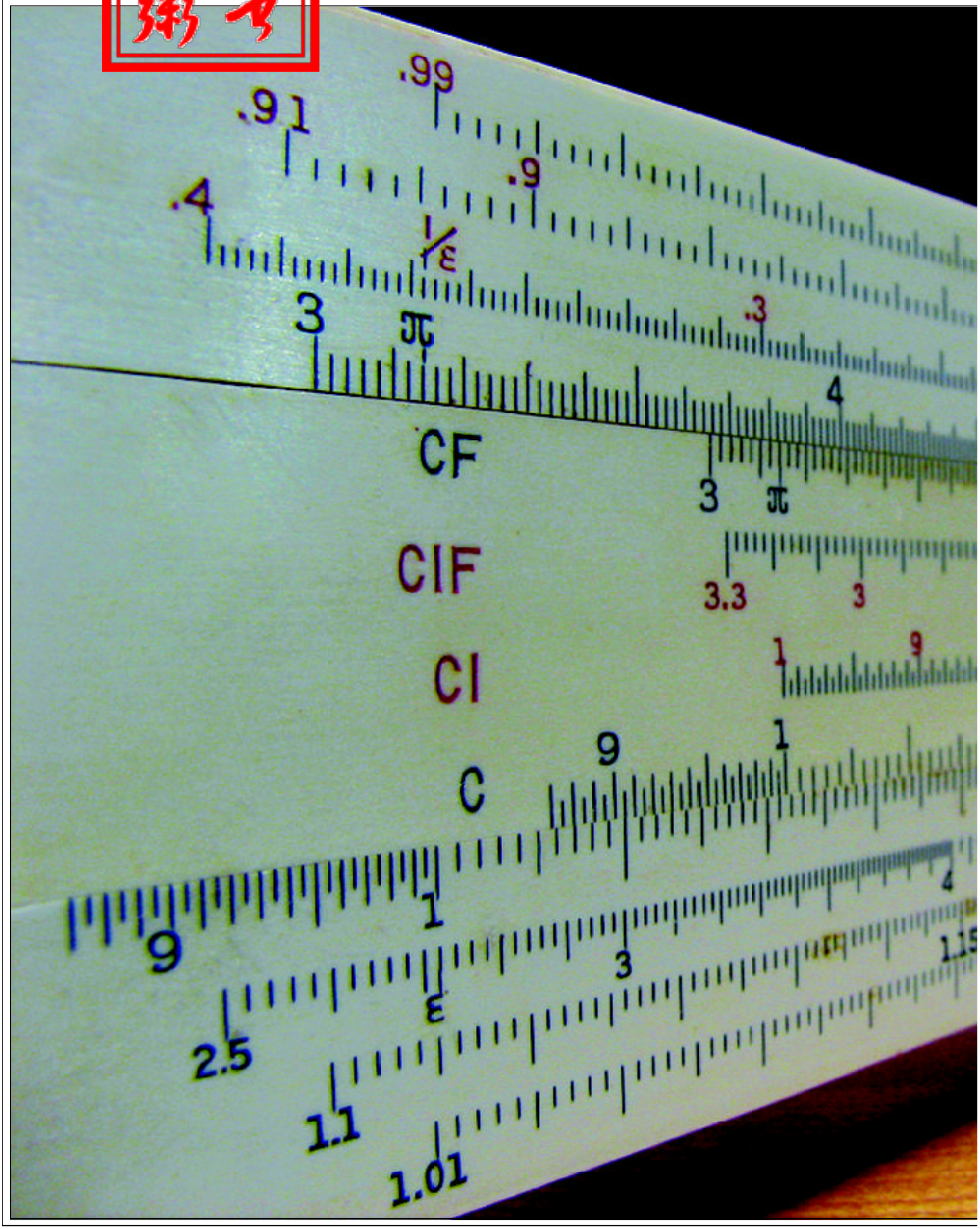
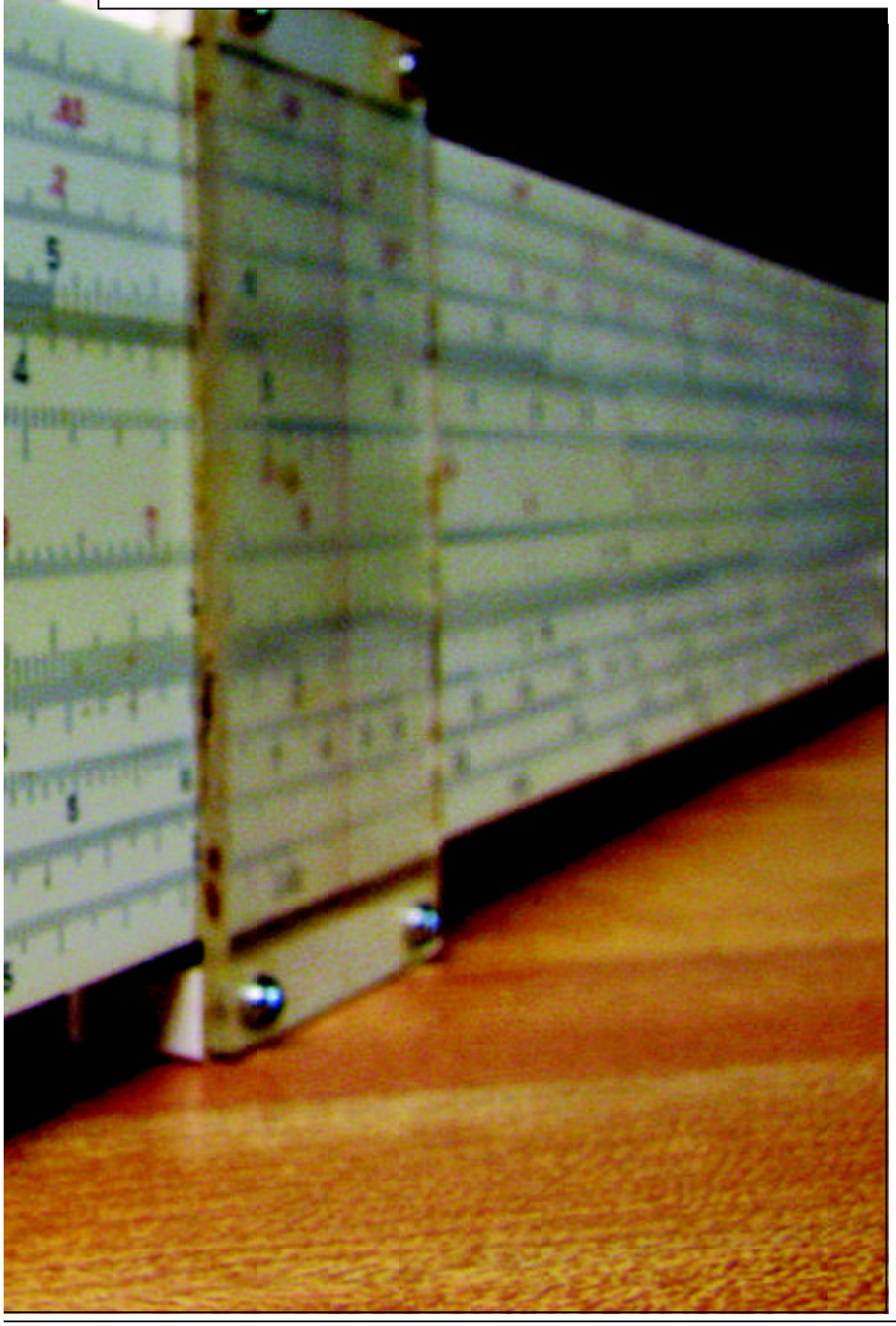
特别策划

Features



算法的力量

□ 策划 / 本刊编辑部



从学生时代，我们就被告知以“编程 = 算法 + 数据结构”。然而长期以来，国内的软件开发领域过分关注企业级开发，导致了目前中国软件开发人员对于算法的基础越来越不重视，这势必影响到软件产业发展的创新动力。在国家号召“自主创新”的背景下，重提算法的意义就显得格外重要了。

今天，我们可以看到一些真正的大型软件企业四处开设研究中心，类似机构并不直接生产软件产品，而是专门从事理论研究、发表论文。这些成果顺其自然地成为企业可持续发展的动力。软件公司巨头在研究上投入的资本将作为他们的核心技术，创造更多价值。因此，国家号召“自主创新”是有其道理与意义的，这也是为了国产软件能在全球软件的大市场中占有一席之地。

算法的应用，在图形图像、金融、安全、制造等多个领域正发挥着举足轻重的作用。然而要用杂志短短数页文字来记载这些并非易事，但至少我们能够讲述一些思想，让程序员从此开始正视算法，正视扎实的基础。

算法的力量

□ 文 / 李开复

算法是计算机科学领域最重要的基石之一，但却受到了国内一些程序员的冷落。许多学生看到一些公司在招聘时要求的编程语言五花八门，就产生了一种误解，认为学计算机就是学各种编程语言，或者认为，学习最新的语言、技术、标准就是最好的铺路方法。其实，大家被这些公司误导了。编程语言虽然该学，但是学习计算机算法和理论更重要，因为计算机语言和开发平台日新月异，但万变不离其宗的是那些算法和理论，例如数据结构、算法、编译原理、计算机体系结构、关系型数据库原理等等。在“开复学生网”上，有位同学生动地把这些基础课程比拟为“内功”，把新的语言、技术、标准比拟为“外功”。整天赶时髦的人最后只懂得招式，没有功力，是不可能成为高手的。

算法与我

当我在1980年转入计算机科学系时，还没有多少人的专业方向是计算机科学。有许多其他系的人嘲笑我们说：“知道为什么只有你们系要加一个‘科学’，而没有‘物理科学系’或‘化学科学系’吗？因为人家是真的科学，不需要画蛇添足，而你们自己心虚，生怕不‘科学’，才这样欲盖弥彰。”其实，这点他们彻底弄错了。真正学懂计算机的人（不只是“编程匠”）都对数学有相当的造诣，既能用科学家的严谨思维来求证，也能用工程师的务实手段来解决问题——而这种思维和手段的最佳演绎就是“算法”。

记得我读博时写的Othello对弈软件

获得了世界冠军。当时，得第二名的人认为我是靠侥幸才打赢他，不服气地问我的程序平均每秒能搜索多少步棋，当他发现我的软件在搜索效率上比他快60多倍时，才彻底服输。为什么在同样的机器上，我可以多做60倍的工作呢？这是因为我用了一个最新的算法，能够把一个指数函数转换成四个近似的表，只要用常数时间就可得到近似的答案。在这个例子中，是否用对算法才是能否赢得世界冠军的关键。

还记得1988年贝尔实验室副总裁亲自来访问我的学校，目的就是为了想了解为什么他们的语音识别系统比我开发的慢几十倍，而且，在扩大至大词汇系统后，速度差异更有几百倍之多。他们虽然买了几台超级计算机，勉强让系统跑了起来，但这么贵的计算资源让他们的产品部门很反感，因为“昂贵”的技术是没有应用前景的。在与他们探讨的过程中，我惊讶地发现一个O(n*m)的动态规划(dynamic programming)居然被他们做成了O(n*n*m)。更惊讶的是，他们还为此发表了不少文章，甚至为自己的算法起了一个很特别的名字，并将算法提名到一个科学会议里，希望能得到大奖。当时，贝尔实验室的研究员当然绝顶聪明，但他们全都是学数学、物理或电机出身，从未学过计算机科学或算法，才犯了这么基本的错误。我想那些人以后再也不会嘲笑学计算机科学的人了吧！

网络时代的算法

有人也许会说：“今天计算机这么快，算法还重要吗？”其实永远不会有太快的计算机，因为我们总会想出新的应用。虽然在摩尔定律的作用下，计算机的计算能力每年都在飞快增长，价格也在不断下降。可我们不要忘记，需要处理的信息量更是呈指数级的增长。现在每人每天都会创造出大量数据(照片，视频，语音，文本等等)。日益先进的记录和存储手段使我们每个人的信息量都在爆炸式的增长。互联网的信息流量和日志容量也在飞快增长。在科学研究方面，随着研究手段的进步，数据量更是达到了前所未有的程度。无论是三维图形、海量数据处理、机器学习、语音识别，都需要极大的计算量。在网络时代，越来越多的挑战需要靠卓越的算法来解决。

再举另一个网络时代的例子。在互联网和手机搜索上，如果要找附近的咖啡店，那么搜索引擎该怎么处理这个请求呢？

最简单的办法就是把整个城市的咖啡馆都找出来，然后计算出它们的所在位置与你之间的距离，再进行排序，然后返回最近的结果。但该如何计算距离呢？图论里有不少算法可以解决这个问题。

这么做也许是最直观的，但绝对不是最迅速的。如果一个城市只有为数不多的咖啡馆，那这么做应该没什么问题，反正计算量不大。但如果一个城市里有很多咖啡馆，又有很多用户都需要类似的搜索，那么服务器所承受的压力就大多了。在这种情况下，我们该怎样优化算法呢？

首先，我们可以把整个城市的咖啡馆做一次“预处理”。比如，把一个城市分成若干个“格子(grid)”，然后根据用户所在的位置把他放到某一个格子里，只对格子里的咖啡馆进行距离排序。

问题又来了，如果格子大小一样，那么绝大多数结果都可能出现在市中心的一个格子里，而郊区的格子里只有极少的结果。在这种情况下，我们应该把市中心多分出几个格子。更进一步，格子应该是一个“树结构”，最顶层是一个大格——整个城市，然后逐层下降，格子越来越小，这样有利于用户进行精确搜索——如果在最底层的格子里搜索结果不多，用户可以逐级上升，放大搜索范围。

上述算法对咖啡馆的例子很实用，但是它具有通用性吗？答案是否定的。把咖啡馆抽象一下，它是一个“点”，如果要搜索一个“面”该怎么办呢？比如，用户想去一个水库玩，而一个水库有好几个入口，那么哪一个离用户最近呢？这个时候，上述“树结构”就要改成“r-tree”，因为树中间的每一个节点都是一个范围，一个有边界的范围(参考：http://www.cs.umd.edu/~hjs/rtrees/index.html)。

通过这个小例子，我们看到，应用程序的要求千变万化，很多时候需要把一个复杂的问题分解成若干简单的小问题，然后再选用合适的算法和数据结构。

并行算法：Google的核心优势

上面的例子在Google里就要算是小case了！每天Google的网站要处理十亿个以上的搜索，GMail要储存几千万用户的2G邮箱，Google Earth要让数十万用户同时在整个地球上遨游，并将合适的图片经过互联网提交给每个用户。如果没有好的算法，这些应用都无法成为现实。

在这些的应用中，哪怕是最基本的问题都会给传统的计算带来很大的挑战。例如，每天都有十亿以上的用户访问Google的网站，使用Google的服务，也产生很多很多的日志(Log)。因为Log每分每秒都在飞速增加，我们必须有聪明的办法来进行处理。我曾经在面试中问过关于如何对Log进行一些分析处理的问题，有很多面试者的回答虽然在逻辑上正确，但在实际应用中是几乎不可行的。按照他们的算法，即便用上几万台机器，我们的处理速度都跟不上数据产生的速度。

那么Google是如何解决这些问题的？

首先，在网络时代，就算有最好的算法，也要能在并行计算的环境下执行。在Google的数据中心，我们使用的是超大的并行计算机。但传统的并行算法运行时，效率会在增加机器数量后迅速降低，也就是说，十台机器如果有五倍的效果，增加到一千台时也许就只有几十倍的效果。这种事倍功半的代价是没有哪家公

司可以负担得起的。而且，在许多并行算法中，只要一个结点犯错误，所有计算都会前功尽弃。

那么Google是如何开发出既有效率又能容错的并行计算的呢？

Google最资深的计算机科学家Jeff Dean认识到，Google所需的绝大部分数据处理都可以归结为一个简单的并行算法：Map and Reduce（http://labs.google.com/papers/mapreduce.html）。这个算法能够在很多种计算中达到相当高的效率，而且是可扩展的（也就是说，一千台机器就算不能达到一千倍的效果，至少也可以达到几百倍的效果）。Map and Reduce的另外一大特色是它可以利用大批廉价的机器组成功能强大的server farm。最后，它的容错性能异常出色，就算一个server farm里面的机器宕掉一半，整个farm依然能够运行。正是因为这个天才的认识，才有了Map and Reduce算法。借助该算法，Google几乎能无限地增加计算量，与日新月异的互联网应用一同成长。

算法并不局限于计算机和网络

举一个计算机领域外的例子：在高能物理研究方面，很多实验每秒钟都产生几个TB的数据量。但因为处理能力和存储能力的不足，科学家不得不把绝大部分未经处理的数据丢弃掉。可大家要知道，新元素的信息很有可能就藏在我们来不及处理的数据里面。同样的，在其他任何领域里，算法都可以改变人类的生活。例如人类基因的研究，就可能因为算法而发明新的医疗方式。在国家安全领域，有效的算法可能避免下一个911的发生。在气象方面，算法可以更好地预测未来天灾的发生，以拯救生命。

所以，如果你把计算机的发展放到应用和数据飞速增长的大环境下，你一定会发现，算法的重要性不是在日益减小，而是在日益加强。■

李开复给程序员的七个建议

1. 练内功。不要只花功夫学习各种流行的编程语言和工具，以及某些公司招聘广告上要求的科目。要把数据结构、算法、数据库、操作系统原理、计算机体系结构、计算机网络，离散数学等基础课程学好。大家不妨试试高德纳所著The Art of Computer Programming里的题目，如果你能够解决其中的大部分题目，就说明你在算法方面有一定的功力了。

2. 多实战。通过编程的实战积累经验、巩固知识。很多中国大学毕业生缺乏编程和调试经验；学习C语言，考试过关就算学会了；课题项目中，只要程序能够编译，运行，并且输入输出满足要求就算了事。这些做法是不行的。写程序的时候，大家必须多想想如何把程序写得更加精炼、高效、高质量。建议大家争取在大学四年中积累编写十万行代码的经验。我们必须明白的是：好程序员是写出来的，不是学出来的。

3. 求实干。不要轻视任何实际工作，比如一些看似简单的编码或测试。要不懈追求对细节一丝不苟的实干作风与敬业精神。我发现不少程序员对于知识的掌握很肤浅，不求甚解，没有好奇心，不会刨根问底。比如，学会了C++，是否了解一个对象在编译后，在汇编代码中是如何被初始化的？这个对象的各个成员在内存中是如何存放的？当一个成员函数被调用时，编译器在汇编代码中加入了哪些额外的动作？虚函数的调用是如何实现的？这些东西恐怕在编程语言或编译原理中都没有详细提到，只有通过踏实的实干才能真正掌握。

4. 重视数学学习。数学是思维的体操，数学无处不在。学计算机至少要学习离散数学、概率论、布尔代数、集合论和数理逻辑。这些知识并不难，但是对你未来的工作帮助会很大。尤其当你对一些“数学密集型”的领域如视频、图像处理等有兴趣时，这些知识将成为你手中的利器。

5. 培养团队精神，学会与人合作。今天的软件工程早已经不是一个人可以单独操作的，而必须靠团队合作才能成功。不懂得合作的人是不能成大器的。大家要多去寻找可以与人一起做项目的机会。

6. 激励创新意识，培养好奇心，不要死记硬背。没有掌握某种算法技术的根本原理，就不会有应变和创新的能力。想成为一位好程序员（其实从事任何一个行业都是如此），重要的是要养成钻研，好奇，创新，动手，合作的优秀习惯，不满足于填鸭，不满足于考试交差，不满足于表象。这不是学几门课能够一蹴而就的。

7. 有策略地“打工”。在不影响学业的前提下，寻找真正有意义的暑期工作或兼职。去找一个重视技术的公司，在一个好的“老板”指导下完成真正会被用户使用的程序。不要急于去一个要你做“头”而独挡一面的地方，因为向别人学习才是你的目的。找工作也是一样，不要只看待遇和职衔，要挑一个你能够学习的环境，一个愿意培养员工的企业，一个重视你的专业的公司。最后，还要挑一个好老板。

希望大家都能把握机会，养成好的学习习惯，把算法学精学透；希望大家都能有一个美好的未来！

算法为魂

□ 文 / 凌小宁

《程序员》的欧阳先生写信邀我写篇关于算法的文章。他在信中说:“国内一般的开发企业和开发者偏浮躁，缺乏扎实的基本功，这直接导致他们自主创新的能力低下”。这使我想起我在微软中国研究院工作时接触到的学生。在我面试过的几百个学生中，我惊讶的发现相当一部分学生算法的功底不够高，不能胜任微软的编程工作。“不够高”有多高？讲一个例子。一次有个学生问我:算法是什么？与程序有什么区别？我很惊讶有这样的问题，并在惊讶之余随口答道:“算法是魂，程序是衣。”

这回忆使我感到有责任借这个机会写点什么。就以“算法为魂”为题吧。我想以此文来挑战每位学生:你有没有毅力来摄取算法这软件之魂；我也想以此文来挑战每位算法老师：你有没有远见卓识来努力培育学生的软件之魂，成为中国未来软件人才的“灵魂”工程师。

由此可见，此文是写给学软件的学生和教软件的老师的。但我相信其他的软件工作者也会从中得到些益处。我不假设读者需要具备算法知识，虽然了解一点算法知识会对了解本文有帮助。

在进入正题之前，让我们先回顾一点历史，这能帮我们理解算法的本质。

历史回顾

算法一词（Algorithm）是由九世纪数学家al-Khwarizmi的名字翻译而来。它初期的概念是指解决问题或执行任务的确定的过程。1842年，Ada Byron为他设想的自动计算器写了世界上第一个算法。但这算法未能被真正实现，因为Ada Byron未能造出他的自动计算器(http://en.wikipedia.org)。

现代意义上的计算机算法的概念是在1936年Alan Turing提出现代计算机的基本模型图灵机之后才清晰起来(http://www.turing.org.uk/turing):算法是解决问题或执行任务的过程；它能够一步一步地在图灵机或等价的机器(如现代的计算机)上执行。一个典型的计算机算法可以输入零个，一个，或多个数据，经过有限个确定的、精确的步骤，得到一个或多个结果。对这一过程的设计和分析称为算法的设计与分析。随着计算机和软件的普及，它已成为世界所有大学计算机学科的最重要的核心课程之一。

在大多数算法教材中，算法常常是用伪程序来描述的。伪程序较接近高级计算机语言，但更易写易懂。其实算法的表述可以有多种形式:自然语言（英文，中文，…），各种计算机程序设计语言，甚至硬件。由此可见，算法的本质是问题解决过程的概念，而相应的程序只是一种它的表述。现在你应该明白为什么我说“程序为衣”了吧，而且那只是若干件衣中的一件而已。

但是“算法为魂”是什么意思呢？如果我再进一步说:算法是程序员之魂，算法是软件架构师之魂，算法是软件管理者之魂。那是什么意思呢？

现在让我们进入正题——算法为魂!

算法为魂

初级算法课程主要是讨论典型问题的典型算法的设计方法和分析方法。以我在微软做研发十三年的经验，我可以毫不犹豫地告诉大家，我读书时在这门课上花的每一分钟都是绝对值得的:

◆ 算法研究的典型问题包括计算机最常用的分类、排序、搜索、遍历、集合运算等等。我在微软写的众多的程序中的每个程序，都会碰到至少一个，常常是多个这样的问题。

◆ 算法的常用设计方法包括循环、递归、分治、动态规划、线性规划、搜索与枚举、启发式搜索等等。这些方法都是可以举一反三的通用方法。我在微软写过的和看到的程序，几乎都是用的这些方法。

◆ 算法的主要分析方法是建立与算法相应的数学公式，来计算该算法所用的运行时间和存储空间（称为算法的性能分析）。通过这样严格的数学训练可以获得一种直觉的对算法性能的估计能力。这种能力是微软的软件大腕儿都具备的。

举两个例子。大约在1995年间，微软的软件开发工具部门（Visual Studio）需要一个图形自动布局算法。当时微软内部没有这项技术，准备以百万美元从

外面购买。我当时正好在这个部门工作，又因为对算法的兴趣，就“抢”下了这个项目。这是个相当困难的问题。值得庆幸的是，我在学校获得的算法知识帮助我成功地设计了这个算法，并使其在微软的五个产品（Visual Studio、Visual InterDev Studio、SQL、Access等）中得以应用。我也因此而获得了我的第一项美国技术专利。

第二个例子。在微软，软件工程师的设计都要经过设计评审会（Design Review）通过，以保证设计质量。我参加的最有趣的一次设计评审会也发生在软件开发工具部门。当时，大家对被评审者提出的设计方案没有一致意见，以至争得眼红脖子粗的，僵持不下，甚至动了粗口。这时有位资深程序员站出来不紧不慢地说：这个设计的基本算法的速度增长是指数级的，当输入数据量增长到一定量级时，这个设计将不能满足用户响应时间。在他详细解释了他的分析后，所有人都同意了他的结论，并否认了这个设计方案。想一想，如果你是那位被评审者，你应该感到惭愧，因为你的算法功底不够，做出了不适合的设计；如果你是那位资深程序员，你应该感到自豪，因为你为公司排除了一个错误的设计，一个潜在的产品中的隐患。

这两个例子和我在微软所见所闻和所经历的，都使我坚信，算法对程序员来说真的是极为重要的，称它为程序员之魂是绝不为过。如果你仍不相信，就再看看当今最成功的高科技公司，微软的对头Google。Google今天的成功，最重要的原因之一就是它的搜索引擎中的许多算法优于竞争对手的，使得它的搜索结果更好。如果你到Google的招聘网站去看看，你会发现它对程序员的首要技术要求就是算法！

有个学生告诉我，他的理想不是当程序员而是当软件架构师。我说：当架构师是个很好的理想，但不当程序员而当架构师不是理想是幻想。我知道的架构师，包括微软首席架构师比尔·盖茨，视窗和办公软件的总架构师，都曾是最优秀的程序员。软件架构设计不仅仅是画些框图写些流程而已，它需要坚实的基础（包括算法基础）和丰富的经验。

出个题给大家。假如你是个软件服务架构师，假如你公司的老板让你为公司设计个电子商务网站，假如你的市场调查部门告诉你预期的用户数大约为第一年五万，第二年三十万，第三年一百万，假如你的需求分析部门告诉你宽带用户操作的平均响应时间应小于N秒：请你设计个软件服务系统架构；它能满足上述要求并具有最低成本。这里有三个关键的要求。第一，用户响应时间要短。在互联网时代，性能就是金钱。你的系统太慢，用户就跑到你的竞争对手那去了；第二，系统是可扩充的。你的系统应能容易地随着用户的增加而扩充，并避免结构上的改变。这样就能减少你的成本，达到对你的第三个要求——低成本。现在你应看出这题目的用意了。而系统性能分析，虽然不在目前大学算法课程的范围内，但显而易见，算法分析是完成这题目的基本功之一。

还有不少学生告诉我，他们想成为软件开发管理人员。也是很好的理想。但我总是鼓励他们先做个优秀的程序员。为什么？软件开发不同于其它工业开发，它要求给与程序员足够的自由空间和灵活性，才能发挥他们的创造性，以得到高效率。因此，以权利和地位施加高压的管理方法并不是最好。在微软，软件开发团队最有效的管理方法之一是以身作则。最优秀的开发经理和开发组长都是最优秀的程序员。他们能帮助组员解决技术上的难题，包括算法上的难题。他们领导程序和算法的技术审核。他们以优秀的编程算法技术经验，赢得组员的尊重，并成为高效率的软件开发领导者。不难看出，扎实的算法编程功底对软件开发管理人是多么重要。

其实，研发算法本身就是件极有趣的事。随着网络及多媒体的普及，各种应用的日新月异，对新算法的需求也越发多起来，例如，压缩（视频、音频、图形、图像）、streaming、多媒体信息处理、海量信息存储、搜索与管理、安全（加密、解密）、分布系统等等。随着硬件的进步（例如多处理器多核PC等），并行算法也已成为当务之急。所以，算法研发本身就是个很广的舞台，这个舞台的支柱就是算法基础。

算法基础学习

如果你找到好的方法，学好算法基础并不是件很困难的事。这里我给大家提四个建议。

第一，先学好相关的数学。高等代数是算法分析中最常用到的，必须掌握。离散数学中的布尔代数、集合论、排列组合等是算法的基础的基础，应该掌握好。数理逻辑或形式逻辑教你推理和证明方法，对建立算法性能公式和算法性能分析很有帮助。微积分中的的极限理论可以帮助你更好地理解算法性能公式。

第二，讨论计算机算法不能脱离数据结构。因此，数据结构是学习算法的基础。有些大学把算法和数据结构放在一起教，是一种可行的办法。

第三，学会方法比记住结论更重要！设计算法的几种通用方法，例如：循环、递归、分治、搜索法、动态规划和线性规划等等，都是程序设计中最常用的方法，都是可以举一反三的“灵丹妙药”，应成为算法学习的重点。

第四，像学习软件专业的其他课程，学习算法的最好的方法就是编程练习。找一个你最感兴趣的问题，设计出几个不同的算法解，对每个算法，推导出性能公式，为每个算法编出程序，运行这些程序并统计出实际性能，比较你的理论结果和实际结果，最后，做出分析。这个过

程，既可以帮助你建立坚实的算法编程基础，又可以培养你对算法及其性能的直觉。这两者对程序员都是极为重要的。

算法教学革新

现在我要为老师们写几句。相信这些话对学生和工业界的朋友们也会有些益处。

到目前为止，我都在讲好话。其实，当前大学的算法教学太过传统，已不能完全适应近十几年来信息产业的飞速发展所带来的挑战。我相信这也是为什么当前算法教学走下坡路的原因之一。现在就让我们看看这些挑战，看看这些挑战给我们带来的算法教学革新的机遇。

◆ 基于新计算平台的算法。传统的基础算法教学假设算法的计算平台是单机单中央处理器系统。由于因特网，网络，及单机多核系统等新型计算平台的出现和大量普及，这一假设对许多实际应用中的算法设计已不再正确。微软的一个资深架构师Herb Sutter在最近的一个国际会议上声称：今天的软件工程师完全不能适应正在来临的单机多核时代。另外，在网络和互联网上，算法的性能瓶颈常常已不再是单机算法而是网络带宽，传统的算法性能分析方法已远远不够用了。虽然学习单机单中央处理器的算法依然是最基本的，学习分布算法，客户端 - 服务器算法和并行算法已变成为当务之急。

◆ 基于新型媒体的算法。传统的基础算法教学大都以文字和数字作为算法的输入数据类型。这也开始变得不尽人意了。随着半导体技术的飞速发展，计算机的速度和内存空间迅速增加。根据公认的摩尔定律，到2010年普通PC的速度将超过30G 赫兹，内存将超过30GB，而硬盘将达到4TB。根据戈德尔定律（Gilder’s Law），网络带宽也在快速增加。我相信，在这样的超级硬件能力下，多媒体（视频、音频、图像、图形）将在不远的将来成为计算机和互联网的主要信息形式。于是多媒体算法（压缩、streaming、存储、表达、处理、搜索等等）将在变得越来越重要。我们所面临的挑战是：算法教学怎样反映这种技术大趋势？

◆ 算法的可读性、可扩充性和可维护性。长期以来，算法教学总是把算法的性能（速度和所用存储空间的大小）作为衡量算法好坏的唯一的标准。其实，随着计算机应用领域的飞速发展，算法越来越复杂，相应的程序越来越大，加上硬件性能价格比几何级数地增长，算法的性能（速度和空间）常常不再是最重要的考量了。现在，在软件工业界的一个不争的事实是：算法和程序的可读性，可扩充性和可维护性常常要比性能来得更重要！有统计数据表明，一个算法或程序的维护费用要比它们的开发费用高几倍。如果我们能把一个算法设计得易读，易懂，易于修改扩充，易于维护，我们将能有效地降低软件的成本。现在学术界已开始认识到这个来自工业界的挑战，例如美国麻省理工学院教算法教授Charles Leiserson（http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science）。

这些挑战是严峻的紧迫的。我耳闻目睹微软的许多研发人员每天都面临着这样的挑战。如果我们的学校能够改革算法教学，使其包含更多的新信息时代的内容，将会对软件产业产生重要影响，也能有效的提高学生学习算法的积极性。这是时代带给我们的新的机遇，算法教学改革的机遇！

坦率地说，算法教学的改革不是件易事。我们需要与软件工业界和研究部门合作，以确定当前最基本的问题，最主要的算法需求，最核心的算法，和最重要的设计方法。我们需要解决教师的知识更新问题。我们需要精简传统的算法教学内容又能保持严格的算法理论训练。我们需要面向实际又能保持理论性和普遍性的原则。我真希望看到有一所大学能站出来，有一所软件学院能站出来，有一位教授能站出来，迎接这个挑战，抓住这个机遇，为算法教学开出一片新天体。如有需要，我愿为此作出力所能及的微薄贡献。

最后，我想重复本文开始的两句话。我想以此文来挑战每位学生：你有没有毅力来摄取算法这软件之魂。如果你做到了，你就为你未来的事业奠定了一个很好的基础，不管你将成为程序员，软件架构师，或软件研发管理者。我也想以此文来挑战每位算法老师：你有没有远见卓识来接受新信息时代对算法的挑战。如果你做到了，你将成为新信息时代中国未来软件人才的“灵魂”工程师。

算法资料连接

1. 算法与数据结构中文网站：http://algorithm.diy.myrice.com
2. Dictionary of Algorithm and Data Structure: http://www.nist.gov/dads
3. Algorithm courses on Web: http://www.cs.pitt.edu/~kirk/algorithmcourses
4. Complete Collection of Algorithm Animation: http://www.cs.hope.edu/~alganim/ccaa
5. MIT Open Courseware: http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science

作者简介

凌小宁，现兼任北京大学软件学院软件技术系系主任；北大，上海交大，和北航客座教授。于1993年加入微软至今。现在是微软美国研究院总部项目规划经理。曾担任微软中国研究院的软件开发总工程师，开发部经理；是微软中国研究院的创始人之一。曾参与开发微软Visual Studio，Access，SQL，Visual InterDev，MSN，Office 等多个产品。曾获得图形学，用户界面，图像学方面的美国专利。曾获得美国OSU博士学位，中国北大的硕士和学士学位。凌小宁博士的邮件地址是：xiaonl@hotmail.com，如果您对本文有更多想法，可以直接写信与他联系。

黑铁时代的算法“悖论”

□ 文 / 王咏刚

在我的印象里，从来也没有哪件事情能像算法在国内程序员心中的“映射”那样奇特和复杂。

套用古希腊人的说法，历史的演进总要以秩序日趋混乱和话语权逐渐分散为代价。1970年以前，当那些令人景仰的IT先驱们沉浸于计算技术的美妙绝伦时，算法在为数不多的程序员心中，也许就和克洛诺斯在天国中的地位一样，是足以统治整个黄金时代的精神力量。在随之而来的白银时代（大概以UNIX的兴起和繁荣为起迄时间吧）和青铜时代（当然是Windows和桌面计算一统天下的时代）中，算法仍足以与其他蓬勃兴起的技术要素（比如语言和平台）分庭抗礼。但在网络时代的中国，算法却陷入了颇为尴尬的境地：有关算法重要性的话题竟然成了展开BBS论战的最好诱因；“算法无用论”居然被用做了网络“砖家”们赖以出名的金字招牌；在主流技术媒体中，算法的位置也早已被难以计数的“新框架”和“新标准”所取代——对算法而言，这是一个不折不扣的黑铁时代。

在这样的时代里讨论算法的价值并介绍相关的学习方法，这并不是一件十分招人喜欢的事。幸好，我是一个不太愿意关心或追随“主流”舆论的程序员，又是一个没事儿时总想提高程序效率和可用性的工程师——很愿意结合自己的感受谈些与算法相关的话题。祝大家阅读愉快！

算法的重要性
——这真有讨论的必要吗？

作为旁观者（AKA潜水员），我看过许多有关算法重要性的网上口水战。说实话，我一直都不觉得这件事真有讨论的必要，直到有一天，我读到了这样一条有关算法的“悖论”：

因为算法很重要，所以，在我们想要设计或封装某种高水平的算法之前，总已经有人把类似的事情做得很好很好了；于是，程序员只要知道算法库的接口就可以了；由此可知，算法一点儿也不重要。

这则“悖论”出自一位经常帮别人扣上“重新发明轮子”的大高帽的网友，或许我们可以把它看做“轮子帮”的看家本领之一。要推翻这种所谓的“悖论”其实不算太难，细心的读者应该很容易发现其中不符合逻辑或曰无法“自洽”的地方。不过在这里，我并不想把这篇“漫话”性质的文章变成论证严密的逻辑辨析，还是用一句引语和两个实例来藐视这种不负责任的“悖论”吧。

在MIT那本颇有分量的《算法导论》第二版中，作者是这样描述算法的重要性的：

是否在算法知识和算法技能方面拥有扎实的基础，这是区分真正的编程高手与编程菜鸟的一个标志。借助现代计算技术，即便对算法知之甚少，你也可以完成某些任务；但是，如果有一个良好的算法基础，你所能做的事情就会更多、更多。

在《JOEL说软件》一书中，著名的Joel Spolsky先生在全书的开头就抛给了所有程序员一个巨大的问号：C语言中处理字符串的方式是最优的解决方案吗？

我知道99.99%程序员都会熟练使用strcpy或是strcat，但一定有相当多的程序员无法圆满回答这样一个看似简单的问题。既然C语言的运行库已经为我们提供了处理字符串的基本数据结构和基本算法（没错，算法无处不在，即便是获取字符串长度这样极简单的操作，不同的算法实现也有效率高低的差别），我们还有必要仔细测量这些可复用的“轮子”

是圆是方吗？

很遗憾，Joel告诉我们，C语言对字符串的处理是“最差的方法之一”；在许多情况下，“用户应该像躲避瘟疫一样地避开使用ASCIZ字符串”。Joel还很得意地告诉我们，他参与开发的Excel软件在内部使用Pascal格式的字符串，这是“Excel的速度快得出奇的一个原因”。

如果说Joel与Excel的故事还不足以说明算法知识或算法技能在我们身边的重要性，那么，我这里还有一个和大家更近一些的真实案例。

2003年，我们为上海一家银行开发一套企业级报表管理软件。当时，银行每天打印的报表超过10万页，此外还有大约1万余份的PDF或XLS报表需要存储、索引和管理。程序员们的工作十分顺利，系统很快上线运行，客户对产品还算满意。到了2003年底，每日产生的报表量翻了一番。按说，这样的潜在需求增长原本是签在合同里的，程序员设计系统时也“充分”考虑了软硬件的冗余处理能力。但不幸还是发生了：业务量增加以后，原本只需要40分钟即可完成的报表导入、解析和索引流程硬是延长到了5个小时，在磁盘阵列和永久存储设备（主要是DVD库）之间同步数据的操作也要多花上4个小时。客户每天日终处理之后留给系统的可支配时间被大大超支，其结果自然是第二天上班的业务人员在中午吃饭前多半查不到头一天的任何报表——用客户方项目负责人的话说就是：“后果很严重，领导很生气。”

一名项目经理立即被派往现场解决问题。实地考察之后，项目经理拍着胸脯对客户说：“只要增加3台服务器和1部DVD库，并将原有服务器的内存扩大1倍，系统效率就可以满足设计要求。”

客户方项目负责人十分爽快地说：“好呀，好呀，给你们一周时间采购和部署，下周一我一定要看到一套健康、稳定、合格的系统。”

项目经理说：“可是……新设备的采购费用呢？难道不需要新签一份合同吗？”

客户一脸诧异地说：“合同？不是去年就签过了吗？费用？夏天我们就把款付给你们了！我们现在需要的是满足合同约定的系统。是不是要采购新设备，那是你们的事情，好像与我们无关吧？”

很显然，国内许多设计师和程序员都有这样一种思维定式：

计算机的处理速度越来越快，程序员不需要也不应当绞尽脑汁以提高代码的运行效率，而是应把更多的精力放在架构设计和过程管理方面；当业务量成倍增加时，程序员要做的第一件事不是优化代码，而是申请更多、更好的硬件设备。

客观地说，这样的想法有一定的道理，但千万别把它绝对化。因为更加残酷的事实是，硬件的发展似乎总也无法满足新应用的增长需要。无论硬件速度多么惊人，总会有新的应用将CPU、内存、硬盘或是带宽资源消耗得一干二净——为了新版游戏而不断升级显卡的玩家对此一定深有体会。另一方面，项目经理在上海的遭遇也证明，如果你把所有希望都押在硬件括容上面，你一定会碰到被客户逼着自掏腰包的窘迫场面。

其实，任何有算法或数学基础的程序员都很容易判断出，如果一套系统在2003年的典型硬件环境中无法处理每日20万页左右的报表数据，那么，该系统使用的算法几乎一定存在极大的改进空间。事实也的确如此，当我们的项目经理不得不潜下心来，仔细审读和调试源代码后，他很快就找到了两个要命的算法问题：为了简化设计，报表解析程序选用了更容易实现的多遍扫描和递归下降算法，而存储模块使用的索引文件也仅仅采用了最为简单的二叉有序树。

那些天生对O(g(n))敏感的程序员一定可以找到许多有效的解决方法，我们的项目经理也不例外。在这里，我想说的只不过是：

学习和掌握算法知识，这并不意味着程序员一定要亲自设计或实现特定的算法；但如果缺少这些知识，你可能会在面临一大堆可用算法的时候不知道该怎样择善而从。

一点声明——误会是这样产生的

算法很重要，这一点毫无疑问。但必须严正声明的是：

我从来都没有说过算法是程序设计领域里惟一重要的东西。

对一个程序员来说，与算法同等重要的还有好几件事情，比如数学，比如英语，比如逻辑推理和判断能力，比如合作与沟通能力，比如语言和设计模式，比如……这些事情共同构成了程序员之所以能成为一门职业的知识基础。

如果一定要说几件不那么重要的事，我大概会说，一个具体的类库（比如Log4j），一项具体的技术（比如AJAX），一个具体的技巧（比如解决JavaScript的Closure问题），或是诸如此类的东西都是相对而言不那么重要的事情；因为一个基础过硬的程序员完全可以在两三周的时间内学通、学会它们中的任何一个——我知道这样的说法很容易产生误会，但我还是希望借此提醒更多的程序员深入思考相关问题。

一个常见错误——企业级应用必然排斥自主创新吗？

不小心看到过这样一段文字：

长期以来，国内的软件开发领域过分关注所谓“企业级开发”，导致了目前中国软件开发人员对于算法基础越来越不重视，这势必造成软件产业发展缺乏创新的动力。

因为自己此前做了将近八年的企业级应用开发，所以对这样的断言十分敏感：难道企业级应用一定会排斥自主创新吗？难道优秀的算法只能诞生在微软或

是CMU吗？难道在中国程序员中占绝大多数的行业软件开发队伍就永远没有机会享受设计或实现顶尖算法的愉悦了吗？或者，换个角度考虑，即便国内软件开发人员不重视算法基础的事实的确存在，它与企业级开发之间存在必然的因果关系吗？

据我所知，IBM是为世界各地提供企业级软件及服务最多的公司之一，即便在IBM专为企业而设计的应用软件（比如DB2 Content Manager）中，也隐藏着许许多多精妙的算法（比如工作流模块中的任务调度算法）。

另一个真实的例子是，以前的一位同事负责过一个电信行业的系统集成项目。在项目早期，他简单地将Oracle，Crystal Report Server，Oracle Application Server Portal等几种软件集成在一起，自己只开发了少量的存储过程和控制脚本就实现了所有功能。但随后他逐渐发现，这种简单集成的做法无法有效推广到其他类似的应用中。为提高软件的适应能力，他亲自编写了一个从多个异构系统中采集数据并重新包装、分发的小模块。就是为了更好地实现这样一个小巧的“数据粘接”模块，他几乎查阅和尝试了所有流行的数据归并和排序算法，以至于他在做完项目后骄傲地对我说：“现在如果让我回学校再考一次《数据结构》或是《算法导论》，准保95分！”

我知道，国内有很多开发企业级应用的程序员每天多半是在Struts或Spring的框架里填写一些“模式化”的代码。他们也许会认为，自己的工作离算法很远很远，与其抓耳挠腮地研读算法源码，还不如在Hibernate大虾的Blog上多泡两个钟点。对此我想说的是：

程序员的工作并没有高低贵贱之分，但如此多的程序员集中在简单应用中的事实却正好折射出中国软件产业的尴尬现状：是产业本身缺乏自主创新意识（或曰层次不高）造成了国内企业级软件开发的过度“简单化”和“蓝领化”，而不是企业级应用的存在影响了自主创新或是算法本身的生命力。

我想，即便一时无法改变这样的现状，聪明的程序员仍然可以利用算法知识大幅提升自己的开发经验和设计水平。比如，同样是使用SQL Server+ASP.NET的组合，熟悉数据库的索引和查询机制，了解.NET中的对象调度和事务处理算法的程序员总是可以编出更加高效，更容易适应不同规模应用的程序来；再比如，尽管大多数人都可以把内存回收的重任交给Java或.NET环境提供的垃圾收集机制，但熟悉垃圾收集算法的程序员总能比较清楚地知道，自己编写的程序可能在什么样的情况下占用更多的内存，这样的内存消耗会不会成为系统的瓶颈之一。

所以，即便你是一个从事企业级软件开发的程序员，你也应该对自己有足够的自信：

每个人都有权利享受算法本身的乐趣，每个人都有权利参与到创新中来！

两种学习方法——你是数学家还是工程师？

学习算法至少有两种方法，一种是数学家的方法，一种是工程师的方法。

很可惜，很多人只知道第一种方法，对第二种方法几乎闻所未闻。以至于许多程序员一提“算法”或类似的字眼儿，就会皱起眉头说：“可恶！又是算法！数不清的公式，蜘蛛网一样的图示，烦都烦死了！”

其实，对于很多只需要在实际应用领域使用某种算法的程序员来说，学习一种算法并不意味着你一定要懂得算法背后的数学原理，也不意味着你一定要背会若干公式和代码，我们完全可以用更加便捷也更加有效的方式学习复杂的算法。

例如，数据结构或算法类教材讲到最短路径搜索算法时，总会先用有向图或无向图的知识精确定义最短路径问题，然后用集合或矩阵的术语详细描述Dijkstra算法、A*算法等常用的搜索算法。这样的教学形式很难对一个身处紧张工作中的程序员产生多大的吸引力。与此相反，在斯坦福大学博士生Amit Patel的个人主页上，我们可以找到一份更加适于工程师学习的最短路径搜索算法指南“Amit's Thoughts on Path-Finding and A-Star”（http://theory.stanford.edu/~amitp/GameProgramming/）。

在这份指南中，我们看不到晦涩的概念引入，找不到繁杂的数学证明。大家所能见到的只是一段段从实际应用（游戏）出发的，有趣而浅易的讲解，以及一幅幅描述搜索路径的，形象而明白的示意图。

建议大家（无论对游戏编程是否有兴趣）都去看一看Amit的教程。最重要的不在于Amit教会了我们多少算法，而在于Amit很好地展示了学习算法的另一种有效途径——从实践出发的方法，或曰工程师的方法。

给自己和自己从事的工作一个准确定位，然后寻找一种学习算法的最佳道路——我自己就一直是这么做的。用数学家的方法学习算法当然不错，如果有可能，我甚至愿意重新回到大学校园温习相关的数学和算法知识；但只要我还在从事具体的编程工作，只要我的需求重在使用算法而不是发明算法，我就会沿着属于工程师的道路一直走下去。

几句结束语

◆ 算法很重要——但没有程序员本人更重要。

◆ 算法很有用——这与你的工作类型关系不大。

◆ 算法不难学——当然，这取决于你的学习态度和学习方法。

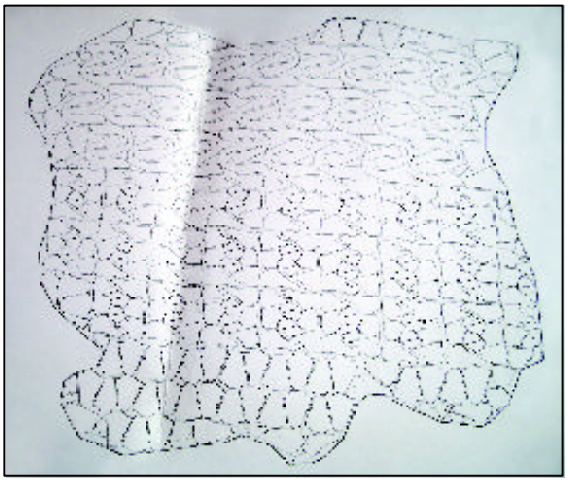
◆ 希望和大家一起把算法学得好一些，再好一些——如果有时间的话。

周培德访谈录

□ 记者 / 欧阳璟 刘龙静

周培德教授住在北京理工大学的家属区里，记者怀着忐忑不安的心情拜访这位老教授之前，生怕这是一场授课式的采访。然而结果却让我们始料未及……

一上来，周教授就为我们展示了如下这张图：



算法解决皮鞋下料问题的手绘图

这是周老师退休以后，一位对算法颇有兴趣的皮鞋制造商专门请周老师设计的下料算法。随后，他便和记者开始侃侃而谈。

记者：现代计算机算法大致应当怎样分类？各个分支的主要情况是怎样的？

周培德：如果让我对算法发展方向作全面的介绍，恐怕不合适。但是算法分类，我还是有一定的了解。

算法按照基本操作分有两类，分别是数值算法与非数值算法。数值算法主要是以算术运算来进行的，大

◀周培德，1941 年生，北京理工大学退休教授，武汉大学数学系毕业。从 1982 年开始研究算法，曾独立发表学术论文七十余篇，并有两本专著及两本研究生教材。退休后仍然从事算法设计方面的工作，帮助很多国内企业解决实际生产问题。

学的时候有一门课程是计算分析，后来叫做数值计算，这门课程当中介绍的基本上都属于数值算法；而非数值算法，也就是我所关注的这个部分，主要通过计算机的赋值、比较、重写或者逻辑运算等操作解决问题。目前我所关注的主要方面是在非数值算法，它主要解决如下的一些问题：排序、搜索、串匹配、图、组合学、几何学、数论等。

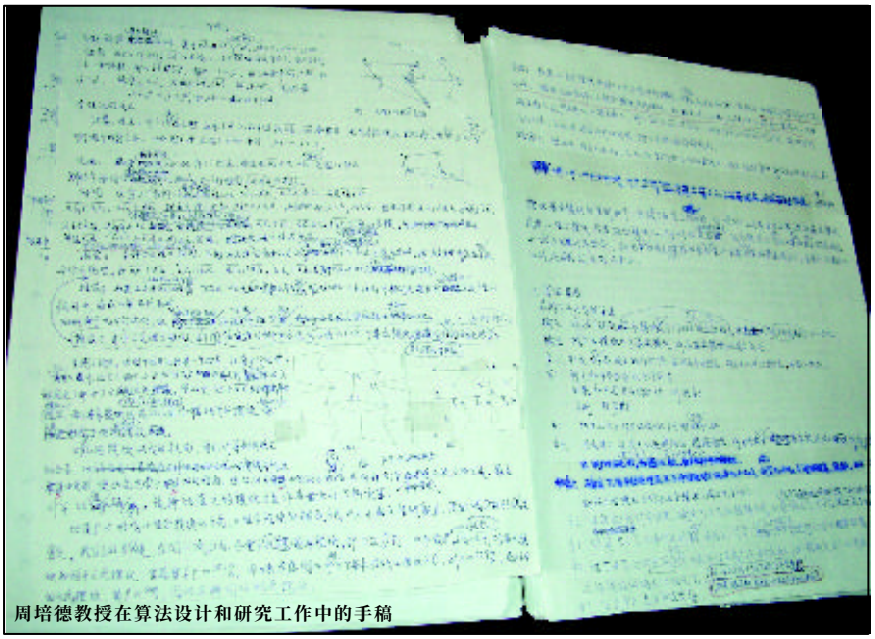
按执行顺序又分成串行算法与并行算法。目前我们常常看到的计算机，主要都是串行计算的机器，并行机当然也有，但是相比串行计算机，数量上要少得多。

此外还有确定性算法以及随机算法的区分。确定性算法在应用中占了很大的比例。大体上，一个计算序列中，上一步计算完毕以后，下一步的计算是确定的。那么随机算法或者叫概率算法，其上一部的计算与下一步的计算衔接是不确定的，更多的时候，它求得的解其正确性有一定机率，而不是确切的。比如是素数的判定等算法。

记者：有人说算法是软件的灵魂，有人则认为现代软件开发中，算法的地位已经大大下降了，您怎么看？请您结合亲身经历谈谈对与算法重要性的认识。

周培德：算法是计算机科学的精髓，更是软件的灵魂。上述的观点，我是非常认同的。我还要说，无论怎样强调算法的重要性，都是不过分的。这并不是说我个人是研究算法的，所以就王婆卖瓜，自卖自夸。而是算法本身就应该受到应有的重视。目前软件开发中算法的地位大大下降的趋势是不正常的，其结果必然极大影响软件产品的质量。这种趋势应尽快扭转，不研究算法势必带来软件开发水平停滞不前的现状。与国外软件产业相比，必然处于落后的地位。根据我了解的情况，中国的软件开发水平相比印度要落后一大截，尤其是在软件开发的一些规范上做得不如他们好。国内的软件开发人员做软件都是一个人一个样。另外，他们对于算法的重视程度也比我们要高得多。因此我们应该更加重视这个问题，长此以往，对基础技术不重视将对国家经济、科技的发展将造成影响。

记者：现在主流的开发平台上都对主要的算法与数据结构进行了高效的封装，例如STL、Java Collection，压缩、加密解密等程序库也随处可用，一般开发者可以很容易地使用一些经典的算法。请问在这种情况下，还有必要把算法钻研得很深吗？


周培德教授在算法设计和研究工作中的手稿

周培德：这是一个老问题，在上个世纪八十年代，就有人提出这个问题。但经过实践证明，这种“不研究算法也可以”的观点是错误的。程序员使用现有的程序库也可以，这样能够避免重新造轮子，针对一些公共的、基础的、通用的软件确实可以为提高生产力提供价值。但现有程序库往往都不能做到包罗万象，在实际工作中提出的问题是五花八门的，需要解决这些实际问题必须研究新的算法。程序库相当于人们挖了一口井，如果只喝到这一口井中的水，欣赏这一口井下的风景，势必会让人们的眼光过于狭隘，因此不能满足于井底之蛙的角色。

我们学校在90年代初期，有几个老师提出“数值分析”这门课应该停开，其原因正是因为该课程当中的很多算法都已经包含在许多标准的程序库中了。后来这门课就真的停了好几年，我当时去找系主任，我说我不讲这门课，但是我非常反对将这样一门重要的课程停开。将来我们的学生毕业以后，只会用这些固定的程序，几个参数一套，就让程序跑起来。但是在实际工作中，如果涉及到一些条件变化，他们就不知道该怎么办了。再后来，这门课又重新开起来了。后来我有毕业出去的学生回来，谈起这件事情深有感触。他们说在大学读书，应该学习通过自学方式掌握难以学懂的知识，这种能让人产生启发的课程不能停开啊！

你比如我刚才给你看的那张按下料算法画出的图，这就是程序库中不具备的，需要你自己去设计，自己去开发。

记者：Brian Kernighan和Rob Pike在《程序设计实践》里说，常用的算法与数据结构是很少的，而真正要具备的是高超的洞察力和算法设计能力。您认为应当如何培养洞察力和算法的设计能力？

周培德：培养洞察力与算法的设计能力要做到以下三个方面：

1. 雄厚的基础，包括数学、程序设

计、算法设计方法、数据结构以及程序语言等。比如我有些学生做可视图，就是机器人在一个房间里面自由的走动和观察房间。这里面就涉及到一个问题，一些通过传感器获得的点到底是应该排列还是组合？这就涉及到数学基础了。

2. 灵感，经过长时间的训练、积累与沉淀才能产生。这不是一朝一夕或者几个月的事情，算法是越学越熟。我现在帮忙带的两个博士就有差别，好学生能提出很多问题，因为我每次给他布置的问题都认真去解决，随后，他自己也能提出有价值的问题了。比如我最近一个课题，“可见光物体的边缘提取”。光照在物体上会产生影子，但是计算机没有办法去区别影子是否属于障碍物。如果是自动运行的汽车，看到这种影子怎么办？这些问题都需要用灵感去解决。

3. 毅力，经得起失败与成功的考验，还要经得起时间和空间的考验。经得起失败这个很好理解，但是经得起成功是什么意思啊？我的书出版了，应该是个值得高兴的事情，但是我的成果还需要时间的考验，因为这些成果都是在现今条件下取得的，那许多年以后会怎样？所以这些考验都是需要认真面对的。

谈到培养学生，我对他们的要求是很严格的。一个班里算法课程能上90分的一般也就一两个人，三分之一的人不及格，大部分都是60多分，80分以上的人都很少。原来在学校里面教书，很多教职工子弟在我班里上课，一些人成绩不理想，他们家里父母就找到我。他们说，你为什么把分数卡得那么紧？我认为学生要以学知识为主，如果没有掌握必学的知识，那么给再多分数又有何用？此外，我鼓励学生自主创新，对设计了新算法并有成效的学生，以加分的办法奖励。

记者：在解决实际问题时，往往要在算法上做出创新。请您结合自己在算法上的创新谈谈这个问题。

周培德：比如说前面给你看的下料问题，这不仅是皮鞋的下料裁剪，它还涉及造船、服装、制造等等不同的行业，因此可以创造出很大的经济价值。之前我所研究的点集三角剖分，它应用在地形图处理、人体表面表示等很多表面处理的方向上。另外还有有限元分析、最短路径、k中心问题等，这些算法已经在很多行业有了相当广泛的应用。比如三角剖分就可以为在广大地区输电线布局提供良好的算法支持；而k中心问题则可以针对医学上的化疗，军事上的密集轰炸等多种方向解决问题。在原有基础上，我还自己专门设计了点线集三角剖分、线段集三角剖分等等，这些问题都可以在《计算几何》（第二版）一书中找到。

周培德教授近期出版的专著《计算几何》，这本书的原创内容超过62%，其中有88种周教授自己设计的算法

综合起来，创新无非就是下面几个问题：

1. 正确地理解和陈述问题；

2. 考虑已有算法是否可用，是否需要修改；

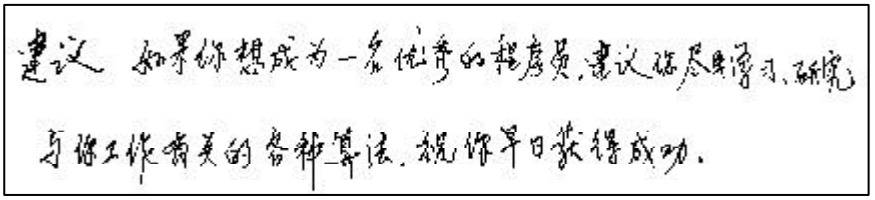
3. 如果没有现成的算法可用，就设计新的算法：考虑现有算法设计方法中哪种方法或哪几种方法可综合使用，并提出新思考，用计算机可执行的操作构成操作序列实现新思想，检验结果，再循环这个过程。

记者：数学在算法设计研究中有着怎样的作用？程序员应当特别重视学习哪几门数学课程？

周培德：数学在算法设计中起基础作用，不了解必要的数学知识，设计新算法将是非常困难的。离散数学、组合数学都是非常重要的基础学科，此外，数据结构、程序设计、高级语言等等则是专业课程中的基础，都应该好好学习。此外，还包括一些专门方向上的数学课程，比如小波分析、图像处理等等，这些课程都需要结合自己工作的实际情况来学习。算法这门课程与一般课程不太一样，如果不是当时我抱着浓厚的兴趣，则很难坚持下来。现在的学生对于算法很不感兴趣可能是因为它太难，只有在此方面培养兴趣，才能很好地理解算法。我并不希望所有的学生都钻研算法，但是在一代程序员中总应该有一些程序员是精通算法的，这样才能形成国产软件的力量。

记者：请给《程序员》读者提些建议。

周培德：如果你想成为一名优秀的程序员，建议你尽早学习、研究与你工作有关的各种算法。当然，并不是每一个人都需要在算法上有很深刻的造诣，毕竟每一个程序员所选择的发展方向是不一样的。也许几年以后，有些程序员将从事项目管理、质量控制等相关的工作，在这些工作中就不必用到算法。



基于归纳的算法设计思想

□ 文 / 黄林鹏

算法是什么？听说面试时常有考官问起，我也常常自问，但却无法给出扼要的解释。《程序员》杂志的编辑知道我最近翻译了一本算法设计的书，让我写一点体会，正好谈谈算法设计和数学证明之间的关系。其中涉及的例子和原理，大多直接取自 Udi Manber 的著作，其他原理则来自我对一些理论问题的非形式的理解。

Udi Manber 何许人也？

说起 Udi Manber 先生，可是鼎鼎大名，有空的朋友看完本文，再去访问网页 http://manber.com 就知道了。Udi 是在线信息搜索引擎的先驱，曾是美国亚利桑那大学计算机系的教授，离开学校后先是在雅虎担任执行官，然后担任亚马逊(amazon.com)的副总裁、首席算法师（CAO）和其旗下搜索网站 A9.com 的首席执行官。他提出的 UDI 测试已成为衡量搜索引擎质量的评估标准。大陆媒体最早关注他是由于“万人公开评测百度超越Google”的报道，因为该测试选用的测试标准，就是他提出的。不过最近媒体报道他已经前往Google，担任负责工程事务的副总裁。另外中国台湾许多高校选用的算法教材就是 Udi编写的，原因嘛，除了教授算法的老师是 Udi 的学生外，最主要的应该是该书的鲜明特色。

Udi 算法教材的特色：算法设计＝归纳证明

Udi算法书的特色就在于将算法设计与归纳证明进行类比，指出两者基本上是雷同的技术。这很有意思，因为大家从小学开始就学习如何使用（自然数上的）数学归纳法，很熟悉也很有体会，现在虽然面临的是算法设计这一陌生的领域，但现在知道了两者原来是一回事，岂不高兴。看过Udi书的朋友，如果再瞄一眼华罗庚先生编写的《数学归纳法》就知道他所言不虚。其实算法设计就是数学证明：算法将输入映射为输出，证明将假设与结论相关联，两者本来就是一回事。按照现在的流行话，就是多穿了个马甲。但最早指出它们是同一“家伙”的，却是赫赫有名的 Curry 和 Howard，他们揭示了程序和（构造主义）逻辑的本质关系，下面就是 Curry - Howard 的秘笈：

程序(算法、表达式)一方	逻辑一方
表达式的类型(程序的规范，算法的输入输出关系)	公式
表达式(程序、算法)	公式的证明
表达式的规约(程序、算法的执行)	证明的范式化

看到这里，大家可能会问：如果有了证明，是否就可以得到算法呢？这其实是一个很有意思的研究方向，就是程序自动生成（或者叫自动算法设计），有兴趣的读者可以参看诸如《马丁洛夫类型理论》这样的书籍。然而，目前这并不实际，因此Udi所指出的算法设计和归纳方法之间的雷同有着更现实的意义，该关系可以参阅下表：

算法设计一方	归纳证明一方
针对较小的输入的处理过程	归纳基础
假设已经知道小于 n 的问题的算法	归纳假设
从(若干)较小的问题的解得到较大的问题的解	归纳证明
算法设计技术	归纳证明技巧

虽然该原理并不是对所有问题都有效，但大部分算法都可以直接应用该原理得到，因此该原理可以看成算法设计的“方法论”。下面来介绍Udi的算法设计思想。

什么是算法？为什么要设计算法？

字典上算法的解释是“求解数学问题的一个过程，该过程步骤有限，通常还涉及重复的操作。广义地说，算法就是按部就班解决一个问题或完成某个目标的过程。”

我们日常生活中所执行的算法一般不太复杂，执行的次数也不太频繁，由于回报太低，通常不值得耗费太多的精力来开发算法。例如，当你从超市采购回家，打开购物袋放置食品的过程一般不是最有效的，你应该考虑袋中食品的形状以及厨柜的结构，但很少会有人去想这事，更不会有人专门去设计算法。但进行大规模商业包装和物品放置就必须有一个好的流程。计算机可以处理非常复杂的任务，而且同

一任务往往必须执行多次，因此值得花费时间来设计有效的方法，就算最终得到的方法更加复杂或难以理解也无妨，因为潜在的回报是巨大的。

如何进行算法设计？

算法设计是一个创造性的过程，首先你必须多多学习了解已有的有效算法，要能熟练复合已有的算法并应用于实践，但要成为一名好的体系结构师和设计出崭新的算法，还必须了解算法设计后面的原理。而学习如何创造事物的最好方式是去尝试创造它。基于数学归纳法的算法设计是一种能对算法进行深入解释的直观的优雅框架。该方法学的核心是对证明数学定理的智力过程和设计组合算法的过程进行类比。数学归纳法原理的主要思想是命题无需从什么都没有开始证明，命题的证明可以通过先证明对于一个小的基本实例集命题是正确的，然后由“若对于相对较小的命题实例成立可以导出较大的命题实例成立”的方式证明对所有实例来说命题都是成立的。将该原理转化为算法设计则提示了一种侧重于如何将简单问题的解扩充为较大问题的解的方法。给定一个问题，如果我们能说明如何应用类似问题（但参数较小）的解决方法进行问题求解，任务就完成了。因此它的基本思想是如何扩充一个解决方案而不是从零开始进行构造。

原理到实践

下面以两个个简单的问题的算法设计过程进行说明。

问题一：给定一串实数a_n，a_{n-1}，…，a_1，a_0，和一个实数x，计算多项式$P_n(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$的值。

分析一下，这个问题包含n+2个数，而归纳方法是根据较小规模问题的解来解决原问题。换句话说，首先应该把问题简化成较小规模的问题，再进行递归求解，或者说，通过归纳的方法得到答案。一个自然的想法是通过去掉a_n来简化问题，由此得到的多项式计算问题是：

$$P_{n-1}(x)=a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+\cdots+a_1x+a_0$$

除了减少了一个参数，这是一个几乎一样的问题。因此，可以通过归纳求解。

归纳假设：已知如何在给定a_{n-1}，…，a_1，a_0和点x的情况下求解多项式（即已知如何求解$P_{n-1}(x)$）。

现在通过归纳法，利用假设来求解问题。首先，必须对基本情形进行求解，即计算a_0，这是平凡的；然后，必须说明如何在较小规模问题答案（就是$P_{n-1}(x)$的值）的辅助下，求解原始问题（计算$P_n(x)$）。这一步是直接的；只要简单地计算x_n，把它乘以a_n，再把结果与$P_{n-1}(x)$相加即可：

$$P_n(x)=P_{n-1}(x)+a_nx^n$$

这个算法是正确的，但不是高效的，因为它需要$n+n-1+n-2+\cdots+1=n(n+1)/2$次乘法和n次加法运算。现在将算法修改一下，以便能更好地使用归纳法进行求解。

一个改进是我们观察到有许多冗余的计算，即x的幂被到处计算。若计算x^n时可利用x^{n-1}的值，则可以节省许多次乘法运算。由此可把x^k的计算包含在归纳假设中以得到改进：

更强的归纳假设：已知如何计算多项式$P_{n-1}(x)$的值，也知道如何计算x^{n-1}。

这个归纳假设更强，因为它需要计算x^{n-1}，但是它更有效。现在计算x^n仅需要一次乘法（就是x^{n-1}乘以x），然后再用一次乘法得到a_nx^n，最后用一次加法完成计算。共需要2n次乘法和n次加法。这个算法看起来不错，高效、简单，并且易于执行。然而，还有更好的算法存在，下面通过另一种方式使用归纳法。

通过去掉第一个系数a_0。则更小规模的问题变成了计算由系数a_n，a_{n-1}，…，a_1表达的多项式，即：

$$P'_{n-1}(x)=a_nx^{n-1}+a_{n-1}x^{n-2}+\cdots+a_1$$

（注意，现在a_n是第$(n-1)$个系数，a_{n-1}是第$(n-2)$个系数）所以有新的归纳假设。

归纳假设（翻转了顺序的）：已知如何计算由系数a_n，a_{n-1}，…，a_1表达的多项式在x上的值（知道如何计算$P'_{n-1}(x)$）。

显然，$P_n(x)=xP'_{n-1}(x)+a_0$。所以，从$P'_{n-1}(x)$计算$P_n(x)$仅需要一次乘法和一次加法。完整的算法可以用如下的表达式来说明：

$$a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0=((\cdots((a_nx+a_{n-1})x+a_{n-2})\cdots)x+a_1)x+a_0$$

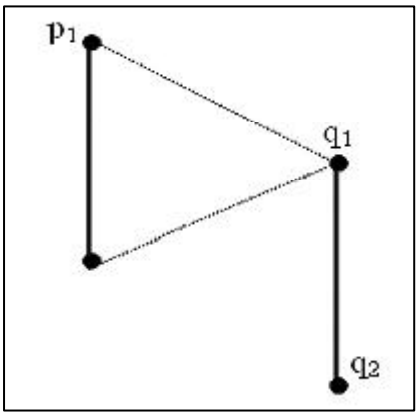
这个算法被称之为Horner规则以纪念英国数学家W.G.. Horner。算法设计好了，用什么语言来表示它就看你的喜好了。

问题二：已知集合S有n个元素x_1，x_2，…，x_n，求其中最大的和第二大的元素。

可见直接进行2n-3次比较就可以得到问题的解。但我们还可以做得更好。不妨假设n是2的幂。先试着用分治法将大小为n的集合S分成大小为n/2的两个子集P和Q。现在直接进行归纳，即假设已知P和Q中最大的和第二大的元素，分别记为p_1，p_2和q_1，q_2，然后计算S中最大的和第二大的元素。很明显，只需要两次比较就可以找到S的这两个元素。第一次比较最大数p_1和q_1，第二次比较在第一次比较‘失败’的元素和胜者的第二大元素之间进行（见图）。用这种方法得出递归关系式$T(2n)=2T(n)+2$，$T(2)=1$，解为$T(n)=3n/2-2$。，优于2n-3，但算法还可以进一步优化。

仔细观察图中的比较，可以看到算法不再使用q_2，故q_2的计算是无用的。如果我们能精简对q_2的计算，就能进一步减少比较次数。然而，在p_1和q_1比较之前，我们并不清楚p_2和q_2中哪一个不再需要。现在考虑把第二大元素的计算推迟到后面，这时需要保留第二大元素的候选者的一个列表。假定在下面的归纳假设中第二大元素未知：

归纳假设：给定一个<n的集合，知道如何找到最大的元素以及第二大元素的一个“短短的”候选者列表。



（“短短的”到底是多少尚未定义，在后面的算法设计过程中会发现合适的值的。）

算法的执行过程如下：给定大小为n的集合S，把它分为大小为n/2的子集P和Q。由归纳假设可知两个集合最大的几个元素是p_1、q_1，再加上第二大元素的候选者集合，C_p和C_Q。首先比较p_1和q_1，取其中大者，比方说p_1，作为S的最大数。然后舍去C_Q，因为C_Q中的所有元素都小于q_1，而q_1最多是第二大元素，接着再把q_1加入C_p。最终我们得到了一个最大的元素和一个候选者集合，从中可以直接选出第二大的元素。现在查找最大数所需的比较次数满足递归关系式T(n)=2T(n/2)+1，T(2)=1，解得T(n)=n-1。很容易看到，集合的大小扩大一倍，我们能在候选者集合中再加入一个元素，所以$\log_2 n$足以满足候选者集合大小的需要了。于是算法结束前还需要查找第二大元素，开销为$\log_2 n-1$次比较，因而总的比较次数是$n-1+\log_2 n-1$，这是最少的次数了。n为2的幂时的归纳假设如下：

修正的归纳假设：给定一个大小<n的集合，知道该如何求出最大的元素和第二个元素的候选者集合，这个集合最多不超过$\log_2 n$个元素。

递归算法的语义和计算问题等

基于归纳的方法得到的通常是递归的算法。所谓递归简单地说就是借助自己来刻画自己？常见的问题是：这样是有意义的吗（记得传说中曹冲和豺狼的故事吗？那个小孩说豺就是狼旁边的那只；而狼就是豺旁边的那只。你能得到什么信息。）？用处何在？先从用处说起，借助递归，我们可以使用一个或若干个等式来描述对象，比如说f。换句话说，可以定义使等式成立的那个对象就是我们要刻画的对象。这当然好了，因为描述对象的性质比直接定义对象更容易。问题是能从f满足的等式推出f到底是什么吗？在数学上，这就是大家熟悉的解方程（组），即从等式得到如f=exp(exp中不包含f的出现)这样形式的解。

再来考察算法，非形式地说，算法可以看成是一个从输入到输出的映射或者函数，如果我们能用等式描述该函数的性质，那么以后该如何应用该函数呢？有两种方式，一是利用前面得到的形如f=exp的解，二是直接对等式进行反复展开。■

下面通过一个简单的例子来阐述得到解的过程，假定定义函数f如下：

f(x)=if(x=0)then 1 else x*f(x-1)

其中x是f的参数，在定义f时起辅助作用，通过λ抽象将其从等式的左边移去，得到：

f=λx. if(x=0)then 1 else x*f(x-1)

再对等式右边的f进行抽象，得到：

f=(λf .λx. if(x=0) then 1 else x*f(x-1))f

记(λf .λx. if(x=0) then 1 else x*f(x-1))为F，上述等式化为：

f=F(f)

而这里的f，使用数学的语言来说，就是F的不动点。换句话说，现在你要定义的f就是F的不动点。那么给定F，F的不动点是什么？是否存在一个算法Y，对于任意的F，它都能计算出F的不动点？即：

YF = F(YF)

事实上这样的算法是存在的，它就是不动点算子，Chruch和Turing就曾分别定义出YC和YT如下：

Y_C = λf.VV，　V=λx.f(xx)

Y_T=ZZ，Z=λzf.f(zzf)

熟悉λ演算的读者不妨验证一下。

这说明通过f的递归等式来刻画算法f是可行的，该算法就是F的不动点YF，即f=YF。

现在回到计算上来，给定x，计算f(x)的问题。前面我们已经知道f = YF，那么f(x)=(YF)(x)，但接下来的计算却十分烦琐。前面说过还有一种计算就是对等式不断展开，直到计算出值为止。现在的问题就是这两者计算方式得到的值会是一样吗？有兴趣的读者可参阅[2]。

参考书

算法引论：一种创造性方法
黄林鹏等 / 译　原作者：[美]UDI Manber
出版社：电子工业出版社
出版日期：2005年6月

程序设计语言——原理与实践（第二版）
黄林鹏等 / 译　原作者：[美]Kenneth C.Louden
出版社：电子工业出版社
出版日期：2004年4月

算法:百度工程师的利器

□ 文 / 周利民

“工欲善其事,必先利其器”,对于百度工程师来说,算法就是他们解决难题的利器。

为什么这么说?因为百度搜索引擎研发的各个环节都离不开算法。我们需要快速,准确、实用、创新和不断改进的算法来满足用户的需求。

百度面对的是海量的互联网数据,以及每天上亿次的检索请求。它要求百度能够收录和索引超过10亿的中文网页,并提供快速的检索服务。这只有高效率的算法才能完成。

百度招聘的工程师在加入公司后,有一道入门练习题,就是编写一个数据扫描分析程序,要求写出的程序能在1分钟之内扫描分析完千万量级的数据,才算及格。高水平的程序员可以利用高效的算法在10秒以内解决问题,甚至只要六七秒。但如果没用对算法,花一星期的时间,也做不到1分钟之内。

大家可以设想,百度有十亿以上的网页,如果要在一周甚至三天内处理一遍,平均每秒处理要多少个?每天1亿次的检索又意味着峰值时每秒要处理多少次检索?事实上,针对一个问题,我们可以想出很多的算法,但如果效率不高,是无法真正投入使用的。

Web搜索引擎是一个很新的研究领域,因为从它诞生到现在不过10年左右的时间。学术界IR(Information Retrieval)领域的研究为搜索引擎提供了不少算法方面的理论基础模型,但这些理论距构建一个好的Web搜索引擎还有很大一段距离。这需要我们探索和开发很多新的算法及系统。实际上,百度搜索引擎中的很多算法都极具创新性,而且都是基于实际应用的需求。这是和学术界研究工作的一个较大差异。学术界的算法研究主要是为了解决某个学术方面的问题,不是太关注实用性,以及效率。

举个例子来说,在传统的中文分词算法研究中,学术界最关注的是能达到多高的准确率,但对算法的运行速度上考虑的相对较少。可在百度,如果使用的分词算法速度太慢,就根本无法应用。此外,百度面对的是Web上的大量数据,大部分传统的IR算法都会遇到信息爆炸的问题,我们需要想出很多新的方法来解决这些问题。

Web上的数据是不断变化的,用户的检索需求也是不断变化的。百度就是要在这不断变化的两者之间需找一个最佳匹配。所以百度的算法需要持续的进行改进,以迅速适应这些变化。比如对搜索引擎来说有一个方面的技术很重要,就是判断一个网站是否在作弊的方法。由于那些针对搜索引擎作弊的人,如果能提高搜索引擎排名,将获得巨大的经济利益,所以他们会不断使用各种方法去猜测百度算法中潜在的漏洞,进行攻击。这是一个很复杂的问题,而且仍在不断发展变化中。这就要求我们能够迅速的发现这类问题,提出算法,并应用到百度搜索引擎中。在最短时间内消灭问题。否则作弊行为很快就会泛滥成灾。

在百度,算法的应用是融入到研发部门每个人的工作中的。在这里,不是经理告诉工程师做什么,怎么做,用什么算法,而是需要工程师自己在某个领域去发现问题,提出算法,评估效果,并不断改进。这要求每个工程师在算法上的基本功很强,并能灵活的加以应用,以解决实际问题。现在,百度有不少的程序员,他们大部分的时间是用在发现问题,分析问题,思考解决问题的方法上。实际编写代码所花的时间并不多。

有不少人觉得,现在的搜索引擎已经足够好了,算法上没有太多改进余地了。我不赞成这个观点。虽然每次调查的数据显示,超过90%的人对搜索引擎提供的服务表示满意,但是第一次搜索就能找到满意结果的用户只有50%左右,很多用户都是在多次更换关键词之后才搜索到自己想要的结果。这说明我们还有巨大的改进空间。

可以说,搜索引擎开发中使用的基本算法大部分都在大学课程中涵盖了。对于一个人来说,在学校学习过这个算法,和能够灵活运用是两个概念。只有通过参与较多的项目开发和程序编写,将算法和应用相结合,才能在这方面得到较好的发展。

对于算法学习,我的建议还是多思考,多做项目和程序。在做的过程中肯定会遇到一些问题,这是正常的。好的程序员善于从问题和失败中学到东西,举一反三,设法避免以后出现同样或类似的错误。另外,还要善于从别人身上学习,有意识的进行思考和总结,这是比较有效的方式。■

周利民,现任百度公司首席架构师。

迈进算法世界的大门

□ 文 / 潘文斌

某些程序员觉得算法很神秘，是个遥不可及，平时根本用不到的东西。其实不然，算法在每个人平时编程时都会用到。简单来讲，算法其实就是一个定义好的计算步骤，这个步骤接受一些数据的输入，通过计算，给出一些数据的输出。也就是说，算法是把输入转变成输出的过程。这个概念和程序的概念很像，不同的是，一个算法通常可以用好几种方式来实现。

如果你想要编程实现从一个排好序的数列中找到一个特定的数。这时你可以决定从头到尾的遍历数列，或者每次都看数列里中间那个数，以决定下一步在数列的前半或者后半中查找。这就是两个不同的算法的例子。从这个例子里我们可以看出，算法的具体实现还要依赖于数据结构。如果数列存储在一个数组里面，那么我们编程实现就会很方便，因为可以直接访问数列中点位置的数。但是，如果数列是存储在链表的结构里，实现起来就会很麻烦，编出来的程序也会跟用数组存储的程序有很大不同，尽管这是对同一个算法的实现。算法和数据结构都决定了，就可以编程解决问题了，所以有“算法＋数据结构＝程序”的说法。

对于每一个输入，算法要能给出预期的输出。如果一个算法只能针对某些输入给出正确的结果，那它是不能正确解决问题的。而我们通常还会用时间复杂度和空间复杂度两项来衡量算法的效率。值得注意的是，分析时间和空间复杂度的时候，我们主要看的是平均情况和最差情况所需要的时间空间。

需要解决的问题有很多类型，每种类型的问题也都有很多针对性很高的算法。但是，算法的思想是通用的。下面我们就来讲讲几种最基础的算法思想。

分治法

分治法是最常见，也是最简单的算法。它的基本思想是如果一个问题本身很复杂，但是它可以拆分成几个简单的或者已经解决过的小一点的问题的话，我们可以先解决拆分后的小问题，然后把得到的结果合并起来作为原问题的结果。分治法是个很好的能够化繁为简的方法，但是它也要求满足两个要素。一是需要解决的问题能够分解成小的问题，二是小问题的结果可以方便而且正确的合并成需要解决问题的结果。第二个要素往往是分析算法正确性的关键。

在实际应用的时候，分治法往往和递归结合起来。待解决的问题通常被拆分成两个和待解决的问题类型相同，但是需要处理的输出比较小的问题。这样使用递归就可以将问题进一步化简，直到可以解决的地步。

下面的问题就是把小数组正确合并的问题，也就是如何将两个排好序的数组合并成一个排好序的大数组的问题。这个过程有点像洗牌。比较两个数组中最小的元素，把比较小的那个作为新数组中最小的元素放入新数组。然后比较两个数组剩余的元素里最小的元素，这样做直到所有的元素都被放入新数组为止。整个算法的流程是：待排序的大数组 - >待排序的两个小数组 - >待排序的四个更小的数组 - >…… - >n个只有一个元素的数组 - >排好序的n / 2 个数组 - >… - >排好序的两个数组 - >排好序的一个数组。

这个数组排序的算法叫做归并排序（Merge - Sort），它严格按照分治法的流程：拆分 - >解决 - >合并来运行。拆分的步骤是把一个大数组拆分成两个小数组；解决的步骤是分别对两个小数组进行排序（递归的）；合并的步骤是将两个排好序的小数组合并成排好序的大数组。下面是它的伪代码表示，其中Merge - Sort (A,p,r)是对数组A的子数组A[p..r]进行归并排序，Merge (A,p,q,r)是对两个子数组A[p,q]和A[q+1,r]进行归并。为了对整个数组进行排序，只需要运行Merge - Sort(A,1.sizeof A)就可以了。

```
Merge - Sort (A,p,r)
1 if p<r
2    then q <- (p+r)/2
3       Merge - Sort (A,p,q)
4       Merge - Sort (A,q+1,r)
5       Merge (A,p,q,r)
```

很显然合并过程是可以在O(n)时间内完成的，因为对于每个放入新数组

的元素，合并过程只进行了一次比较和读写。假设T(n)是完成整个归并排序需要的时间，由上面的伪代码可以看出，T(n)=2T(n/2)+O(n)，且T(1)=O(1)。解这个等式可以得到，归并排序算法的时间效率是O(nlgn)。比起插入排序的$O(n^2)$的时间复杂度，归并排序要快得多。

动态规划

动态规划算法实际上是分治法的延伸。有的时候，分治法在递归过程中可能会导致重复的子问题出现。比如，在将一个大数组A[p,r]分解的过程中，由于拆分和合并的难度问题可能没办法拆分成两个完全不重叠的子数组A[p,q]和A[q+1,r]，只能拆分成有部分重叠的两个子数组A[p,r-1]和A[p+1,r]。这样的情况下分治法还是可以得到正确的解，但是，考虑拆分的下一步，两个子数组都将拆分出一个A[p+1,r-1]的子数组。这样在解决子问题的时候，这个子数组将被计算两遍。而且由于递归的影响，拆分的层次越多，子问题就要被重复计算越多的次数，而且是成指数级增长，严重影响算法的效率。

解决这个问题的方法也很简单，我们只需要保证这些重复计算只计算一次就可以了。为了保证这个，我们需要维护一张表。这张表里保存每个需要重复计算的问题的结果。当我们第一次计算过一个子问题并把它的结果保存在表里以后，再次碰到这个子问题就可以直接查表结果以代替重复计算了，这就是动态规划的思想。

贪心算法

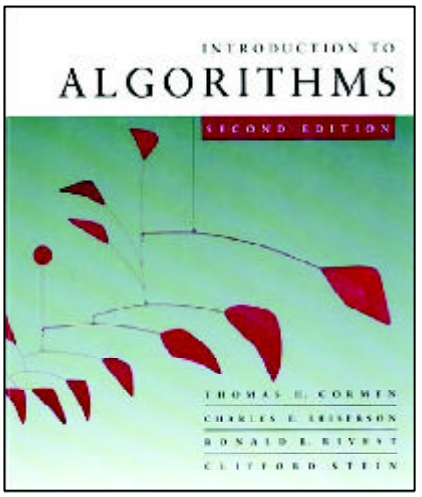
贪心算法是一种很容易理解但又很“短视”的算法，它一般被用在求最优解的问题上。贪心算法的思想是每次只考虑一步，每次都去用当前看起来最好的选择，使得当前的情况是最优的，而不去考虑这个选择对将来的影响。这种算法虽然简单，但局限性很大。它不一定能找到问题的最优解，但是一般也可以找到一个比较优化的解。

日程安排是个典型的可以用贪心算法解决的问题。假设有n个需要一段连续时间的活动，每个活动i都有一个起始时间s_i和一个结束时间f_i。如果两个活动i和j满足$[s_i, f_i)\cap[s_j, f_j)=\Phi$，那么说明这两个活动是不重叠的。问题的目标是安排一个日程表，使得日程表上有最多数量的活动，并且每两个活动都是互不重叠的。

如果以贪心的角度来考虑问题的话，自然日程表上第一个活动会选择结束时间最早的那个活动，这样一天可以剩下更长的时间安排行程。选择了第一个活动以后，所有跟这个时间有冲突的活动自然都需要排除。然后再从剩余的活动里面挑选结束时间最早的活动。这样重复下去直到没有活动可以选择为止，我们就得到了一张日程表。

这个日程表上面显然每两个活动都是不重叠的，幸运的是，日程表上的活动数也是最多的，贪心算法得到的是一个最优解。我们可以这样证明，假设贪心算法给出的日程表是$i_1<i_2<\cdots<i_k$，而某个最优解是$j_1<j_2<\cdots<j_l$。从贪心算法的过程我们可以看出，i_1是第一个结束的活动，所以i1的结束时间一定比j_1要早。这样，把最优解中的j_1用i_1替换掉，我们仍旧得到一个不冲突的日程表，而且这个日程表上活动的数目跟没替换之前是一样多的，所以新日程表仍旧是一个最优解。在新的日程表的，j_2的结束时间显然又比i_2的晚，所以j_2也可以被i_2替换掉，得到另一个新的最优解。这样递归推理下去，最优解日程表中的所有活动都可以被贪心算法给的日程表中相应的活动替换，所以在日程安排这个问题上，贪心算法得到的是最优解。

但是，不是所有求最优解的问题都能用贪心算法解决的，最著名的例子是背包问题。这个问题有一个已知容量的背包以及一些货物，目标是在背包里放进尽可能重又不超过背包容量的东西。例如已知背包容量是100kg，另有三件分别是70kg，50kg，40kg的物品。如果用贪心算法的话，自然装第一件东西时就想装进最重的那件，此时背包以不能容纳其他物品，装载的物品共为70kg。这显然不是最优解，最优解应该是放弃最重的物品的50kg+40kg=90kg。



这三种最基本的算法思想是很多算法领域的基础。分治法的适用性很好，在搜索，排序，计算几何，存储优化等方面都有应用。动态规划一般用在序列处理中，包括字符串处理，文本处理，以及计算生物等等。贪心算法通常用来求最优解，或者在一些NP问题里求近似最优解。如果想对算法有进一步的研究的话，可以参考一本叫Introduction to Algorithms (Second Edition, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein)的书，这本书是算法入门的宝典。

本文作者潘文斌，美国杜克大学毕业，目前在IBM中国软件开发中心工作。

走近算法

——从三道题目看问题求解

□ 文 / 刘汝佳 整理 / 吴莹莹

对于崇尚程序的人来说，“算法”是一个神圣的词语，它与数学、程序设计紧密相关。另一个和算法密切相关的词是问题求解(Problem Solving)，市面上已经有一些书籍把它作为标题的一部分——尽管这些书和讨论Java/C#的书籍相比是沧海一粟。

对于一个熟练得几乎可以闭着眼睛写出工作中需要完成的大部分代码的程序员来说，问题求解意味着什么？对于一个认为程序无所不能的人来说，这是一个不可抗拒的挑战——你敢编程解决真正的难题么？

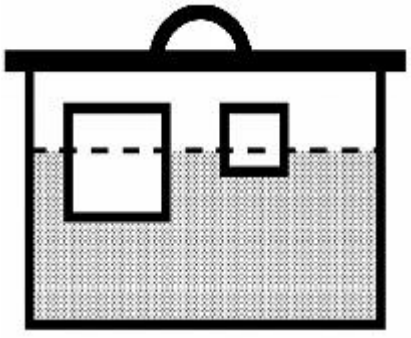
本文尝试着通过几道题目为读者带来对于问题求解的感性认识，希望能够为大家带来一些深入探索算法艺术的思考，更加深入地了解程序设计的核心。

让我们从一道看似简单的题目开始。几乎所有程序员都可以写出解决它的程序，但是完全没有接触过算法的人却难以写出完整而优美的程序。

问题1：在桶里注入密度为1的水，然后放入大小不一的立方体，最后盖一个盖子并压紧。给出桶底面积S，高H，水的体积V以及n个立方体的边长和密度，求最后的水位高度。已知各个立方体的底面互不重叠。

这道题目很直观，可是当你动手编程时可能会发现程序并不是那么容易写。立方体可能沉底，可能漂浮、悬浮，还可能顶住桶盖，而且这些状态可能随着水位的改变而相互转换！显然，如果用计算机“模拟注水”来计算水位非常容易写错。我们不妨反过来想：如果已知水位h，能否计算出水的体积呢？不仅可行，而且相当容易：根据立方体的密度可以计算出它在正常情况埋在水里的深度h'。如果现在的水位还不到h'，那么埋在水里的深度为h'；如果水位过高导致立方体顶住桶盖，则也可以根据水位离桶顶的高度计算出实际埋在水里的深度。不管哪种情况，每个立方体实际埋在水里的深度是惟一确定的，水的体积也就很容易算出了。

如果把水位h和水的体积V的关系写成一元函数V=f(h)，则h越大，V越大。这样的函数称为单调函数。我们已经知道可以很方便的用h来计算V，那么在算法设计中有一种称为二分搜索(binary search)的策略反过来从V计算h，只要V和h的函数关系的单调的。



尽管这个问题的条件很多，但是与最终结果相关的条件却只有一个，通过一些数学上基础的方法，就可以很快简化问题。通常，我们称这种思想为“反客为主”。

在更多的时候，解决问题时需要的不仅是算法思想，还需要算法知识。

问题2：有n个智能程序将参加淘汰赛。淘汰赛赛程设置如下：每次选择两个没有被淘汰的程序进行比赛，胜利的程序留下，失败的淘汰，比赛没有平局。比赛一直进行到只剩下一个程序，这个程序就是冠军。如果在以前的历史记录中，程序A战胜了程序B，那么在这次比赛中A便一定能够战胜B。如果在以前的历史记录中，程序A和程序B之间没有比赛，那么在这次比赛中，既有可能A战胜B，也有可能B战胜A。因此合理的安排淘汰赛程可能会使某个程序取得冠军。给出你程序以前的比赛纪录m条（形如“A曾经战胜B”），判断哪些程序有可能获得冠军。

和刚才的题目不一样，这道题目需要思考一下，而不容易立刻得到一个“看起来似乎能用”的算法。也许你会有一些思路，甚至你会觉得你已经“快把题目做出来了”，但如果没有图的知识，很难把这些思路用一种清晰的方式表示出来，

即使这个思路是正确的。如果学习过图，你会很自然的把程序之间的关系用图来表示——有向边(u，v)表示程序u曾经战胜过程序v。如果u和v没有比赛过，那么用两条有向边(u，v)和(v，u)表示。这样，我们得到了一个有n2-m条边的图。那么可以得出：一个程序a可能夺冠当且仅当从a出发可以到达其他每个点(证明留给读者)。这个结论并不是很难得到——也许你在看到这里之前已经用另外一种方式叙述了一遍：一个程序要想夺冠必须能直接或间接的打败其他所有程序。

这样，我们的第一个算法产生了：从每个点出发进行一次图遍历，就可以判断出它是否可以到达其他每个点。这样做是正确的，但时间效率太差。算法分析得出：这个算法的时间复杂度为$\Theta(n^3)$。事实上，原题中规定n不超过100000，m不超过1000000，在极限情况下用这个算法写出的程序至少要算好几个月。

既然用图来给本题建模，图论算法知识可以帮助我们降低算法的时间复杂度。在讨论有向图的连通性时，我们往往会考虑强连通分量（强连通分量的直观含义是“相互可达的点集”）。所有强连通分量之间的可达关系组成一个有向无环图，则在这个图中，存在指向所有其他连通分量的连通分量中所有点均为所求。如果学过求强连通分量的线性时间算法，则这个算法的总时间复杂度降为$\Theta(n^2)$，比刚才的算法快了很多，但仍要计算十分钟以上的时间。

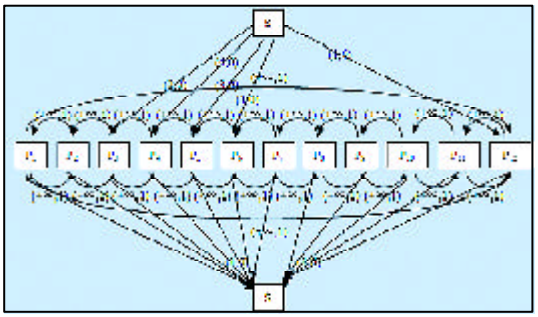
能否更快点呢？这一步需要一点创造性思维。考虑打过的比赛次数最少的一个程序，则没有和它打过比赛的程序和它应该属于同一个连通分量。这一点不需要求连通分量就可以得到了。根据抽屉原理，仅增加这一次初始合并后，剩下最多2m/n个点。这样处理以后，在一台典型配置的个人电脑上，这个程序的运行时间不会超过1秒。

尽管还存在更加优秀的算法，但是我们仍然只需要利用一些离散数学的基础知识就可以轻松设计出解决问题的算法，因此数学作为计算机的一门基础学科是相当重要的。

在算法学习中，经典问题的模型和算法也是一个很重要的方面。下面举一个例子。

问题3：n个盒子被放成一圈，每个盒子按顺时针编号为1到n，每个盒子里都有一些球，且所有盒子里球的总数不超过n。这些球要按如下的方式转移：每一步可以将一个球从盒子中取出，放入一个相邻的盒子中。目标是用尽量少的移动使所有的盒子中球的个数都不超过1。

这个问题的解法很多，但要么思维难度太大要么时间效率不高。对于有一定算法素养的人来说，恐怕最容易想到的“保证正确”的算法是套用一个经典问题：最小费用最大流问题。



首先，将每个盒子看成图上的一个点，并设原点和汇点S，T。如图所示，从原点向每个有球的盒子引一条有向边，容量为盒子内的球数，费用为0。而每个盒子向T点引一条有向边，容量仅为1，费用为0。

然后对每一个盒子Pi，分别向前一个和后一个相邻的盒子Prev(Pi)和Next(Pi)各连一条有向边，容量为正无穷，费用为1，表示一个球若从一个盒子转移到相邻的盒子，需要1步。

显然，在这样构造出来的图求最小费用最大流，一定可以得到一个可行解，并且最小的费用就是最少需要的步数。

对于熟悉最小费用最大流问题的读者来说，这样的算法显然是正确的，但实现起来并不是很简单。幸运的是，可以把这个模型做为中转，而设计出更容易理解和实现贪心算法（留给读者思考）。它的时间复杂度为$O(n^2)$，对于本题来说已经是相当理想了。

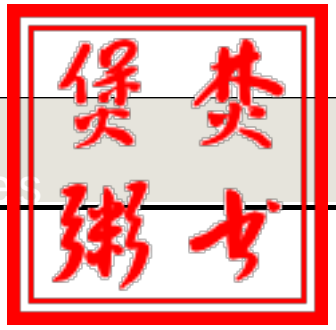
尽管目前在很多工具和库中都已经有对算法做好封装的代码可供使用，但是大多数时候，实际问题是无法用这些封装好的包来帮你完成的。如果能在原理上对算法有一个清晰的认识，实际工作中将会体现出它应有的价值。

作者简介

刘汝佳，2002年摘得ACM总决赛银牌，连续五年担任中国信息学奥林匹克国家集训队教练。著有《算法艺术与信息学竞赛》一书。

编者按

对于学生而言，算法是一门难度很大、非常枯燥且没有实际价值的课程。然而它在软件世界中如此重要，以至于每一个大学的计算机课程中都包含了这样一门功课。在掌握算法设计之前，人们往往都要熟悉很多数学、计算机体系结构等方面的基础知识，另外一方面，学生很少碰到真正程序员在工作中遇到的实际问题，因此一些机构组织主动开设了算法相关的竞赛，以求达到培养学生算法设计能力的目标。本文的作者在此方面有所专长，希望能让您读过本文后，对算法竞赛以及算法产生一点兴趣，并投入其中。限于篇幅，本文的五个示例裁减为三个，文章全文请参见Csdn网站杂志频道。



大话“20世纪10大算法”

□ 整理 / 柯化成

记得小时候看过一套连环画《说唐演义》，故事情节其实已经完全记不得了，但书里头有个《英雄谱》至今记忆犹新，第一名西府赵王李元霸，胯下千里一盏灯，掌中一对擂鼓瓮金锤；第二名天宝大将宇文成都，胯下赛龙五斑驹，掌中凤翅镏金镗；第三名，银锤太保裴元庆裴三公子……如此这般一直排到第十八条好汉单雄信——中国人真是自古就有好排座次的习俗，今天社会上各种各样的排行榜更是五花八门——从大学排名到超级女生，再到饕餮美食、流行音乐，名车美酒，凡此种种，不一而足。程序员们不禁要问，算法有排行榜么？

有，还真有！不过，还倒不是中国人评出来的：

本世纪初，美国物理学会(American Institute of Physics)和IEEE计算机社团 (IEEE Computer Society)的一本联合刊物《科学与工程中的计算》发表了由田纳西大学 的Jack Dongarra和橡树岭国家实验室的Francis Sullivan联名撰写的“世纪十大算法”一文，该文“试图整理出在20世纪对科学和工程领域的发展产生最大影响力的十大算法”。作者苦于“任何选择都将是充满争议的，因为实在是没有最好的算法”，他们只好用编年顺序依次列出了这十项算法领域人类智慧的巅峰之作——给出了一份没有排名的算法排行榜。有趣的是，该期杂志还专门邀请了这些算法相关领域的“大拿”为这十大算法撰写十篇综述文章，实在是蔚为壮观。本文的目的，便是要带领读者走马观花，一同回顾当年这一算法界的盛举。

1946 蒙特卡洛方法

在广场上画一个边长一米的正方形，在正方形内部随意用粉笔画一个不规则的形状，呃，能帮我算算这个不规则图形的面积么？蒙特卡洛(Monte Carlo)方法便是解决这个问题的巧妙方法:随机向该正方形内扔N（N是一个很大的自然数）个黄豆，随后数数有多少个黄豆在这个不规则几何形状内部，比如说有M个:那么，这个奇怪形状的面积便近似于M/N，N越大，算出来的值便越精确。别小看这个数黄豆的笨办法，大到国家的民意测验，小到中子的移动轨迹，从金融市场的风险分析，到军事演习的沙盘推演，蒙特卡洛方法无处不在背后发挥着它的神奇威力。

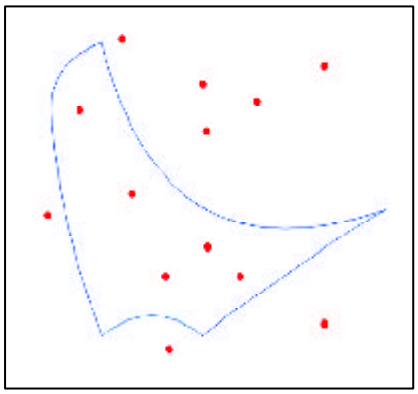


图 1:蒙特卡洛方法求不规则形状面积

蒙特卡洛方法由美国拉斯阿莫斯国家实验室的三位科学家John von Neumann（看清楚了，这位可是冯诺伊曼同志！），Stan Ulam和 Nick Metropolis共同发明。就其本质而言，蒙特卡洛方法是用类似于物理实验的近似方法求解问题，它的魔力在于，对于那些规模极大的问题，求解难度随着问题的维数（自变量个数）的增加呈指数级别增长，出现所谓的“维数的灾难”（Course of Dimensionality）。对此，传统方法无能为力，而蒙特卡洛方法却可以独辟蹊径，基于随机仿真的过程给出近似的结果。

最后八卦一下，Monte Carlo这个名字是怎么来的？它是摩纳哥的一座以博彩业闻名的城市，赌博其实是门概率的高深学问，不是么？

1947 单纯形法

单纯形法是由大名鼎鼎的“预测未来”的兰德公司的Grorge Dantzig发明的，它成为线性规划学科的重要基石。所谓线性规划，简单的说，就是给定一组线性（所有变量都是一次幂）约束条件（例如$a_1*x_1+b_1*x_2+c_1*x_3>0$），求一个给定的目标函数的极值。这么说似乎也太太太抽象了，但在现实中能派上用场的例子可不罕见——比如对于一个公司而言，其能够投入生产的人力物力有限（“线性约束条件”），而公司的目标是利润最大化（“目标函数取最大值”），看，线性规划并不抽象吧！线性规划作为运筹学(operation research)的一部分，成为管理科学领域的一种重要工具。而Dantzig提出的单纯形法便是求解类似线性规划问题的一个极其有效的方法，说来惭愧，本科二年级的时候笔者也学过一学期的运筹学，现在脑子里能想起的居然只剩下单纯形法了——不过这不也正说明了该方法的简单和直观么？

顺便说句题外话，写过《万历十五年》的黄仁宇曾说中国的传统是“不能从数目字上管理”，我们习惯于“拍脑袋”，而不是基于严格的数据做决定，也许改变这一传统的方法之一就是全民动员学习线性规划喔。

$$\begin{pmatrix} a_{11}^{(0)} & a_{12}^{(0)} & \cdots & a_{1,n-m}^{(0)} \\ a_{21}^{(0)} & a_{22}^{(0)} & \cdots & a_{2,n-m}^{(0)} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1}^{(0)} & a_{m2}^{(0)} & \cdots & a_{m,n-m}^{(0)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \\ x_{n-m} \end{pmatrix} = \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \\ b_m^{(0)} \end{pmatrix}$$

图 2:线性规划问题

1950 Krylov子空间迭代法
1951 矩阵计算的分解方法

50年代初的这两个算法都是关于线性代数中的矩阵计算的，看到数学就头大的读者恐怕看到算法的名字已经开始皱眉毛了。Krylov子空间叠代法是用来求解形如Ax=b的方程，A是一个n*n的矩阵，当n充分大时，直接计算变得非常困难，而Krylov方法则巧妙地将其变为$Kx_{i+1}=Kx_i+b-Ax_i$的迭代形式来求解。这里的K(来源于作者俄国人Nikolai Krylov姓氏的首字母)是一个构造出来的接近于A的矩阵，而迭代形式的算法的妙处在于，它将复杂问题化简为阶段性的易于计算的子步骤。

1951年由橡树岭国家实验室的Alston Householder提出的矩阵计算的分解方法，则证明了任何矩阵都可以分解为三角、对角、正交和其他特殊形式的矩阵，该算法的意义使得开发灵活的矩阵计算软件包成为可能。

1957 优化的Fortran编译器

说实话，在这份学术气息无比浓郁的榜单里突然冒出一个编译器(Compiler)如此工程化的东东实在让人有“关公战秦琼”的感觉。不过换个角度想想，Fortran这一门几乎为科学计算度身定制的编程语言对于科学家（尤其是数学家，物理学家）们实在是太重要了，简直是他们形影不离的一把瑞士军刀，这也难怪他们纷纷抢着要把票投给了它。要知道，Fortran是第一种能将数学公式转化为计算机程序的高级语言，它的诞生使得科学家们真正开始利用计算机作为计算工具为他们的研究服务，这是计算机应用技术的一个里程碑级别的贡献。

话说回来，当年这帮开发Fortran的家伙真是天才——只用23500行汇编指令就完成了一个Fortran编译器，而且其效率之高令人叹为观止：当年在IBM主持这一项目的负责人John Backus在数十年后，回首这段往事的时候也感慨，说它生成代码的效率“出乎了所有开发者的想象”。看来作为程序员，自己写的程序跑起来“出乎自己的想象”，有时候还真不一定是件坏事！

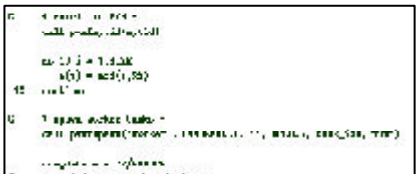


图 3:Fortran语言片段

1959-61 计算矩阵特征值的QR算法

呼，又是一个和线性代数有关的算法，学过线性代数的应该还记得“矩阵的特征值”吧？计算特征值是矩阵计算的最核心内容之一，传统的求解方案涉及到高次方程求根，当问题规模大的时候

十分困难。QR算法把矩阵分解成一个正交矩阵（什么是正交矩阵？！还是赶紧去翻书吧！）与一个上三角矩阵的积，和前面提到的 Krylov 方法类似，这又是一个迭代算法，它把复杂的高次方程求根问题化简为阶段性的易于计算的子步骤，使得用计算机求解大规模矩阵特征值成为可能。这个算法的作者是来自英国伦敦的J.G.F. Francis。

1962 快速排序算法

不少读者恐怕和我一样，看到“快速排序算法”（Quick Sort）这个条目时，心里的感觉是——“这可总算找到组织了”。相比于其他一些对程序员而言高深莫测的数学物理公式，快速排序算法真是我们朝夕相处的好伙伴——老板让你写个排序算法，如果你写出来的不是快速排序，你都不好意思跟同事打招呼。其实根本不用自己动手实现，不论是ANSI C，C++ STL，还是Java SDK，天下几乎所有的SDK里都能找到它的某种实现版本。

快速排序算法最早由Tony Hoare爵士设计，它的基本思想是将待排序列分为两半，左边的一半总是“小的”，右边的一半总是“大的”，这一过程不断递归持续下去，直到整个序列有序。说起这位Tony Hoare爵士，快速排序算法其实只是他不经意间的小小发现而已，他对于计算机贡献主要包括形式化方法理论，以及ALGOL60编程语言的发明等，他也因这些成就获得1980年图灵奖。

快速排序的平均时间复杂度仅仅为O(Nlog(N))，相比于普通选择排序和冒泡排序等而言，实在是历史性的创举。

1965 快速傅立叶变换

如果要评选对我们的日常生活影响最大的算法，快速傅立叶变换算法应该是当仁不让的总冠军——每天当拿起话筒，打开手机，听mp3，看DVD，用DC拍照——毫不夸张的说，哪里有数字信号处理，哪里就有快速傅立叶变换。快速傅立叶算法是离散傅立叶算法（这可是数字信号处理的基石）的一种快速算法，它有IBM华生研究院的James Cooley和普林斯顿大学的John Tukey共同提出，其时间复杂度仅为O(Nlog(N))；比时间效率更为重要的是，快速傅立叶算法非常容易用硬件实现，因此它在电子技术领域得到极其广泛的应用。

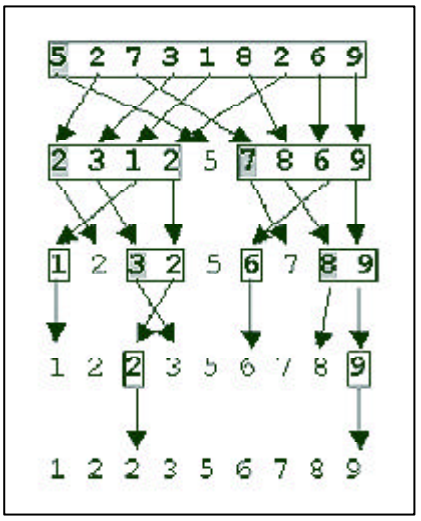


图4：快速排序算法

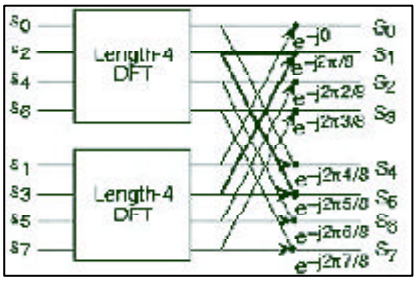


图5：快速傅立叶算法的蝶型变换

1977 整数关系探测算法

整数关系探测是个古老的问题，其历史甚至可以追溯到欧几里德的时代。具体的说：

给定一组实数$X_1, X_2, \ldots, X_n$，是否存在不全为零的整数$a_1, a_2, \ldots a_n$，使得：$a_1x_1 + a_2x_2 + \ldots + a_nx_n = 0$ 这一年BrighamYoung大学的Helaman Ferguson 和Rodney Forcade解决了这一问题。至于这个算法的意义嘛，呃，该算法应用于“简化量子场论中的Feynman图的计算”——太深奥的学问拉！

1987 快速多极算法

日历翻到了1987年，这一年的算法似乎更加玄奥了，耶鲁大学的Leslie Greengard和Vladimir Rokhlin提出的快速多极算法用来计算“经由引力或静电力相互作用的N个粒子运动的精确计算——例如银河系中的星体，或者蛋白质中的原子间的相互作用”，天哪，不是我不明白，这世界真是变得快！

所谓浪花淘尽英雄，这些算法的发明者许多已经驾鹤西去。二十一世纪的头五年也已经在不知不觉中从我们指尖滑过，不知下一次十大算法评选的盛事何时再有，也许我们那时已经垂垂老去，也许我们早已不在人世，只是心中唯一的希望——里面该有个中国人的名字吧！

参考文献

本文基于下列材料：

[1] Dongarra and Sullivan, Top Ten Algorithms of the Century, Computing in Science and Engineering, January/February 2000.

[2] Barry Cipra, The Best of the 20th Century: Editors Name Top 10 Algorithms SIAM News Volume 33, Number 4, May 2000, page 1.

作者简介

柯化成，今年3月毕业于浙江大学计算机学院，4月加盟Google中国工程研究院，担任软件工程师。使用C++语言和Python语言编程，关注设计模式和软件体系结构。