第七章）。事实上，在诸多中采用贪心算法做出选择的过程，也被称为“考虑到所有可能情况下的使用分支限界技术（或者这种方式的扩展解决方案）所做出的做好的最好的第一个选择”，在核心上，我们在给出一个暴力解决方法，而让一切能够运作的，是在一半情况下，它可以通过一定的方法，根据选项的前景作出选择。

To make this more concrete, let’s consider a specific example. In fact, let’s revisit one we’ve worked with in several ways before, the 0-1 knapsack problem. In 1967, Peter J. Kolesar published the paper “A branch and bound algorithm for the knapsack problem,” where he describes exactly this approach. As he puts it, “A branch and bound algorithm proceeds by repeatedly partitioning the class of all feasible solutions into smaller and smaller subclasses in such a way that ultimately an optimal solution is obtained.” These “classes” are what we get by constructing partial solutions.

为了让这个概念更加具体化，我们来考虑一个例子。实际上，我们可以重新考虑一个已经给出很多种解答方法的问题：0-1背包问题。在1967年，Peter J. Kolesar发票了一篇论文《背包问题的分支限界算法》。在其中它写道，“分支限界算法不断的将各种可行的解决方案逐一分解为小块，最终在其中获得最好的解”，这恰如其分地给出了它的定义。这里的“块”就是我们得到的解决方案的一部分。

For example, if we decide to include item x in our knapsack, we have implicitly constructed the class of all solutions including x. There is, of course, also the complement of this class, all solutions that do not include x. We will need to examine both of these classes, unless we can somehow reach the conclusion that one of them cannot contain the optimum. You can picture this as a tree-shaped state space, a concept mentioned in Chapter 5. Each node is defined by two sets: the items that are included in the knapsack, and the items that are excluded from it. Any remaining items are as yet undetermined.

例如，如果我们决定在背包中放入物品x，我们隐含地建立了一个集合，这个集合中包含了所有x被放入背包的情况。当然，相对的，也就有了另一个集合，在这个集合中x都不被包含于其中。然后我们需要开始检查这两个集合，知道我们可以得出最优解不在其中一个集合中的结论。如同第五章所阐述的那样，你可以把这个过程想想成一个树形的状态图，每个节点都被两个集合定义：在一个集合中，背包不包含这个节点；在另一个集合中，背包包含这个节点。剩下的节点都没有被决定。

In the root of this (abstract, implicit) tree structure, no objects are included or excluded, so all are undetermined. To expand a node into two child nodes (the branching part), we decide on one of the undecided objects and include it to get one child, and exclude it to get the other. If a node has no undecided items, it’s a leaf, and we can get no further.

在这个（抽象的，暗含的）树结构的节点上，没有物品处于“在”或者“不在”背包的状态下，所以所有的物品的状态都是不确定的。每作出一个物品在或者不再背包中的状态，我们实际上都将一个节点分裂成了两个节点（也就是两个分支），然后做出选择，抛弃其中的一个节点以其所属的分子。如果一个节点再没有状态不定的物品，它就是一个叶子节点，在那里算法结束。

It should be clear that if we explore this tree fully, we will examine every possible combination of included and excluded objects (a brute force solution). The whole idea of branch and bound algorithms is to add pruning to our traversal (just like in bisection and search trees), so we visit as little as possible of the search space. As for approximation algorithms, we introduce upper and lower bounds. For a maximization problem, we use a lower bound on the optimum (based on what we’ve found so far), and an upper bound on the solutions in any given subtree (based on some heuristic).19 In other words, we’re comparing a conservative estimate of the optimum with an optimistic estimate of what we can find in a given subtree. If the conservative bound is better than the optimistic bound on what a subtree contains, that subtree cannot hold the optimum, and so it is pruned (the bounding part).

很清楚，如果我们完全探索这棵树，我们就会检查所有物品被包含或者不被包含的解（也就是暴力寻解）。而分支限界法的核心思想，就是在每做出一个选择后剪除另一个选择下的所有剩余情况（就像二分选择和查找树做的那样），所以我们做出了最少的选择决定。对于近似算法，我们引入了上界与下解，对于最大值问题，我们为最优解设置了一个下届（基于我们目前找到什么），而对于任何给定的字数，我们则设置了一个上界（基于某种启发法）。换句话说，我们首先对于子树的含有最优解的情况做了保守的估计，再根据这种估计作出选择。如果保守范围比最优范围大，那么子树就不可能含有最优解，所以它被剪除了。

In the basic case, the conservative bound for the optimum is simply the best value we’ve found so far. It can be extremely beneficial to have this bound be as high as possible when the B&B starts running, so we might want to spend some time on that first. (For example, if we were looking for a metric TSP tour, which is a minimization problem, we could set the initial upper bound to the resulf of our approximation algorithm.) To keep things simple for our knapsack example, though, let’s just keep track of the best solution, starting out with a value of zero. (Exercise 11-16 asks you to improve on this.)

在基本的例子中，最优解的保守边界只是我们目前找到的最好的值。这个值在B&B算法开始之前越高越好，所以我们可能希望在B&B开始之前先优化这个值。（例如，如果我们正在寻找一个数值型TSP问题的解，这是个最小值问题，那么我们可以首先将初始的商界设置为我们近似算法的解答。为了让背包问题的例子变得简单一些，我们只考虑最佳解决方案，从值为0开始（练习11-16要求你改进它）。

The only remaining conundrum is how to find an upper bound for a partial solution (representing a subtree of the search space). If we don’t want to lose the actual solution, this bound has to be a true upper bound; we don’t want to exclude a subtree based on overly gloomy predictions. Then again, we shouldn’t be too optimistic (“This might have infinite value! Yay!”) because then we’d never get to exclude anything. In other words, we need to find an upper bound that is as tight (low) as we can make it. One possibility (and the one used by Kolesar) is to pretend we’re dealing with the fractional knapsack problem, and then use the greedy algorithm on that. This solution can never be worse than the actual optimum we’re looking for (Exercise 11-17), and it turns out it’s a pretty tight bound for practical purposes.

那么唯一剩下的问题是如何找到一个原子问题（表示搜索范围的子树）的上界。如果我们不希望丢失一个解，这个上界必须是一个真实的上界；我们不希望根据一个过度负面的假设而剪除一个子树。以及，再次地，我们不应该太乐观（这个分支可能含有无数的值！耶！），不然我们不能剪除任何字数。换句话说，我们需要找到一个我们可以做出的最严格的上界。一个可能性（同时也是Kolesar使用的可能性）是假装我们我们在解决分数背包问题，然后使用贪心算法来解决它。解决方案可能总是不比实际我们在寻找的最优解差（联系11-17），而最终这个上界在实践中也是一个非常严格的上界。

You can see one possible implementation of the 0-1 knapsack B&B in Listing 11-2. To keep things simple, the code only calculates the value of the optimum solution. If you want the actual solution structure (which items are included), you’ll need to add some additional bookkeeping. As you can see, instead of explicitly managing two sets for each node (included and excluded items), only the weight and value sums of items included so far are used, with a counter (m) indicating which items have been considered (in order). Each node is a generator, which will (when prompted) generate any promising children.

在列表11-2中，你可以看到一个使用B&B来解决0-1背包问题的可能的实现。为了让事情变得简单，代码只计算了最优情况下的最终值。如果你希望看到实际的解决结构（哪些物品被包含了），你可能需要往里面加入额外的代码。如同你所看到的，与其为每个节点保留两个集合（该物品被包含或者不被包含于背包内），我们使用的只有物品的权重和值的和，然后使用了一个计数器m来表达哪些物品（按照顺序）被考虑过。每个节点都是一个生成器，用于生成之后的子节点。

* [**11-8**](https://github.com/owlman/PythonAlgorithms/wiki/11-8)**And the Moral of the Story Is …故事的寓意是……**

All right. This chapter may not be the easiest one in the book, and it may not be entirely obvious how to use some of the topics here in your day-to-day coding. To clarify the main points of the chapter, I thought I’d try to give you some advice on what to do when a monster problem crosses your path.

好吧，本章不会是本书最简单的一张，而如何在你日复一日的编程工作中如何融入本章所学也不是一个很明朗的问题。为了总结本章的主要论点，我想我已经尝试着给了你一些建议：怎么对待你所遇到的怪兽级别的问题。

• First, follow the first two pieces of problem solving advice in Chapter 4. Are you sure you really understand the problem? Have you looked everywhere for a reduction (for example, do you know of any algorithms that seem even remotely relevant)?

首先，遵守第四章的前两个处理问题的建议。你真的明白问题的本质是什么吗？你已经将问题规约为最本质的状态了吗（比如，你是否知道现有的任何一种哪怕跟这个问题只沾一点边的算法吗）？

• If you’re stumped, look again for reductions, but this time from some known NPhard problems, rather than to problems you know how to solve. If you find one, at least you know the problem is hard, so there’s no reason to beat yourself up.

如果你感到为难，再次尝试规约问题，不过这次尝试将问题规约为一个已知的NP难问题，而不是规约为一个你知道如何解决的问题。如果你找到了一个这样的例子，至少你知道你的问题是很难解决的，所以你不用怀疑自己的能力。

• Consider the last bit of problem solving advice from Chapter 4: are there any extra assumptions you can exploit to make the problem less monstrous? The longest path problem is NP-hard in general, but in a DAG, you can solve it easily.

考虑第四章最后一个解决问题的建议：你可以作出一些关于问题的假设使得这个问题弱化吗？如果问题是NP难的，那么你举步维艰；但如果它是DAG，那么你就可以很容易地解决它。

• Can you introduce some slack? If your solution needn’t be 100 percent optimal, perhaps there is an approximation algorithm you can use? You could either design one or research the literature on the subject. If you don’t need polynomial worstcase guarantees, perhaps something like branch and bound could work?

你可以引入一些弱化条件吗？如果你的解决方案并不需要是100%最优的，也许你可以用一个近似算法？你可以设计或研究问题的语义，如果你不需要保证你的解的最坏情况是多项式复杂的，也许分支限界法可以解决问题呢？

Listing 11-2. Solving the Knapsack Problem with the Branch and Bound Strategy

列表11-2 使用分支限界战略解决背包问题

If all else fails, you could implement an algorithm that seems reasonable and then use experiments to see if the results are good enough. For example, if you’re scheduling lectures to minimize course collisions for students (a kind of problem that’s easily NP-hard), you may not need a guarantee that the result will be optimal, as long as the results are good enough.

如果这些尝试都失败了，那么你可以尝试实现一些看起来可能的算法，然后做一些实验来看结果是否足够好。例如，如果你在设计一个排课表，使得学生来说课程冲突变得最小（这类问题很容易是NP难的），那么只要结果看起来足够好，你并不一定需要保证结果是最优解。

* [**11-9**](https://github.com/owlman/PythonAlgorithms/wiki/11-9) **Summary总结**

This chapter has been about hard problems and some of the things you can do to deal with them. There are many classes of (seemingly) hard problems, but the most important one in this chapter is NPC, the class of NP-complete problems. NPC forms the hard core of NP, the class of decision problems whose solutions can be verified in polynomial time—basically every decision problem of any real practical use. Every problem in NP can be reduced to every problem in NPC (or to any so-called NP-hard problem) in polynomial time, meaning that if any NP-complete problem can be solved in polynomial time, every problem in NP can be, as well. Most computer scientists find this scenario highly unlikely, although no proof as yet exists either way.

The NP-complete and NP-hard problems are legion, and they crop up in many contexts. This chapter gave you a taste of these problems, including brief proof sketches for their hardness. The basic idea for such proofs is to rely on the Cook-Levin theorem, which says that the SAT problem is NPcomplete, and then to reduce in polynomial time either from that, or from some other problem we have already shown to be NP-complete or NP-hard.

The strategies hinted at for actually dealing with these hard problems are based on controlled sloppiness. Approximation algorithms lets you control precisely how far your answer will be from the optimum, while heuristic search methods such as branch and bound guarantee you an optimal solution but can take an unspecified amount of time to finish.

本章是关于难题以及你面对难题时能做的事。难题被分为很多类，但本章最重要的一类难题是NP完全问题。NP完全问题是NP问题的核心，这类问题的决定算法的解决方案在任何实际使用中的都至少是多项式时间的。每一个NP问题都可以在多项式时间内被规约为每个NPC问题（也就是任何所谓的NP-难问题），也就是说，如果任何NP完全问题可以在多项式时间内解出，那么每个NP问题也可以在多项式时间内解出。大多数计算机科学家都发现解出的可能性微乎其微，虽然迄今为止并没有任何人能够证明它可以或者不可以。

NP完全问题和NP难问题是一个军团，并且他们经常在各种情况下出现。本章给出了其中的一些问题，也给出了它们的困难度的一些简单证明。这些证明的基本思想都基于Cook-Levin的理论：首先，SAT问题是一个NP完全问题；如果一个问题能够在多项式时间内被规约为一个SAT问题，或者任何一个通过这种规约所产生的NP或者NP难的问题，那么这个问题本身也是NP完全问题或者NP难问题。

处理这些困难问题的战略是引入可以控制的问题弱化。近似算法是你可以控制你的问题和最优解之间差多少，而启发性查找法，例如分支限界方法，可以保证提供一个最佳解，但不确定会花费多少时间完成。

If You’re Curious …

拓展阅读

There are lots of books out there that deal with computational complexity, approximation algorithms, and heuristic algorithms; see the “References” section for some ideas.

关于可极端复杂度与近似算法，启发式算法，已经有很多书籍了。见本章的“参考书籍”部分。

One area that I haven’t touched upon at all is that of so-called metaheuristics, a form of heuristic search that gives few guarantees but that can be surprisingly powerful. For example, there is artificial evolution, with so-called genetic programming, or GP, as one of its most well-known techniques. In GP, you maintain a virtual population of structures, usually interpreted as little computer programs (although they could be Hamilton cycles in the TSP problem, for example, or whatever structure you’d like to build). In each generation, you evaluate these individual (for example, computing their length when solving the TSP problem). The most promising ones are allowed to have offspring—new structures in the next generation, based on the parents, but with some random modifications (either simple mutation, or even combinations of several parent structures). Other metaheuristic methods are based on how melted materials behave when cooled down slowly (simulated annealing), how you might search for things when avoiding areas where you’ve recently looked (tabu search), or even how a swarm of insect-like solutions might move around in the state space (particle swarm optimization).

本章完全没有讨论的部分是所谓的“元启发算法”，一种启发搜索算法的形式，其并不对结果作出过多保证，但却十分强大。例如，有一种人工进化算法，叫做遗传规划（GP），是其中最著名的技术之一。在GP中，你维护一个虚拟的结构群体，很少被表达为计算机程序（虽然可以被表示为TSP问题的哈密顿回路，或者你喜欢的任何结构）。在每一次迭代中，个体被进化（例如，在解决TSP问题是计算它们的长度）。而其中最有前景的几个被允许在下一次迭代中基于上一代拥有新的特性，但又含有一些随机的改变（要么是简单的突变，要么是几个父结构的组合）。别的元启发方法基于你在搜索时如何回避已经搜索过的路径，或者基于模拟一群昆虫——就像在状态空间内你如何移动（例子群优化）。

* [**11-10**](https://github.com/owlman/PythonAlgorithms/wiki/11-10) **练习（不全）**

11-1. We’ve seen several cases where the running time of an algorithm depends on one of the values in the input, rather than the actual size of the input (for example, the dynamic programming solution to the 0-1 knapsack problem). In these cases, the running time has been called pseudopolynomial, and it has been exponential as a function of problem size. Why is bisecting for a specific integer value an exception to this?

11-1 在一些例子中，我们已经知道，一个算法的运行时间爱你基于输入的值，而不是输入的值的数量（例如，0-1背包问题的动态规划）在这些例子中，运行时间被称为伪多项式时间的，并且问题的规模是指数型的。为什么对于整数值的二分查找是它的一个特例？

11-2. Why can every NP-complete problem be reduced to every other?

11-2 为什么每个NP完全问题都可以被规约为每一个NP完全问题？

11-3. If the capacity of the knapsack problem is bounded by a function that is polynomial in the number of items, the problem is in P. Why?

11-3 如果背包问题的容量上界是由物品数量的多项式方程决定的，这个问题就是个P问题，这是为什么？

11-4. Show that the subset sum problem is NP-complete even if the target sum, k, is fixed at zero.

11-4 说明子集的和问题是NP完全问题，即使目标和，K，是0.

11-5. Describe a polynomial-time reduction from the subset sum problem with positive integers to the unbounded knapsack problem. (This can be a bit challenging.)

描述一个从正整数子集和问题到无界背包问题在多项式时间内的规约（这个问题可能有一点挑战性）。

11-6. Why is a four-coloring, or any k-coloring for k > 3, no easier than a three-coloring?

为什么一个四色图，或者任何一个K大于3的K色图着色问题，不比三色图着色问题简单？

11-7. The general problem of isomorphism, finding out whether two graphs have the same structure (that is, whether they’re equal if you disregard the labels or identities of the nodes) is not known to be NP-complete. The related problem of subgraph isomorphism is, though. This problem asks you to determine whether one graph has a subgraph that is isomorphic to another. Show that this problem is NP-complete.

一般的同型问题，也就是找出两个图是否有一样的结构的问题（也就是说，它们是否能够通过标记节点来证明其拓扑结构相等，点与边一一对应）还没有被证明是一个NP完全问题。相关的子图同型问题却是一个NP完全问题。这个问题是决定一个图是否含有另一个图的子图，请说明为什么这个问题是NP完全问题。

11-8. How would you simulate the undirected Hamilton cycle problem using the directed version? 11-9. How would you reduce the undirected Hamilton cycle problem (directed or undirected) to the undirected Hamilton path problem? 11-10. How would you reduce the Hamilton path problem to the Hamilton cycle problem? 11-11. Why don’t the proofs given in this section let us conclude that finding the longest path in a DAG is NP-complete? Where do the reductions break down? 11-12. Why haven’t we shown that the longest path problem without positive cycles is NP-complete? 11-13. In the greedy 2-approximation for the unbounded knapsack problem, why can we be certain that we can fill more than half the knapsack (assuming that at least some objects will fit in it)? 11-14. Let’s say you have a directed graph, and you want to find the largest subgraph without cycles (the largest sub-DAG, so to speak). You’ll measure the size in the number of edges involved. You think the problem seems a bit challenging, though, so you’ve decided that you’ll settle for a 2-approximation. Describe such an approximation. 11-15. In Christofides’ algorithm, why is there a matching of the odd-degree nodes with a total weight equal to at most half that of the optimum Hamilton cycle? 11-16. Devise some improvement on the starting-value for lower bound on the optimum in the branch and bound solution for the 0-1 knapsack. 11-17. Why is the greedy fractional solution never worse than the actual optimum in 0-1 knapsack? 11-18. Consider the optimization problem MAX-3-SAT (or MAX-3-CNF-SAT), where you’re trying to make as many of the clauses in a 3-CNF formula true. This is clearly NP-hard (because it can be used to solve 3-SAT), but there is a curiously effective and oddly simple randomized approximation algorithm for it: just flip a coin for each variable. Show that in the average case, this is an 8/7-approximation (assuming that no clause contains both a variable and its negation). 11-19. In Exercises 4-3 and 10-8, you started building a system for selecting friends to invite to a party. You have a numerical compatibility with each guest, and you want to select a subset that gives you a highest possible sum of compatibilities. Some guests would come only if certain others were present, and you managed to accommodate this constraint. You realize, however, that some of the guests will refuse to come if certain others are present. Show that solving the problem suddenly got a lot harder. 11-20. You’re writing a system for parallel processing that distributes batch jobs to different processors in order to get all the work done as quickly as possible. You have the processing times for n jobs, and you are to divide these among m identical processors so that the final completion time is minimized. Show that this is NP-hard, and describe and implement an algorithm that solves the problem with approximation ratio 2. 11-21. Use the branch and bound strategy and write a program that finds an optimal solution to the scheduling problem in Exercise 11-20.

1. You can assume that getting *down* from Pollux is easy enough. Perhaps there’s a water slide? [↑](#footnote-ref-1)
2. “An economics professor and a student were strolling through the campus. ‘Look,’ the student cried, ‘there’s a $100 bill on the path!’ ‘No, you are mistaken,’ the wiser head replied. ‘That cannot be. If there were actually a $100 bill, someone would have picked it up.’” (From *Compensation*, by G. T. Milkovich and J. M. Newman.) [↑](#footnote-ref-2)