**[知其所以然（以算法学习为例）](http://mindhacks.cn/2008/07/07/the-importance-of-knowing-why/)**

**BY**

[**刘未鹏**](http://mindhacks.cn/author/pongba/)

 – JULY 7, 2008**POSTED IN:**[学习方法](http://mindhacks.cn/topics/learning-method/), [算法](http://mindhacks.cn/topics/algorithms/)

**Updated(2008-7-24)：更新见正文部分，有标注。**

其实下文的绝大部分内容对所有学习都是同理的。只不过最近在[正儿巴经地](http://blog.csdn.net/pongba/archive/2008/06/05/2513263.aspx)学算法，而后者又不是好啃的骨头，所以平时思考总结得就自然要比学其它东西要多一些。

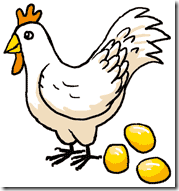
问题：目前几乎所有的算法书的讲解方式都是欧几里德式的、瀑布式的、自上而下的、每一个推导步骤都是精准制导直接面向目标的。由因到果，定义、引理、定理、证明一样不少，井井有条一丝不乱毫无赘肉。而实际上，这完全把人类大脑创造发明的步骤给反过来了。看起来是阳关大道，实际上车马不通。

[](http://mindhacks.cn/wp-content/uploads/2009/02/clip-image001.jpg)

而对读者来说，这就**等于直接告诉你答案&做法了**，然后**让你去验证**这个答案&做法是可行&成立的。而关于答案&做法到底是怎么来的，从问题到答案之间经历了怎样的思维过程。却鲜有书能够很好的阐释。就我有限的阅（算法）书经验，除了波利亚的《怎样解题》还算合格之外（也并非最理想），其它的（包括有名的《算法导论》、《如何解题：现代启发式方法》、《Algorithms》、《编程珠玑》，甚至TAOCP——公平地说由于高老大对算法领域历史了解得非常通透，所以许多地方能够从原始脉络来讲述一个问题，譬如令人印象深刻的从竞赛树到堆的讲解就寥寥一页纸道出了堆这个数据结构的本质来，而像刚才列的几本有名的书却都没有做到），在思维的讲述上都算不上合格**（当然不是说这些书没有价值，作为知识性的参考书籍，它们将知识整理出系统结构，极大的便利了知识的掌握，就像《什么是数学》所做的工作一样）**，为什么我这么说呢，因为我发现每每需要寻找对一个算法的解释的时候，翻开这些书，总是直接就看到关于算法逻辑的描述，却看不到整个算法的诞生过程背后的思想。

**我们要的不是相对论，而是诞生相对论的那个大脑。我们要的不是金蛋，而是下金蛋的那只鸡。**

***Update(2008-7-24):****收到不少同学的批评，想来这个开头对一些著作的语气过重了，实际上，注意，****我完全不否认这些著作的价值，我自己也在通过阅读它们来学习算法，并且有很多收获****。这篇文章更多的只是建议****除了阅读这些著作之外还需要做的功课****。此外，对于这类知识讲述（欧几里德）方式的批判西方（尤其是在数学领域）早就有了，早在欧拉和庞加莱的时候，他们俩就极其强调思维的传授，欧拉认为如果不能传授思维，那数学教学是没意义的。而庞加莱本人则更是对数学思维有极大的兴趣和研究（****我前阵子在讨论组上还转载了一篇庞加莱的著名演讲，就是说这个的，参见***[***这里***](http://groups.google.com/group/pongba/browse_frm/thread/3dfc84e0506486cc)*）。我只是在说目前的算法书没有做到思维讲述的层面，因此建议阅读这些书之余应该寻找算法的原始出处，应该寻根究底，多做一些功课，知道算法到底是怎么诞生的，并且我说明了为什么应该知其所以然，有哪些好处（见下文），我还给了几个例子譬如红黑树作者讲红黑树的，g9讲后缀树的，以及Knuth讲heap的。唉，其实挺正统的观点，授人以渔，不管是东方西方都有类似的古老谚语。而我只是从认知科学的角度加了点解释，windstorm称之为“解释文”。而已。可惜被开头的语气搞砸了，算了，既发了也就不改了。*

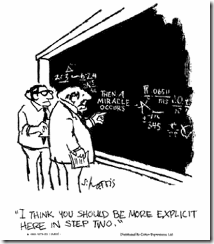
[](http://mindhacks.cn/wp-content/uploads/2009/02/clip-image002.gif)

为什么会这样，其实是有原因的。

我们在思考一个问题的过程中有两种思维形式：

* **联想**：这种思维某种程度上可以说是“混乱”的（虽然从一个更根本的层面上说是有规则的），所谓混乱是指很多时候并不确定联想到的做法最终是否可行，这些联想也许只是基于题目中的某个词语、语法结构、问题的某个切片、一些零星局部的信息。这个过程是试探性的。最后也许有很大一部分被证明是不可行的。很多时候我们解决问题用的都是这种思维，简言之就是首先枚举你关于这个问题能够想到的所有你学过的知识，然后一一往上套看看能否解决手头的问题。这种思维方式受限于人脑联想能力本身的局限性。我在[《跟波利亚学解题》](http://blog.csdn.net/pongba/archive/2008/04/18/2302905.aspx)中就提到了几个例子。联想本身需要记忆提取的线索，所以受到记忆提取线索的制约，如果线索不足，那怎么也联想不起来。而提取线索的建立又取决于当初保存记忆的时候的加工方法（[《找寻逝去的自我》](http://www.douban.com/subject/1315575/)里面有阐述），同时，面对一个问题，你能够从中抽取出来的联想线索又取决于你对问题的认识层度/抽象深度，表浅的线索很可能是无关的，导致无效的联想&试错（[《Psychology of Problem Solving》](http://www.douban.com/subject/2845839/)里面有阐述）。总之，联想这个过程充满了错误的可能。
* **演绎&归纳**：演绎&归纳是另一种思维形式。它们远比联想有根据。其中演绎是严格的，必然的。归纳也是有一定根据的。在面对一个问题的时候，我们有意无意的对问题中的各个条件进行着演绎；譬如福尔摩斯著名的“狗叫”推理——狗+生人=>吠叫 & 昨晚狗没有叫 => 那个人是熟人。就是一个典型的对问题的各个条件进行演绎的推理过程。还有就是通过对一些特殊形式的观察来进行归纳，试图总结问题中的规律。然而，不幸的是，面对复杂的问题，演绎&归纳也并不总是“直奔”问题的解决方案的。人的思维毕竟只能一下子看到有限的几步逻辑结论，一条逻辑演绎路径是否直奔答案，不走到最后往往是不知道的，只要答案还未出现，我们大脑中的逻辑演绎之树的末端就始终隐藏在黑暗之中。而当最终答案出现了之后，我们会发现，这棵演绎之树的很多分支实际上都并不通往答案。所以，虽然演绎&归纳是一种“必然”的推理，然而却并不“必然”引向问题的结论，它也是试错的，只不过比联想要更为靠谱一些。

既然认识到，**人类解决问题的两大思维方式实际上都是有很大的试错成分的**（好听一点叫“探索”），那么就不难意识到，对一个问题的思考过程实际上是相当错综复杂的，而且**充满了无效分支**——在思考的过程中我们也会不断的对分支进行评估，做适当的剪枝——因此当我们找到问题的解之后，**一来思维的漫长繁杂的过程已经在大脑里面淡化得差不多了**，只有那些引向最终结论的过程会被加“高亮”——我们在思考的过程中本就会不断的抛弃无效的思路，只留下最有希望的思路。简而言之就是最后证明没用或者早先我们就不抱希望的一些想法就被从工作记忆中扔掉了。**二来，思考过程是我们的空气和水，而“**[**鱼是最后一个感觉到水的**](http://blog.csdn.net/pongba/archive/2008/01/04/2025830.aspx)**”**，我们感觉不到思维法则本身的存在，我们只是不知不觉运用它。**三来，由于我们的目标是问题的解**，解才是我们为之兴奋和狂喜的东西，而不是求解的过程，**过程只是过程，目的才是目的**。这就像一个寻宝者，在漫长曲折的寻宝历程之后，在找到宝藏的时候，他会对宝藏感到狂喜（记得阿基米德的“找到了！”吗？）而迫不及待地要展示出来，而漫长的思考本身却成了注脚。我们是有目的的动物，目的达到了，其它的就相对不那么重要了。最后，对于传授知识的人，也许还有**其四：感到介绍思维过程是不相干的**，毕竟思维过程并不是算法问题的解，算法问题的解才是算法问题的解。然而不幸的是，忽视到达解的那个过程实际上却变成了舍本逐末。我们看到的是寥寥数行精妙绝伦的算法，然后仰天长叹自己想不出来啊想不出来。为什么想不出来，因为你不知道那短短数行算法背后经历的事怎样漫长的思考过程，如果问题求解是一部侦探小说，**那么算法只是结局而已，而**[**思考过程**](http://blog.csdn.net/pongba/archive/2008/05/07/2412144.aspx)**才是情节**。

[](http://mindhacks.cn/wp-content/uploads/2009/02/clip-image003.gif)

既然如此，也就难怪古往今来算法牛人们算法牛，但却没有几个能真正在讲述的时候还原自己的思维过程的（那个“ 渔”字），手把手的教学生走一遍推理的思路，就可以让学生获得思维过程的训练。金出武雄在[《像外行一样思考，像专家一样实践》](http://www.douban.com/subject/1867455/)中说**写论文应该写得像侦探小说一样**，我很赞同。欧几里德式的介绍，除了提供枯燥的知识之外，并没有提供帮助人获得知识的东西——思维（关于对数学书籍的欧几里德式写法的批评其实也是由来已久了，并且有人呼吁了好几种其它的[教学方法](http://en.wikipedia.org/wiki/Mathematics_education" \l "Methods)）。**从这方面，我们所尊敬的一些“圣经”级书籍在传道授业上还不如侦探小说**，前者是罗列一大堆知识，后者则是阐述获得知识的过程——推理&联想。

然而，我们都是人，人类该有的思维形式，我们难道不是都有吗。既然如此，思维本身又有什么需要一遍遍教的呢？

并非如此。

讲述思维过程而非结果有几个极其重要的价值：

* **内隐化**：思维法则其实也是知识（只不过它是元知识——是帮助我们获得新知识的知识）；是[内隐的记忆](http://en.wikipedia.org/wiki/Implicit_memory)。我们在思考的过程中觉察不到思维法则的作用，它们却在幕后实实在在的左右着我们的思维轨迹。要将思维方法内隐化，[需要不断练习](http://blog.csdn.net/pongba/archive/2008/06/05/2513263.aspx)，就像需要不断练习才能无意识状态下就能骑自行车一样。
* **跨情境运用**：思维法则也是知识记忆，是问题解决策略。既然是记忆，就受到提取线索的制约，这就是为什么当波利亚[告诉你](http://www.douban.com/subject/1456890/)要“注意未知数”之后你还是不能真正在所有需要你“注意未知数”的地方都能提醒自己“注意未知数”。很多时候未知数是很隐蔽的，未知数并不会总是头顶一个大帽子上面写着“我是未知数”。所以很多时候缺乏对这个策略的“提醒”线索，这也是为什么**你学会了在解决数学问题的时候“注意未知数”却不一定能在解决现实生活中的问题中时刻都能“注意你的未知数”**（[《你的灯亮着吗？》](http://www.douban.com/subject/1135754/)整本书的价值便在于此），因为解数学题和解决生活中问题的场景不一样，不同的环境线索，在你大脑中激发的记忆也不一样。就连问题求解中，不同的问题之间的细小差别也可能导致思维轨迹很大的不同，有时你的注意力会被一个无关线索激发的联想吸引开去，忘记如“注意你的未知数”这样的重要法则。而一本从思维角度来讲问题求解的书则可以一遍遍将你置于不同的问题场景下然后在该提醒你的时候提醒你，让你醒悟到“**哦，原来*这个*时候也应该想到*这个*啊。**”，做多了这样的思维演习你就会逐渐从中领悟到某种共性，并将一些思维习惯得到强化，于是终于能够在需要运用某策略的时候能适时的想起来了。
* **对问题解的更多**[**记忆提取线索**](http://blog.csdn.net/pongba/archive/2008/06/05/2513263.aspx)：我们平时学习算法时几乎仅止于“理解”，别人把一个方案放在你面前，你去验证一下，心说“哦，不错，这个的确可以工作”。然后就没了。稍微简单一点的算法还好，复杂一点的对于记忆的负担是很大的，这就是为什么有时候我们看到一个绝妙的解法，这个解法看上去不知道从哪里来的，但经过我们的理解，却发现是对的，我们感叹，真巧妙，结果一些天之后，别人问起这个问题，我们说：“**唉，那是个多么巧妙的算法啊，但是我只记得它巧妙，却不记得它到底是怎样的了。**” 为什么？因为在不知其所以然的情况下，算法只是一堆离散的机械步骤，缺少背后的思想的支撑，这些步骤之间就没有一个**本质层面上的关联**（先知亚里士多德早就指出：学习即联接）。所以就跟背历史书也没多大区别。然而，**知道了算法是怎样一步步被推导出来的，我们就一下拥有了大量的记忆提取线索：对算法发现过程中的任何一个关键步骤（尤其是本质）的回忆都可能使我们能够自己动手推导出剩余的内容**。**譬如**你知道堆(heap)是怎样由朴素的决策树演化而来的，它又是为了解决什么问题的，你即便忘记了具体的细节，也可以自己推导出来。**譬如**你知道KMP算法的本质在于消除回溯，至于如何消除回溯却并不是那么难以推导的，所以即便忘了也可以借助于大脑的逻辑演绎能力再现出来。**譬如**你知道Tarjan算法其实只是从后序遍历经过两个优化调整而来的（其中并査集的使用其实只是优化手段——为了能够迅速判断祖先节点是谁——而非算法本质——当然，算法设计的主要任务本来就是通过问题条件中蕴含的知识来“消除冗余计算”和“避免不必要计算”，所以你也可以说并査集的使用是关乎本质的，只不过，知道了为什么需要引入并査集，就会强烈地感觉到一切是顺理成章的了），那这个出了名的绕人的算法也就不那么难以理解和记忆了。**譬如**你知道[排序的本质](http://blog.csdn.net/pongba/archive/2008/06/13/2544933.aspx)，就能够对什么是最优排序，为什么它是最优排序有深刻的认识。四两拨千斤。

[](http://mindhacks.cn/wp-content/uploads/2009/02/clip-image004.jpg)

* **包含了多得多的知识**：**记一个算法，就只有一个算法**。一个萝卜一个坑。**就好比背99乘法表只能解决乘法问题一样**。而记**背后的思想，却有助于解决一类问题**。思想所处的抽象层面往往比到处都是实现细节的算法本身要低，越是低的抽象层次，越是本质，涵盖范围越是广泛。数学的发展本身就体现了这个过程，抽象代数就是非常好的例子。**算法诞生过程中的思路往往包含了比实际算法更本质得多的知识**，实际算法乃至算法的某个特定语言的实现包含了太多表面的不相干知识，它们会阻碍对本质的理解。
* **重在分析推理，而不是联想**：学了一大通算法和数据结构之后的一个**副作用**就是，看到一个问题之后，脑袋里立即不管三七二十一冒出一堆可能相干的数据结构和算法来。联想是强大的[思维捷径](http://www.douban.com/doulist/127649/)，在任何时候都会[抢占](http://en.wikipedia.org/wiki/Attention#Neural_correlates_of_attention)大脑的[工作记忆](http://en.wikipedia.org/wiki/Working_memory)，由不得你控制——比如我问你“如何寻找区间的最大值”，首先进入你的意识的肯定就是学过的那个算法，甚至算法的实现细节都一一跳了出来，也许最先跳出来的还是算法实现中某个最容易弄错的边界细节，或是某个比较tricky的实现技巧！然而这些其实根本不反映一个算法的本质，结果想来想去总是停留在问题的表层。而另一方面，重在思维的传授则可以让人养成从问题本质入手，逐步分析推理的习惯，而不是直接生搬硬套。**当然，完全不可否认，联想本身也是极其重要的思维方法，甚至可以说是人类思维**[**最重要的**](http://en.wikipedia.org/wiki/Hebbian_learning)**特征**。很多时候我们并不知道问题的本质是什么，就需要靠联想、类比来领路探索。只不过，养成优先从问题的本质入手进行考察的好习惯绝对是有更大的好处的。

那到底什么样的才算是授人以渔的呢？波利亚的《如何解题》绝对算是一本，他的《数学的发现》也值得一看。具体到算法书，那就不是光看text book就足够的了，为了深入理解一个算法的来龙去脉前因后果，从一个算法中领悟尽量深刻的东西，则需要做到三件事情：

* **寻找该算法的原始出处**：TAOCP作为一个资料库是绝对优秀的，基础的算法只要你能想到的，几乎都可以在上面找到原始出处。查到原始出处之后（譬如一篇paper），就可以去网上搜来看了。因为最初的作者往往对一个方案的诞生过程最为了解。比如经典数据结构中的红黑树是出了名的令人费解的结构之一，但它的作者Sedgewick[一张PPT，给你讲得通通透透](http://groups.google.com/group/pongba/browse_thread/thread/3513a21065faba68)，比算法导论上的讲法强上数倍。
* **原始的出处其实也未必就都推心置腹地和你讲得那么到位**：前面说过，算法设计出来了之后人们几乎是不会去回顾整个的思维过程细节的，只把直指目标的那些东西写出来。结果就又是一篇欧几里德式的文章了。于是你就迷失在一大堆“定义”、“引理”、“定理”之中了。这种文章看上去整个写得井井有条，其实是把发明的过程整个给颠倒过来了，我一直就想，如果作者们能够将整个的思路过程写出来，哪怕文字多上十倍，我也绝对会比看那一堆定义定理要容易理解得多。话说回来，怎么办？可以再去网上找找，牛人讲得[未必比经典教材上的差](http://blog.csdn.net/g9yuayon/archive/2008/06/21/2574781.aspx)。那倘若实在找不出好的介绍呢，就只能自己**揣摩**了。**揣摩的重要性，是怎么说都不为过的**。揣摩的**一些指导性的问题有**：为什么要这样（为什么这是好的）？为什么不是那样（有其它做法吗？有更好的做法吗？）？这样做是最好的吗？（为什么？能证明吗？）这个做法跟其它的什么做法有本质联系吗？*这个*跟*这个*的区别是什么？**问题的本质是什么**？**这个做法的本质又是什么**？到底本质上是什么东西导致了这个做法如此..？与这个问题类似的还有其它问题吗？（同样或类似的做法也适用吗？）等等。
* **不仅学习别人的思路，整理自己的思路也是极其重要的**：详见[《跟波利亚学解题》](http://blog.csdn.net/pongba/archive/2008/04/18/2302905.aspx)的“4. 一个好习惯”和“7. 总结的意义”。

前一段时间我们讨论组上有不少例子，见[这里](http://groups.google.com/group/pongba/web/toplang-problemsolvingseries)，或[这里](http://del.icio.us/pongbablog/%E8%A7%A3%E9%A2%98)。

[**知其所以然（续）**](http://mindhacks.cn/2010/11/14/the-importance-of-knowing-why-part2/)

**BY**

[**刘未鹏**](http://mindhacks.cn/author/pongba/)

 – NOVEMBER 14, 2010**POSTED IN:**[学习方法](http://mindhacks.cn/topics/learning-method/), [算法](http://mindhacks.cn/topics/algorithms/)

查了一下，上篇[知其所以然（以学习算法为例）](http://mindhacks.cn/2008/07/07/the-importance-of-knowing-why/)是08年7月写的，现在已经是10年11月，过去了两年零4个月，这说明了三件事情：1，一个问题其实你可以一直放在脑子里面，利用[暗时间](http://mindhacks.cn/2009/12/20/dark-time/)对其软泡硬磨，时间足够久你总会有一点新的感悟，问题其实就像那句老话说的那样，不怕贼偷就怕贼惦记，聚精会神的思考一天，也许比不上惦记一个星期（据说[数学家庞加莱就特别会惦记问题](http://blog.csdn.net/pongba/archive/2008/07/24/2705151.aspx)）。2，事实上，当你感觉懂了的时候，你至少得反问自己一句，真的懂了吗？当你确信自己真的懂了的时候，你至少得讲给别人听，别人听懂了吗？考察你自己是否真懂了的一个很好的依据是，你是否有一种“哦，原来是这样啊，这下再也不可能忘记了”的感觉。3，我其实没有忘记这个博客。如我之前说的，[记录只是学习和思考的副作用](http://mindhacks.cn/2009/02/15/why-you-should-start-blogging-now/)，只要还在学习和思考，就必然会有新的记录。

我有一个习惯，看定理必看证明。一个你不明白其证明的定理在我看来比不知道这个定理还要糟糕，因它给你造成一种懂了的错觉。**在没有明白背后的证明之前，任何一个定理对你来说都是等价的——等价于背乘法口诀**（只不过有的长一点有的短一点）。一个原本美妙的定理，把其证明扔掉就是真正的买椟还珠，暴殄天物。

从现实意义来说，**去理解一个定理的证明会带来巨大的好处，首当其冲的好处就是你很难再忘掉它**。这一点其实很容易解释——在理解一个定理的证明之前，定理对你而言是一堆没有内在联系的词句，而在理解了证明之后，定理就**归约为**证明它所需的条件加上逻辑，“逻辑”本来就存在于你的大脑里面，而证明的过程中除了公理和用到的常见定理（往往没几条）之外，宽泛地说，需要你去记的，一般来说也只有一个或两个关键的insights，也就是我们常说的证明中的神来之笔，比如几何证明里面的某条看上去莫名其妙的辅助线，一旦你知道了这条辅助线，那么整个证明就毫无难处，那么该定理的信息量便直接缩减为一条辅助线的信息量；虽然看上去这一步信息并没有缩减多少，但是如果你考虑到类似的辅助线不仅会用在这个特定的定理上，往往会在很多地方用到。很多关键的证明手法是通用的。那么其实你就是把所有以这个辅助线为关键证明手法的定理的集合的信息量归约为了这条辅助线。如果你进而甚至能够理解了作这条辅助线的思想精髓，那就更牛逼了，因为解决问题的思路更具有一般性，理解了寻找正确的辅助线的思路，你就根本不需要去记得某条特定辅助线的作法，你就把所有以作一条或几条辅助线为证明核心的定理的集合的信息量归约为了这个“寻找辅助线的思路”。

**这是一个树状的知识结构，越往上层走，需要记忆的节点就越少**。所谓触类旁通者，其实便是因为他擅长去理解解法背后的更具一般性的东西。所以我还有一个习惯，就是看到美妙的证明和解法总是会去一遍又一遍的去反复揣摩，试图理解想出这个证明的人到底是怎么想出来的，有没有什么一般性的方法可循，很多时候，在这样揣摩的过程中，你会理解到更深刻的东西，对问题性质更深刻的认识，对解决问题的思路更深刻的认识，这些认识不仅对于你理解当前这个定理或问题有极大的帮助，同时也有助于你解决以后会遇到的表面不同但本质一样的问题。

与看定理必看证明类似，看一个问题的解法，必然要看解法所诞生的过程，背后是否隐藏着更具一般性的解决问题的思路和原则。否则一个解法就只是一个问题的解法，跟背口诀一样。即便记住了也无法推广，即便当时记住了也容易遗忘。

举个经典的例子：每本算法书都会讲动态规划，每本讲动态规划的书都会讲背包问题，每次讲背包问题都会讲可重复背包和01背包，我们就拿《Algorithms》这本[还算不错](http://book.douban.com/review/1325850/)的算法书对背包问题的讲解来说吧，重复背包问题的递归公式是这样的：

K(W) = max { K(W-Wi) + Vi : Wi <= W }

这个公式的理解倒是很简单：为了把问题降阶，我们在最终的最优解里面去掉一个元素，对这个元素的可能性进行讨论，它必然是任何Vi之一（前提是Wi <= W，否则就装不下），而在去掉这个元素之后，剩下的元素肯定构成问题 K(W-Wi) 的最优解，于是递归关系出现了。

此外也可以这样来理解：要拿一组最优元素，那么总得开始一个个拿吧，对第一个拿的元素进行讨论，而问题的最优解等于讨论的各个分支的最优解中的最优者；如果拿掉Vi之后，剩下来要怎么拿才能最优呢？这就是一个 K(W-Wi) 的问题了。

01背包问题就大不一样了——每个物品都只有一件，拿掉之后就不能再拿了。我们不妨看看重复背包问题的解法是不是能用到01背包上呢？还是讨论第一个拿的元素，设被拿掉的是第i个元素，问题就归结为把剩下的物品（注意，可拿的物品少了一件）最优地装入容量为 W-Wi 的包里，所以，问题的参数便变成了两个，一个是背包剩余容量 W-Wi，另一个是剩余可拿的物品集合 S\{i} （表示去掉i之后的子集），显而易见第二个参数是物品集合的各种可能的子集，那么其可能性个数就是 2^n ，这就导致子问题的个数是 2^n， 由于要依次计算每个子问题，那么算法复杂度显然也是 2^n ，是不可接受的。

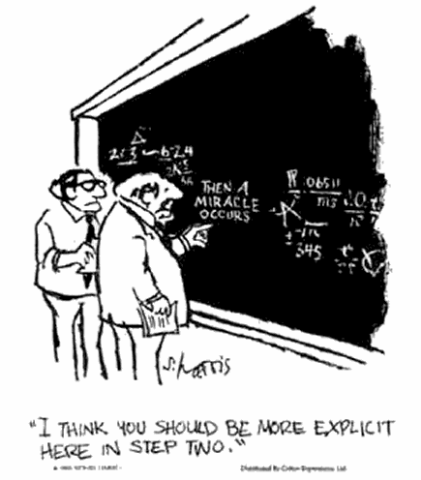
那么，《Algorithms》上又是怎么来讲解01背包问题的解法的呢？以下是原文：

Our earlier subproblems now become completely useless. We must therefore refine our concept of a subproblem to carry additional information about the items being used. We add a second parameter, 0 <= j <= n: K(W, j) = maximum value achievable using a knapsack of capacity w and items 1..j: The answer we seek is K(W, n).

首先作者说了，之前重复背包问题的解法在这里完全废掉了，所以我们必须重新定义子问题，并且子问题的条件必须要包含目前拿剩下的物品。以上这些都还不错，关键是接下来就让人吐血了。作者接着说道，**我们**给子问题加上一个新的参数j…

凭什么啊？

还是让我们回顾一下[这样一幅经典的漫画](http://images.google.com/images?hl=en&source=imghp&biw=1024&bih=620&q=then+a+miracle+occurs&gbv=2&aq=f&aqi=g3g-m6&aql=&oq=&gs_rfai=)吧：



“我们给子问题加上一个参数j”，这就像你在看数学证明时看到无比邪恶的“**我们考虑**…“一样，一看到这样的句子，你就知道，这个问题的证明远远不像看上去那么简单，之所以你一路看下去理解上全无困难，那完全是因为作者直接把最重要的一个insight告诉你了，举个很简单的例子，证明素数无最大，谁都会第一时间想到去反证：假设存在一个最大的素数P，那么找到比P大的素数就是证明中最关键的一步，怎么找的？一般书上是不会说的，你会看到书上这样说：假设P是最大的素数，那么我们考虑P’ = 小于等于P的所有素数的乘积+1。那么P’一来显然大于P，二来不能被小于它的所有素数整除，那么P’就成了大于P的素数。

如果你经常注意反证法，你会发现一个有趣的现象，反证法里面经常会有这样一句“我们考虑”，而“我们考虑”后面几乎肯定接着一个天外飞仙一般的insight。素数无最大这个古老的证明里面的“我们考虑”尚算是比较有迹可循的（我们想要构造一个更大的素数，而素数的等价定义就是“不能被小于它的所有素数整除，为了达到这个目的，构造的方法就较明显了）。但是有非常非常多的证明，其中关键的一步就跟嗑药磕出来做梦做出来走路跌跟头跌出来的一样（不信去翻一翻《[Proofs from THE Book](http://www.amazon.com/Proofs-BOOK-Martin-Aigner/dp/3540404600)》），让你完全不知道他怎么想到的。

话说回来，虽然有很多数学证明的关键步骤是很难逆向工程的（因为很多时候想出那个关键步骤的本人其实也是尝试了各种方法，撞了无数堵墙，在寻求证法的尝试空间中作了N次回溯才“妙手偶得”，与其说是妙手偶得，不如说是绞尽脑汁），但并非全无章法可循，否则陶哲轩也不会写出《[Solving Mathematical Problems](http://www.amazon.com/Solving-Mathematical-Problems-Personal-Perspective/dp/0199205604)》这样的著作来，而求解问题也就成了真正的Black Art了。

算法的解法则比精妙的数学证明稍加更容易逆向工程一点。只要你有耐心仔细地去琢磨算法的关键步骤和本质，总能从中窥探到一些更general的思想和思路来。

此外，很多经典问题，算法书上的讲法虽然时时令我们失望，但如果去网上一搜，则通常会发现更优秀的解释来。比如[背包问题就是如此](http://www.google.com/webhp?hl=en#sclient=psy&hl=en&site=webhp&source=hp&q=%E8%83%8C%E5%8C%85%E9%97%AE%E9%A2%98+%E8%A7%A3%E7%A9%BA%E9%97%B4&aq=f&aqi=&aql=&oq=&gs_rfai=&pbx=1&fp=ab43752bbefd1914)。

简单地说，如果你对于每个问题都能真正弄清以下这几个问题的答案，那么可以肯定的是，你的理解，记忆，以及学习的效率都会得到质的提高：

* 为什么这种解法是对的？
* 为什么那种解法是错的？
* 为什么这种解法不是最优的？
* 证明为什么没有更优的解法。

回到人民群众喜闻乐见的经典例子：背包问题。为什么01背包问题的正确（高效）算法是正确（高效）的。表面的解释是，因为01背包问题的子问题定义是 K(W, j)，其两个维度相乘的可能性一共有nW种，也就是说一共要计算nW个子问题，而计算每个子问题的复杂度是O(1)的。

但是如果仅仅满足于这样的解释，可以说是隔靴搔痒，并没有触及到本质。算法本质上可以看做是在一个解空间当中的搜索问题，所以要分析一个算法的好坏，首先弄清它的解空间的结构，然后分析它是怎么来探索这个解空间的。

弄清解空间的是第一步，例如排序算法，其解空间可以看做是所有可能的下标排列组合，其中有且仅有一个排列是正确的排序排列（简单起见假设元素各不相同）。那么一个算法在探索这个解空间方面的行为就决定了它的效率高低，最简单的，如果一个算法每次只能检查解空间中的一个点，那么这个算法的复杂度就是解空间的大小。对排序算法而言也就是n!。从这个角度来看，我们就会很容易的发现，所有基于比较的排序算法，其复杂度为什么是以O(nlogn)为下界的，因为一次比较操作最多有两个结果，a>b或a<b，既然只有两种结果，那么最多只能将解空间进行2分，如果每次都能完美的2分，那么找到那个唯一点最终需要的步骤就是log(n!) = O(nlogn)。如此就不难理解[什么基于比较的排序算法的复杂度最好不过如此了](http://mindhacks.cn/2008/06/13/why-is-quicksort-so-quick/)。

回到01背包问题，01背包问题的解空间其实也是类似的。一次选取就是一个01数组，其中每个元素代表其所对应的物品要不要选取。很显然，这个解空间的大小是2^n。在01背包的算法里面，每当我们解出K(W, j)（需要O(W)次计算）之后，解空间就会被折半（排除掉1/2的可能性），一共如此做n次，就能得到最终解。由于每次折半的代价是O(W)，便不难理解为什么算法复杂度是O(nW)了。

那么，为什么每次计算出K(W,j)就能使解空间折半呢？那就需要来看看这个算法是如何探索解空间的，算法探索解空间的方式在其递归公式里面：

K(W, j) = max { K(W, j-1), K(W-Wj, j – 1)  + Vj }

也就是说，首先看你要不要选取第一个物品，有两种可能性（两个分支），每个分支都是一个更低阶的子问题，即在其中的任意一个分支下都要决定要不要选取第二个物品（又是两个分支），如此下递归去，可以构建出一棵有2^n方个叶子节点的树，每条从根结点到叶子节点的路径“01..101”就对应一个解，其中每个分叉代表“选”或“不选”当前的物品。

建立在对这个解空间的理解上，我们再来看为什么01背包问题的正确解法能做到O(nW)。（首先你最好将这棵树画在纸上，其中每个节点都是一个子问题K(W,j)，每条分叉都是0或1。）当我们计算出所有的K(W, 1)（需要O(W)次操作）之后，我们容易注意到，所有离叶子节点的距离为1的内部节点K(W, 2)到叶子节点的两个分支都必然只能取其一了，也就是说，有一半的叶子节点被排除掉了（对解空间折半）。当我们进而计算出K(W,2)之后，同样的道理，我们容易看到，到叶子节点距离为2的内部节点的两个分支也只能取其一了，这就进而再次将解空间折半。由于每次折半需要O(W)的复杂度，所以就不难理解算法的总复杂度为O(nW)了。另一种理解的方法是，当我们计算出K(W,j)的时候，从内部节点K(W,j)到根节点的唯一路径便确定了。经过O(nW)次计算，从根节点到那个唯一解（叶子节点）的路径便完全确定了。

**知道怎么做是从正确（高效）解法得到的，而知道为什么必须得那样做则往往是从错误（低效）的解法当中得到的。**

然而遗憾的是，绝大多数算法书或教程都只顾一上来就告诉你正确的做法是什么，对于一些常见的错误解法，或者常见的低效解法，却根本不加分析。经验告诉我们，理解错误的做法为什么错误同样甚至更为重要，往往是在理解了错误的解法为什么错误之后，我们才能深刻的体会到为什么正确的解法是如此正确。

还是拿经典的背包问题来作例子，你几乎看不到哪本书会告诉你一个典型的低效解法为什么低效的深刻原因。我们都知道动态规划的核心在于子问题的划分，同样的问题，不同的划分办法得到的复杂度完全不一样。前面已经提到了，重复背包问题的思路在01背包问题上会带来指数级的复杂度，但是为什么呢？如果你满足于说：因为如果拿重复背包问题的思路来解01背包问题，那么子问题定义的第二个维度（物品的子集）（见前文）是指数级的，那么要计算所有子问题，当然是指数级的。那么你只是看到这个问题的表象。

如果从对解空间的探索方式来说，可以容易看出这个现象的本质，我们回顾一下01背包问题的正确（高效）算法：

K(W, j) = max { K(W, j-1), K(W-Wj, j – 1)  + Vj }

这个算法讨论的是两种情况，“要”或者“不要”选取第j个物品，这两种情况所对应的解空间是完全不交的，这就有效地将解空间划分为了不重复的两个部分。

而再来看利用重复背包问题思路的解法：

K(W, S) = max { K(W-Wi, S\{i}) + Vi : Wi <= W }

这里讨论的是首先拿掉哪一个物品，还是那句话，讨论的每一个分支都对应了算法对解空间的一个切分，我们容易看出，在“先拿物品i”和”先拿物品j“这两个分支里面，存在大量的重复，因为先拿物品i再拿j，和先拿物品j再拿i对应的是完全一样的一组选取。事实上，如果你将这个递归公式画成树状结构，会发现有n!个叶子节点。n!是什么概念？01背包问题的解空间大小本质上就只有2^n次方，穷举也不过O(2^n)的复杂度，结果这样一切分却变成了n!，可见这种对解空间的切分方法的冗余度是多么高了。你不妨看看，每一次计算K(W, S)子问题能对解空间排查多少呢？是否能像前面正确的算法那样，每次都能有效排查一半情况？理解了这一点之后，我们便注意到在划分解空间，也就是定义子问题的时候的一个原则，就是在建立递归公式的时候，尽量将解空间进行不交的切分。同时我们便有了趁手的工具去分析一个动态规划的解法的效率。

最后再举一个例子：算法书上几乎必讲的霍夫曼树。你所看的算法书在讲霍夫曼树的时候给了证明吗？讲过霍夫曼树的[历史八卦](http://en.wikipedia.org/wiki/Huffman_tree#History)吗？也许你看了霍夫曼树的构造方法之后觉得：“哦，这样啊，显然”。但是你可曾想到，在最优编码这个问题上，连[香农本人之前给出的解法](http://en.wikipedia.org/wiki/Shannon-Fano_coding)都只是suboptimal的，而且霍夫曼本人在得到这个算法之前也是绞尽脑汁几近放弃。**如果你10分钟就“理解”了，那么百分之百只是背了课文而已。**

[**知其所以然（三）：为什么算法这么难？**](http://mindhacks.cn/2011/07/10/the-importance-of-knowing-why-part3/)

**BY**

[**刘未鹏**](http://mindhacks.cn/author/pongba/)

 – JULY 10, 2011**POSTED IN:**[算法](http://mindhacks.cn/topics/algorithms/)

不知不觉《知其所以然》系列竟然也写到第三篇了，虽然前面两篇也说了不少，但是总觉得还有东西没有说“透”，或者说没有说“好”。所以这篇试图从不同的角度用更好的例子来继续深入阐述。（感谢[silwile](http://www.douban.com/people/silwile/)对本文的review和意见）

广大码农同学们大多都有个共识，认为算法是个硬骨头，很难啃，悲剧的是啃完了还未必有用——除了面试的时候。实际工程中一般都是用现成的模块，一般只需了解算法的目的和时空复杂度即可。

不过话说回来，面试的时候面算法，包括面项目中几乎不大可能用到的算法，其实并不能说是毫无道理的。**算法往往是对学习和理解能力的一块试金石**，难的都能掌握，往往容易的事情不在话下。志于高者得于中。反之则不成立。另一方面，虽说教科书算法大多数都是那些即便用到也是直接拿模块用的，但不幸的是，我们这群搬砖头的有时候还非得做些发明家的事情：要么是得把算法当白盒加以改进以满足手头的特定需求；要么干脆就是要发明轮子。所以，虽说面试的算法本身未必用得到，但熟悉各种算法的人通常更可能熟悉算法的思想，从而更可能具备这里说的两种能力。

那么，为什么说算法很难呢？这个问题只有两种可能的原因：

1. 算法本身就很难。也就是说，算法这个东西对于人类的大脑来说本身就是个困难的事儿。
2. 讲得太烂。

下面会说明，算法之所以被绝大多数人认为很难，以上两个原因兼具。

我们说算法难的时候，有两种情况：**一种是学算法难。第二种是设计算法难**。对于前者，大多数人（至少我当年如此）学习算法几乎是在背算法，就跟背菜谱似的（“Cookbook”是深受广大码农喜爱的一类书），然而算法和菜谱的区别在于，算法包含的细节复杂度是菜谱的无数倍，算法的问题描述千变万化，逻辑过程百转千回，往往看得人愁肠百结，而相较之下任何菜谱涉及到的基本元素也就那么些（所以程序员肯定都具有成为好厨师的潜力:D）注意，即便你看了算法的证明，某种程度上还是“背”（为什么这么说，后面会详述）。我自己遇到新算法基本是会看证明的，但是发现没多久还是会忘掉，这是死记硬背的标准症状。如果你也啃过算法书，我相信很大可能性你会有同感：*为什么当时明明懂了，但没多久就忘掉了呢？为什么当时明明非常理解其证明，但没过多久想要自己去证明时却发现怎么都没法补上证明中缺失的一环呢？*

初中学习几何证明的时候，你会不会傻到去背一个定理的证明？不会。你只会背结论。为什么？一方面，因为证明过程包含大量的细节。另一方面，证明的过程环环相扣，往往只需要注意其中关键的一两步，便能够自行推导出来。算法逻辑描述就好比定理，算法的证明的过程就好比定理的证明过程。但不幸的是，与数学里面大量简洁的基本结论不同，算法这个“结论”可不是那么好背的，许多时候，**算法本身的逻辑就几乎包含了与其证明过程等同的信息量**，甚至算法逻辑本身就是证明过程（随便翻开一本经典的算法书，看几个经典的教科书算法，你会发现算法逻辑和算法证明的联系有多紧密）。于是我们又回到刚才那个问题：你会去背数学证明么？**既然没人会傻到去背整个证明，又为什么要生硬地去背算法呢**？

那么，不背就不背，去理解算法的证明如何？理解了算法的证明过程，便更有可能记住算法的逻辑细节，理解记忆嘛。然而，仍然不幸的是，绝大多数算法书在这方面做的实在糟糕，证明倒是给全了，逻辑也倒是挺严谨的，可是似乎没有作者能真正还原算法发明者本身如何得到算法以及算法证明的思维过程，按理说，证明的过程应该反映了这个思维过程，但是在下文关于霍夫曼编码的例子中你会看到，其实饱受赞誉的[CLRS](http://www.amazon.com/Introduction-Algorithms-International-Thomas-Cormen/dp/0262533057)和[《Algorithms》](http://www.amazon.com/Algorithms-Sanjoy-Dasgupta/dp/0073523402)*不仅没能还原这个过程，反而掩盖了这个过程*。

必须说明的是，没有哪位作者是故意这样做的，但任何人在讲解一个自己已经理解了的东西的时候，往往会无意识地对自己的讲解进行“线性化”，例如证明题，如果你回忆一下高中做平面几何证明题的经历，就会意识到，其实证明的过程是一个充满了试错，联想，反推，特例，修改问题条件，穷举等等一干“非线性”思维的，混乱不堪的过程，而并不像写在课本上那样——引理1，引理2，定理1，定理2，一口气直到最终结论。这样的证明过程也许容易理解，但绝对不容易记忆。过几天你就会忘记其中一个或几个引理，其中的一步或几步关键的手法，然后当你想要回过头来自己试着去证明的时候，就会发现卡在某个关键的地方，为什么会这样？因为证明当中并没有告诉你为什么作者当时会想到证明算法需要那么一个引理或手法，所以，**虽说看完证明之后，对算法这个结论而言你是知其所以然了，但对于算法的证明过程你却还没知其所以然**。在我们大脑的记忆系统当中，**新的知识必须要和既有的知识建立联系，才容易被回忆起来**（[《如何有效地学习与记忆》](http://mindhacks.cn/2009/03/28/effective-learning-and-memorization/)）**，联系越多，越容易回忆，而一个天外飞仙似地引理，和我们既有的知识没有半毛钱联系，没娘的孩子没人疼，自然容易被遗忘。**（为什么还原思维过程如此困难呢？我曾经在[知其所以然（一）](http://mindhacks.cn/2008/07/07/the-importance-of-knowing-why/)里详述）

正因为绝大多数算法书上悲剧的算法证明过程，很多人发现证明本身也不好记，于是宁可选择直接记结论。当年我在数学系，考试会考证明过程，但似乎计算机系的考试考算法证明过程就是荒谬的？作为“工程”性质的程序设计，似乎更注重使用和结果。但是如果是你需要在项目中自己设计一个算法呢？这种时候最起码需要做的就是证明算法的正确性吧。我们面试的时候往往都会遇到一些算法设计问题，我总是会让应聘者去证明算法的正确性，因为即便是**一个“看上去”正确的算法，真正需要证明起来往往发现并不是那么容易**。

所以说，绝大多数算法书在作为培养算法设计者的角度来说是失败的，比数学教育更失败。大多数人学完了初中平面几何都会做证明题（数学书不会要求你记住几何所有的定理），但很多人看完了一本算法书还是一团浆糊，不会证明一些起码的算法，我们背了一坨又一坨结论，非但这些结论许多根本用不上，就连用上的那些也不会证明。为什么会出现这样的差异？因为数学教育的理想目的是为了让你成为能够发现新定理的科学家，而码农系的算法教育的目的却更现实，是为了让你成为能够使用算法做事情的工程师。然而，事情真的如此简单么？如果真是这样的话干脆连算法结论都不要背了，只要知道算法做的是什么事情，时空复杂度各是多少即可。

如果说以上提到的算法难度（讲解和记忆的难度）属于[Accidental Complexity](http://en.wikipedia.org/wiki/Accidental_complexity)的话，算法的另一个难处便是Essential Complexity了：算法设计。还是拿数学证明来类比（如果你看过《Introduction to Algorithms：A Creative Approach》就知道算法和数学证明是多么类似。），与单单只需证明相比，**设计算法的难处在于，定理和证明都需要你去探索，尤其是前者**——你需要去自行发现关键的那（几）个定理，跟证明已知结论相比（已经确定知道结论是正确的了，你只需要用逻辑来连接结论和条件），**这件事情的复杂度往往又难上一个数量级。**

一个有趣的事实是，**算法的探索过程往往蕴含算法的证明过程**，理想的算法书应该通过还原算法的探索过程，从而让读者不仅能够自行推导出证明过程，同时还能够具备探索新算法的能力。之所以这么说，皆因为我是个懒人，懒人总梦想学点东西能够实现以下两个目的：

1. **一劳永逸**：程序员都知道“[一次编写到处运行](http://en.wikipedia.org/wiki/Write_once,_run_anywhere)”的好处，多省事啊。学了就忘，忘了又得学，翻来覆去浪费生命。为什么不能看了一遍就再也不会忘掉呢？到底是教的不好，还是学得不好？
2. **事半功倍**：事实上，程序员不仅讲究一次编写到处运行，更讲究“一次编写到处使用”（也就是俗称的“复用”）。如果学一个算法所得到的经验可以到处使用，学一当十，推而广之，时间的利用效率便会大大提高。究竟怎样学习，才能够使得经验的外推（extrapolate）效率达到最大呢？

想要做到这两点就必须**尽量从知识树的“根节点”入手**，虽然这是一个美梦，例如数学界寻找“根节点”的[美梦由来已久](http://mindhacks.cn/2008/04/18/learning-from-polya/)（《跟波利亚学解题》的“一点历史”小节），但哥德尔一个证明就[让美梦成了泡影](http://mindhacks.cn/2006/10/15/cantor-godel-turing-an-eternal-golden-diagonal/)（《永恒的金色对角线》））；但是，这并不阻止我们去寻找更高层的节点——[更具普适性的解题原则和方法](http://mindhacks.cn/2008/04/18/learning-from-polya/)。所以，理想的算法书或者算法讲解应该是从最具一般性的思维法则开始，顺理成章地推导出算法，这个过程应该尽量还原一个”普通人“思考的过程，而不是让人看了之后觉得”这怎么可能想到呢？

以本文上篇提到的霍夫曼编码为例，第一遍看霍夫曼编码的时候是在本科，只看了算法描述，觉得挺直观的，过了两年，忘了，因为不知道为什么要把两个节点的频率加在一起看做单个节点——一件事情不知道“为什么”就会记不牢，知道了“为什么”的话便给这件事情提供了必然性。不知道“为什么”这件事情便可此可彼，**我们的大脑对于可此可彼的事情经常会弄混，它更容易记住有理有据的事情**（*从信息论的角度来说，一件必然的事情概率为1，信息量为0，而一件可此可彼的事情信息量则是大于0的*）。第二遍看是在工作之后，终于知道要看证明了，拿出著名的《Algorithms》来看，边看边点头，觉得讲得真好，一看就理解了为什么要那样来构造最优编码树。可是没多久，又给忘了！这次忘了倒不是忘了要把两个节点的频率加起来算一个，而是忘了为什么要这么做，因为当时没有弄清霍夫曼为什么能够想到为什么应该那样来构造最优编码树。结果*只知其一不知其二*。

必须说明的是，如果只关心算法的结论（即算法逻辑），那么理解算法的证明就够了，光背算法逻辑难记住，理解了证明会容易记忆得多。但如果也想不忘算法的证明，那么不仅要理解证明，还要理解证明背后的思维，也就是*为什么背后的为什么*。后者一般很难在书和资料上找到，唯有自己多加揣摩。为什么要费这个神？只要不会忘记结论不就结了吗？取决于你想做什么，如果你想真正弄清算法设计背后的思想，不去揣摩算法原作者是怎么想出来的是不行的。

回到霍夫曼编码问题，我们首先看一看《Algorithms》上是怎么讲的：

首先它给出了一棵编码树的cost function：

cost of tree = Σ freq(i) \* depth(i)

这个cost function很直白，就是把编码的定义复述了一遍。但是接下来就说了：

*There is another way to write this cost function that is very helpful.* Although we are only given frequencies for the leaves, we can define the frequency of any internal node to be the sum of the frequencies of its descendant leaves; this is, after all, the number of times the internal node is visited during encoding or decoding…

接着就按照这个思路把cost function转换了一下：

The cost of a tree is the sum of the frequencies of all leaves and internal nodes, except the root.

然后就开始得出算法逻辑了：

The *first formulation* of the cost function tells us that the *two symbols with the smallest frequencies must be at the bottom of the optimal tree*, as children of the lowest internal node (this internal node has two children since the tree is full). Otherwise, swapping these two symbols with whatever is lowest in the tree would improve the encoding.

This suggests that we start constructing the tree greedily: find the two symbols with the smallest frequencies, say i and j, and make them children of a new node, which then has frequency fi + fj. To keep the notation simple, let’s just assume these are f1 and f2. By the *second formulation* of the cost function, any tree in which f1 and f2 are sibling-leaves has cost f1 + f2 plus the cost for a tree with n – 1 leaves of frequencies (f1 + f2), f3, f4, .., fn. *The latter problem is just a smaller version of the one we started with*.

读到这里我想大多数人有两种反应：

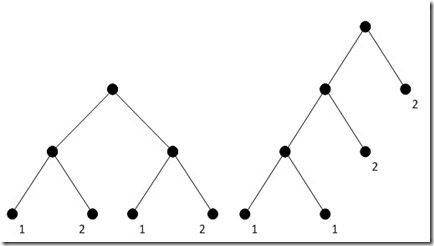
1. 觉得理所当然。
2. 觉得恍然大悟。

因为我当时也是这么觉得的。可是后来当我发现自己无法从头证明一遍的时候，我知道肯定是理解的不够深刻。如果理解的够深刻，那么基本上是不会忘掉的。

如果看完霍夫曼编码这样一个简短证明你觉得顺理成章，一切都挺显然，*那就坏了*，即便是看上去最基本的性质也往往实际上没那么显然。“逢山开路，遇水架桥”在我们今天看来是无比显然的事实，但是试想在没有桥的远古时代，一个原始人走到一条湍急的河流前，他会怎么想，他又能有什么法子呢？这是个他从来没有遇见过的问题。如果后来有一天，他路过另外一条小溪，看到小溪上有一截被闪电劈断的枯树，于是他踏着树干走过了小溪，并意识到“树横过河面”可以达到“过河”这个目的，这就将条件和目的建立了直接的联系（事实上，是自然界展示了这个联系，我们的原始人只是记住了这个联系）。后来他又路过那条河流，他寻思如何达到“过河”这个目的的时候，忽然意识到在他的记忆中已经遇到过需要达成同样目的的时候了，那个时候的条件是“树横过河面”，于是问题便归结为如何满足这个“树横过河面”的条件，而后一个问题就简单多了。（事实上波利亚在他的著作《How to Solve it》中举的正是这么个例子）

为什么那么多的算法书，就看不到有一本讲得好的？因为我们求解问题过程中的思维步骤太容易被自己当作“显然”的了，但除了我们天生就会的认知模式（联系，类比），没有什么是应该觉得显然的，试错是我们天生就会的思维法则么？是的，但是可供尝试的方案究竟又是怎么来的呢？就拿上面的例子来说，一个从没有见过枯树干架在小溪上的原始人，怎么知道用树架桥是一种可选的方案呢？俗话说巧妇难为无米之炊啊。我们大脑的神经系统会的是将目的和条件联系起来，第一次原始人遇到小溪过不去，大脑中留下了一个未实现的目的，后来见到小溪上的树干，忽然意识到树干是实现这个目的的条件，两者便联系起来了，因此问题就规约为如何架树干了。

回到《Algorithms》中的证明上，这个看似简洁明了的证明其实有几处非常不显然的地方，甚至不严谨的地方，这些地方也正是你过段时间之后试图自己证明的话会发现卡住的地方：

1. 作者轻飘飘地就给出了cost function的另外一种关键的描述，而对于如何发现这种描述却只是一语带过："*There is another way to write this cost function that is very helpful..*we can define the frequency of any internal node to be the sum of the frequencies of its descendant leaves“这其实就是我常常痛恨的“*我们考虑*…”，这里作者其实就是在说”让*我们考虑*下面这样一种*奇妙*的转换“，可是怎么来的却不说。但必须承认，《Algorithms》的作者还是算厚道的，因为后面他又稍微解释了一下：“this is, after all, the number of times the internal node is visited during encoding or decoding…”这个解释就有点让人恍然大悟了，但是千万别忘了，这种恍然大悟是一种错觉，你还是没明白为什么他会想到这一点。这就像是作者对你说“*仔细观察*问题条件，我们*容易发现*这样一种奇妙的性质..”，怎么个“仔细”法？凭什么我自己“观察”半天就是发现不了呢？霍夫曼本人难道也是死死盯着问题[“观察”了一学期](http://en.wikipedia.org/wiki/Huffman_tree" \l "History)然后就“发现”了么？我们有理由相信霍夫曼肯定尝试了各种各样的方法，作出了各种各样的努力，否则当年Shannon都没搞定的这个问题花了他一学期，难道他在这个学期里面大脑就一片空白（或者所有的尝试全都是完全不相干的徒劳），然后到学期末尾忽然“灵光一现”吗？
2. 如果“仔细观察”:)，我们会发现两个cost function表达中frequency的概念有微妙的差异，在第一个cost function中，只有叶子节点有frequency，而这个frequency必须和叶子节点的深度相乘。而在第二个cost function中，内部节点也具有了frequency，可是所有节点的“frequency”忽然全都不跟深度相乘了。frequency的不同含义令人困惑。
3. 作者提到：第一个cost function告诉我们频率最低的两个节点必然处于最优编码树的底端，作为最低内部节点的两个子节点。这是一个不严谨的说法，从前文给出的条件和性质，只能推导出编码树的最底层必然能找到频率最低的两个节点，但*它们未必一定要是兄弟节点*，如果树的最底层不止能容纳两个节点的话它们就可以有不同的父节点。“我们不妨考虑”这样一个例子：对A,B,C,D四个字母进行编码，假设它们的频率分别是1， 1， 2， 2。这个时候我们可以构造如下图所示的两棵树，两棵树的cost都是12，都是最优的。但其中一棵树中，两个频率最低的节点并非兄弟。   
   [](http://mindhacks.cn/wp-content/uploads/2011/07/tree2.jpg)

为什么要提到上面这几点不显然和不严谨的地方，因为只要当你看到算法书上出现不显然和不严谨的地方，基本上就意味着作者其实跳过了关键的思维步骤。

不幸的是《Algorithms》这本书里面讲霍夫曼编码已经算是讲的好的了，如果你翻开著名的CLRS，看一看当中是怎么证明的，你就知道我说的什么意思了。有时候这些证明是如此的企图追求formal和严谨，一上来就定义符号一大摞，让人看了就想吐。

说了这么多，*有没有可能把霍夫曼编码讲的更好呢*？前面说过，霍夫曼编码我记了又忘，忘了又记，好几次了，有一次终于烦了，心想如果要自己去证明，会怎么去证，那个时候我已经忘了《Algorithms》里面怎么讲的了。所以我得从头来起，首先，对于算法问题，有一个一般性原则是，**先看一看解空间的构成**。尤其是对于搜索问题（最优化问题可以看做搜索问题的一个特例），这一点尤其重要。霍夫曼编码的可能的编码树是有穷的，如果穷举所有的编码树，然后找到那棵代价最小的，这种方法至少是可行的，有了可行的方法（即便是穷举）至少让我们内心感到踏实。

接下来便是提高搜寻效率的问题。而提高搜寻效率的关键（同样也是一个一般性原则），便是**尽量去寻找问题条件能够推导出来的性质，然后利用这些性质去避免不必要的搜寻**，只要你学过二分搜索就应该理解这个一般性原则：二分搜索的效率之所以高于“穷搜”（O(n)），便是因为它利用了问题中的性质（有序）来避免了不必要的搜寻。有时候这个性质甚至可以直接将时间降为O(1)，例如在一个有序数组中寻找出现次数大于n/2的数（假设该数存在），利用“该数一定出现在数组正中间”这个性质，我们直接就避免了所有的计算。

不过，话虽如此，有时候这些性质并不是那么“显然”的，需要对问题进行深入的折腾才能有可能发现。第三个一般原则：如果你要搜寻的元素是某个满足特定条件的元素（例如寻找最优解的时候，“最优”的定义就是这个“特定条件”），那么可以“**倒过来推**”（数学证明常用手法，结论当条件使），即假设你已经找到了你要找的元素，那么能得出哪些结论，每一个结论都是最优解的一个必要条件，而*每一个必要条件都能够帮助你避免不必要的搜寻*，因为你*只要发现某个候选解不满足某个必要条件，就可以立即将其丢弃*，前面提到的寻找出现次数大于n/2的例子是一个极端情况，我们得出的必要条件导致我们可以直接丢弃除中点元素之外的一切其他元素，再例如如果有人叫你寻找有序数组中最小元素，你会毫不犹豫地把该数组头尾元素中较小的那个给他，因为你知道“如果那个最小元素存在，那么它*必然*位于头尾”——这个必要条件直接允许你丢弃掉n-2个候选解。

回到霍夫曼编码问题，按照这个原则，我们会去*假设已经得到了最优编码树*，那么我们能够发现关于它的什么性质呢？这里又要提到另一个适用于很多最优化问题的原则（前面提到的原则适用于一般性搜索问题），**所谓最优解，就是说比其他所有解都要更好**，虽然这句话听上去像是废话，但是它的一个直接推论——**比与它邻近的所有候选解都要好**——就是一个非常有用的，不是废话的性质了。学过微积分的都知道，光滑函数的最值点必然是大（小）于其邻域内的所有点的，然后再根据这个就自然推出该点的一阶导数（切线斜率）必然为0的性质，这个性质（必要条件）让我们直接省掉了去整个区间内搜索的麻烦，从而可以直接锁定有限几个候选解。那么，既然我们说最优霍夫曼树一定比它“附近”的树更好，我们就想看看，怎么来找到它附近的树。我们知道要从一个点到它附近，往往是对这个点进行一些调整，例如N+1是到达附近的另一个整数。霍夫曼树是一棵树，所以对这棵树的所有的一次“改动”（或“折腾”）都能够到达与它的“改动”距离为1的点（是不是想起“编辑距离”这个概念），怎么改动呢？最符合直觉的（虽然并不是唯一的）改动便是把叶子节点进行互换。

于是我们得到一个重要的推论：

* *在最优霍夫曼树中，无论互换哪两个叶子节点，得到的树都变得更“差”。（严格来说是不会变得更“好”，因为最优树未必唯一）*

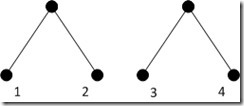
这个性质看上去有点像废话，值得费这么多事么？值得。因为虽然前文说了很多，但都是大多数人大脑里面既有的，一般性的法则，前面说过，如果我们能够从我们已经掌握的一般法则出发来推导出问题的解，那么记忆负担是最小的，因为这里面用到的所有法则我们都很清楚，也知道怎么一步步往下走。

上面这个性质究竟意味着什么呢？如果你假设这两个叶子节点的频率为f1和f2，深度为d1和d2，互换它们的时候，其他叶子节点的cost保持不变，令为常量C，那么互换前总cost为C+f1d1+f2d2，互换后为C+f1d2+f2d1，既然互换之后的树一定更”差“那么就是说f1d1+f2d2 < f1d2 + f2d1，简单变换一下就得到结论：f1(d1-d2)<f2(d1-d2)，也就是说如果d1<d2，那么f1必然>f2，如果d1>d2，那么f1必然<f2。换言之就是*叶子节点的深度越高，频率必须越低*，否则就不可能是最优霍夫曼树。那么，之前我们觉得不那么显然的结论便呼之欲出了：频率最低的叶子节点必然位于树的最底层，频率最高的叶子节点必然位于树的最高层。

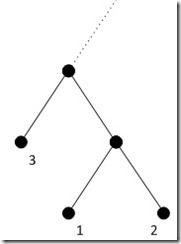
有了这个结论之后，我们便能够对最优霍夫曼树的构建走出确定性的一步，即，将频率最低的两个叶子节点放在最底层。别小看这一步，这一步已经排除了大量的可能性。这里，我们容易一开始天真地觉得最底层只有这两个叶子节点，于是它们拥有共同父节点，*这样一来霍夫曼树的整个拼图便已经拼好了一个小小的角落*。

然后我们会发现，要是它们不是兄弟怎么办呢？这里提到另一个一般原则——**归约**。不是兄弟的情况能否归约为是兄弟的情况？反正我们要求的是*一个*最优解，而不是*所有的*最优解，我们只需证明，如果当这两个最低频率的叶子不是兄弟的时候的确存在着某棵最优霍夫曼树，那么通过交换它们各自的兄弟，从而让这两个叶子团聚之后，修改后的树仍然是最优的就可以了。事实情况也的确如此，证明非常直接——既然这里涉及到的所有4个节点都在最底层同一个高度上，那么互相交换的时候不会改变他们任何一个人的深度值，所以总cost不会改变。

但是接下来我们犯了难，整个树的一个小小的樱桃状的局部是确定下来了，接下来怎么办呢？一个最自然的思路就是考虑第三小的叶子，因为前面说了，元素频率越低就越位于树的底部嘛。第三小的叶子有两种可能的归属，一是跟最小的两个叶子同样位于最底层（这不会违反我们前面得到的推论），这个时候第三小的叶子的兄弟叶子肯定是第四小的叶子，如下图：

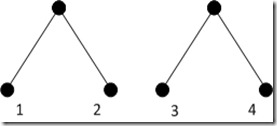
[](http://mindhacks.cn/wp-content/uploads/2011/07/tree3.jpg)

另一种归属就是往上一层去（注意，一旦第三小的叶子往上去了一层，那么剩下的所有叶子都必须至少在这个层以上），往上一层去了之后，它的兄弟是谁呢？不妨将它和刚才第一第二叶子的父节点结为兄弟（前面证明过，同层之前节点互换不会改变编码的cost），如下图：

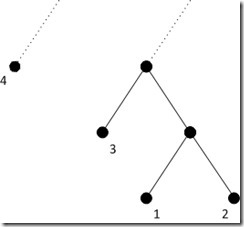
[](http://mindhacks.cn/wp-content/uploads/2011/07/tree5.jpg)

可是现在问题出现了：虽然第一步构建（最小的两个叶子）是确定的，但是到了第二步摆在我们面前的就有两个选择了，到底选择哪个呢？一个办法就是把两种选择都记下来，然后继续往下走。可是别小看两种选择，接下去每一步都有两种选择的话就变成指数复杂度了。所以现在我们便有了动机*回头看一看*，*看问题中是否有什么没有发现的性质能够帮助我们再排除掉其中一个选择*。理想情况下如果每一步都是必然的，确定的，那么N步我们就可以构建出整棵树，这是我们希望看到的，抱着这个良好的愿望，我们仔细观察上面两种构型，一个自然而然的问题是：这两种构型都有潜质成为最优解吗？如果我们能够证明其中一种构型不能成为最优解那该多好？就省事多了嘛。这里引入另一个一般性的解题法则：**特例**。我们的**大脑喜欢具体的东西，在特例中折腾和观察会方便的多**。

上面这个{1, 2, 3, 4}的例子就是个很好的特例，如图（注：图中节点旁的数字一概为*频率值，*而非编号）：

[](http://mindhacks.cn/wp-content/uploads/2011/07/tree31.jpg)

多加折腾一番*也许*我们不难发现，如果将1，2及其父节点跟叶子4进行交换（注意：*交换的时候1，2也被一同带走了*，因为反正1，2两个节点已确定是好兄弟永远不会分家了，折腾的时候只能作为一个整体移动，所以这里也可以说是**交换子树**），那么树的编码将会变得更优，因为这样一次交换会将1和2的深度+1，意味着整棵树的代价+3，而同时会将叶子4的深度-1，也就是说整棵树的代价-4，总体上整棵树的代价就是+3-4=-1（注意，在计算的时候我们只需考虑被交换的局部，因为树的其他部分的代价保持不变）。如下图：

[](http://mindhacks.cn/wp-content/uploads/2011/07/tree4.jpg)

这个交换启发了我们，其实前面一开始说的交换两个叶子节点*可以推广为交换内部节点和叶子节点*，然后很快我们就会意识到其实可以推广到交换任意两个节点。（注意，*当我们说交换内部节点的时候，其实是连同该内部节点作为局部根节点的整个子树都交换过去*）于是前面我们的推论就可以推广为：

* *在最优霍夫曼树中，无论互换哪两个节点，得到的树都变得更“差”（交换内部节点则是连同该内部节点作为局部根的子树一同带走）*

这个推论很容易理解，只不过是多增加了一种“编辑”最优霍夫曼树的方法罢了（记住*最优霍夫曼树无论怎么“编辑”都不会变得更“好”，包括“交换子树”这种“编辑”*），我们前面没有想到这种“编辑”方法是因为它不那么显然，而且当时我们已经想到一种最直接的“编辑”方法了，即交换叶子，就容易顺着那个思路一直走下去，直到我们发现必须寻找新的性质，才回过头来看看有没有其他法子。

当然，并不排除一开始就想到这种推广的可能性，问题求解的过程并不是这么线性的，如果我们习惯了推而广之的思维，也许一下就能想到这个推广来。类似的，也不排除从另一种思路出发想到这种推广的可能性。所以这里只是可能的思维轨迹中的一种，重点在于其中并没有某处忽然出现一个不知从哪里冒出来的，神启一般的结论。

刚才提到，构造最优树的第二步是考虑第三小的叶子，但也有另一种常见的思维：考虑到第一步（即选取频率最小的两个叶子）所做的事情是从N个叶子中选择两个黏在一起作为兄弟，那么也许对于一些人来说自然而然的第二步就是试图继续选取两个节点黏在一起作为兄弟（注意这里不仅可以选择叶子，也可以选择已经生成的内部节点），然后依次类推来拼完整棵树。按照这一思路，第二步的选项仍然还是集中在第三小的叶子上，因为这个选择要么是让第三第四小的叶子结拜为兄弟，要么是让最小两个叶子的父节点和第三小的叶子结拜。

回到刚才我们的推论：在最优霍夫曼树中，无论互换哪两个节点，得到的树都变得更“差”（交换内部节点则是连同该内部节点作为局部根的子树一同带走） 。根据这个推论我们容易计算出，*在最优霍夫曼树当中，两个内部节点n1和n2，如果n1比n2更深，那么n1下面的所有叶子的频率之和必然要小于n2下面所有叶子的频率之和*。如果交换的是一个内部节点和一个叶子节点，则道理是类似的。这个性质的证明和上面的类似，就不赘述了。

这个性质暗示了一个重要的推广结论：*如果我们把每个内部节点的所有叶子的频率之和标在它旁边，那么整棵树的每个节点便都有了一个数值，这个数值遵循统一的规律：即越往深层越小*。这就意味着，我们刚才的二选一困境有办法了！当我们将最小的两个叶子f1和f2合并的时候，生成了一个新的节点M，这个节点有一个数字（为两个叶子的频率之和f1+f2），根据上面的推论，*这个数字f1+f2跟所有频率一同，遵循最小的在最底层的原则*，所以我们下一步必须在剩下的那些互相之间关系待确定的节点（叶子节点和内部节点）之中，即{(f1 + f2), f3, f4}里面选择最小的两个数字结合成兄弟（由于f1和f2这两个节点已经铁板钉钉结为整体了，所以从集合里面可以看做移除）。到这里，我们就发现递归已经出现了，接下去的过程对于绝大多数人应该就真的很显然了。

以上的解释，比《Algorithms》更简短吗？显然不是。反而要长得多（其实真正的思维过程比这要更长，因为中间还会涉及各种不成功的尝试）。但是它比《Algorithms》当中的版本更不容易被忘记，因为其中关键的思维拐点并不是毫无来由的，而是从你已经熟知的，或者说虽然不知道，但容易理解的一般性解题法则出发自然推导出来的，所以你基本上不需要记忆什么东西，因为你需要记的已经在你脑海中了。

在上面的证明过程中，还有一个不像看上去那么显然的事情：在我们寻找最优霍夫曼树的时候，我们曾经试图去比较假想的最优树和它的“临近”的树，从而去探索最优树的性质。但是，究竟什么是临近的树？在前面的讲解中，我们说如果交换A和B这两个叶子节点，便得到一颗不同的树，可以看做和原树的“编辑距离”为1的树。但是，真的这么显然么？难道除了交换叶子的位置，就没有其他办法去“折腾”这棵树了？后来我们看到，可以*交换子树*而不仅仅是叶子，而交换子树让我们得到了至关重要的推论。此外，如果不是交换，而是像AVL树那样“旋转”呢？说到底，二叉树是一个离散的东西，并不像连续值那样，天生就有“距离”这个概念，如果我们离散而孤立地去看待所有的树，那么没有什么临近不临近的，临近本是一个距离的概念，除非我们定义树和树之间的距离函数，才能说临近与否，而距离函数怎么定义才是“显然”的呢？

还有，其实以上只是试图给出最优霍夫曼树的*证明*的一个更自然的过程，而*当年霍夫曼面临这个问题的时候根本还没有人想到要用二叉树呢！更不要说在二叉树的前提之下进行证明了*。根据wikipedia的介绍，霍夫曼同学（当年还在读Ph.D，所以的确是“同学”，而这个问题是坑爹的导师Robert M. Fano[给他们作为大作业的](http://www.huffmancoding.com/my-family/my-uncle)，Fano自己和Shannon合作给出了一个suboptimal的编码方案，为得不到optimal的方案而寝食难安，情急之下便死马当活马医扔给他的学生们了）当年为这个问题憔悴了一个学期，最后就快到deadline的时候“忽然”想到二叉树这个等价模型，然后在这个模型下三下五除二就搞定了一篇流芳千古的论文，超越了其导师。

最后说两个有趣的现象：也许很多人会觉得，越是大师来写入门教科书越是好，其实很多时候并非如此，尤其是在算法设计和数学领域，往往越是在其中浸淫久了越是难写出贴近初学者的书，因为大量对初学者来说一点都不显然的事情在他看来已经是“不假思索”了，成了他的内隐记忆，尤其是当他想要和你解释一个复杂的东西的时候你就会发现他会常常逻辑跳跃，满嘴跑术语，根本没有意识到别人对有些术语和隐含的逻辑根本没有像他那样的理解。

**最适合将一个东西讲给别人听的时候并不是等懂了很多年以后，而是刚刚弄懂的时候**，这个时候从不懂到懂的差别记忆还非常鲜明，能够清清楚楚地记得到底是哪些关键的地方是最折磨人的，也最能够站在不懂者的角度来思考问题。像波利亚这样，成了大师还能够站在不懂者角度去换位思考的，可以说是凤毛麟角。所以说前Amazon CAO（首席算法官）的[《Introduction to Algorithms: a Creative Approach》](http://www.amazon.com/Introduction-Algorithms-Creative-Udi-Manber/dp/0201120372/)绝对是本罕见的好算法书）

[知其所以然（一）](http://mindhacks.cn/2008/07/07/the-importance-of-knowing-why/)里面曾经提到，要弄清来龙去脉，最好去看看原始作者是怎么想的，可是正如上文所说，即便是最初的发明者，在讲述的时候也会有意无意地“线性化”，我就去查看了霍夫曼最初的论文，那叫一个费解，不信你可以自己看看([PDF](http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf))。

可以归约为搜索算法的问题（非常多）一般来说相对还是有一些头绪的，因为搜索空间一般还比较容易界定，难点在于要从问题的条件中推导出用于节省搜索的性质。而策略设计问题则完全是另一个世界，因为策略的设计空间貌似是可列无穷的，常常让人感觉无从下手，摸不着头绪，许多让人挠头的智力问题就有这个特点（例如著名的100个囚徒和1个灯泡的房间就让很多人有这种感觉），策略设计问题也有一些较通用的法则，以后再说。

怎么才能在学算法的时候学到背后的东西呢？有以下几点很重要：

1. **不要觉得每个步骤都很显然**，每个nontrivial的算法背后都有一段艰辛的探索经历，觉得显然的话必然是一种幻觉。**Stay foolish**，才能发现某些环节其实并不是那么显然的。
2. 检验是否真正理解的最佳方法就是**过一段时间之后，自己试着证明一次**。如果真正理解了的话，你的证明便会比较顺畅。如果当时没有真正理解，那么凡是那些你当时觉得显然但其实不显然的地方，都会成为你证明里面缺失的环节。
3. 对于一个算法，**多寻找各种来源的资料**，也许能够找到一个讲的比较深刻的。我在[《数学之美番外篇：快排为什么那么快》](http://mindhacks.cn/2008/06/13/why-is-quicksort-so-quick/)和[《知其所以然（一）》](http://mindhacks.cn/2008/07/07/the-importance-of-knowing-why/)里面都举到了这样的例子。
4. **多试着去抽象背后的一般性法则**，即便后来发现抽象得是错的，也比不去抽象要好。[抽象是推广的基础](http://mindhacks.cn/2009/03/28/effective-learning-and-memorization/)。只有抽象出更深层的法则，才能让你事半功倍，触类旁通，否则一个萝卜永远是一个坑。（注意，其实我们的下意识是会进行一定程度的抽象的，例如前面提到的原始人的例子，小溪和小河（或者小沟）细节上是不同的，但本质上是一样的，我们的大脑会自动进行这种简单抽象，提出事物的共性。正因此，即便你不去有意识地总结一般规律，只要你看的足够多，练的足够多，必然就会越来越谙熟。）

最后留个问题：虽然按照上文的方式来构造霍夫曼树一定能够得到*一个*最优树，但是怎么证明一定能得到呢？乍一看这个问题似乎很多余，因为证明很简单：我们拼装整棵树的每一步都没得选，而且每一步都必然拼凑出最优树的一个小小局部，如果最终还没有得到最优树的话，只能说最优树是不存在的了，然而最优树是一定存在的，因为所有树的集合是有穷的，有穷集必有最值，因此证毕。这个证明固然是没问题的，但它其实是一个间接证明，换句话说，我们在构建树的过程中的逻辑是这样的：“之所以我们选择粘结n1和n2，是因为其他粘法必然违反最优树的两个性质。所以我们别无选择。”但是，这并没有说，我们选择了粘结n1和n2，一定就*符合*了最优树的性质。（也就是说“其他做法都是错”并不能推出“这种做法必然对”，这就像是你在一大堆豆子当中寻找一个特殊的豆子，你拿起一个，看看不是，扔掉，又拿起一个，还不是，扔掉，排除到最后只剩一个豆子了，假设你又知道这个特殊的豆子必然存在，那么这个时候你根本不用看就知道这个豆子一定就是你要找的）那么，你能否直接证明，拼装最优树的过程每一步都*符合*最优树的性质呢？

P.S.

[1] 《逃出你的肖申克》和《BetterExplained》是我喜欢的两个系列，还会继续写，我有很多问题，也在Evernote里面记了不少零碎的思考和资料，但只有当我觉得理解的足够深入，系统，以及手头有足够的有意思和有说服力的例子的时候，我才会把整条线串起来成文，所以这事慢慢来不着急，反正这个博客也不会关掉。

[2] 工作之后可用业余时间急剧减少，已经陆续基本把GReader砍掉了，时间再往前推，砍掉邮件列表，再往前是Twitter，再往前是BBS。现在基本就只剩邮件了。越来越发现当时间有限的时候，看书比看网要有效得多，也不会那么信息焦虑，网络上的那些消息当中真正重要的会自己来找你，不用每天去刷屏。不过有个例外，我过一阵子就会去逛一下Amazon的个性化推荐项目。如果你已经工作，苦于时间有限，我建议你这么做。最近看过的几本值得好好推荐的书有：《Number Sense》，《Reading in the Brain》，《The Vision Revolution》，《The Tell-Tale Brain》，《Kluge》。

[3] 顺便吐槽国内出版社引进Pop Science类书籍的效率和质量，就我观察，台湾引进Pop Science类书籍需要延迟两年左右，大陆则从三四年到无限期不等（某种程度上，一个国家的出版方的认识水平，决定了这个国家大众的认识水平。你去看下我在豆瓣的书单就知道有多少好书与国内读者失之交臂了），例如《Number Sense》这本好书，到现在还没有引进，99年出版的书啊。《Kluge》更是译为《乱乱脑》这种坑爹的书名，封面搞得跟少儿读物一样。《Reading in the Brain》引入的算较快的，但也延迟了一年半了，而且翻译质量也不是很上乘（算是不功不过吧），说到这里要赞中信出版社，最近一年引入了很多给力的Pop Science畅销书，眼光还算不错。最近在Amazon上搜一些好的发展心理学的书，通过Amazon的推荐引擎看到了[《Pink Brain，Blue Brain》](http://www.amazon.com/Pink-Brain-Blue-Differences-Troublesome/dp/0618393110)，这本受到因研究大脑记忆的分子机制而获诺奖的Eric Kandel盛赞的科普09年就出了，到现在国内影子都见不着，还好在卓越上买到了原版。虽然基本还没开始看，但可以郑重推荐给初为父母的同学们:)