每个 CS 系学生都应该知道的事 - 文章 - 伯乐在线

考虑到计算机科学领域的膨胀增长，想要辨识现代计算机科学到底包含什么，成了一件有挑战性的事。我们系进行了这个讨论， 所以我整合一下自己的想法来当作这个问题的解答，“每个 CS 系的学生应当知道哪些事？”

我尝试从 4 方面来回答这个问题：

- 学生想要获得好的工作应当知道哪些事？
- 学生想要得到终生雇佣应当知道那些事？
- 学生想要进入研究生院应当知道哪些事？
- 学生想要有益于社会应当知道那些事？

下面我会把自己的想法分为现代计算机领域的一般性的原则和一些特别推荐两部分来写。

计算机系的学生：把本文当作自学指南随意使用。
增删的建议请邮件或推特联系我。

作品集 portfolio ，而不是简历

自从计算机科学从工程学和数学分离出来之后，计算机程序行业就开始依靠简历来雇佣毕业生。

一份简历无法说明程序员的能力。

每一个计算机系的的学生都应当有其作品集。

一个作品集可以简单到是一个个人博客，上面有工程或实现的帖子。更好些的话每个工程的有其单独页面和可供公共浏览的代码（也许托管到 Github 或者 Google Code 上）。

对开源代码的贡献应当给出链接和说明。

一个代码作品集能够使雇主直接评价雇员的能力。

而 GPA 和简历却做不到。

教授应该设计课题来使作品集更出彩，学生在课程结束时应该花些时间更新这些课程项目。

技术交流

在计算机科学界“独狼”已然成为濒危物种。

当代计算机科学家必须练习与非程序员清晰且有说服力地交流自己的想法。

在小公司，程序员能否和管理层交流她的想法能够影响到公司的成败。

不幸的是，单独增加一个课程并不能有什么改变（当然一个合理的科技交流课程没有坏处）。

应当提供给学生更多的机会来给予他们通过口头讲演的方式展示自己工作和想法。

特别推荐

我建议学生掌握一种演示工具，比如说 PowerPoint 或者（我最喜欢的）KeyNote。（抱歉，尽管我喜爱基于 LaTex 的演示工具，它们还是太静态了）。

不过，要是想生成漂亮的数学文档，LaTex 是无可比拟的。所有的科技课程的写作作业都应该以 LaTex 的形式提交。

一颗工程学的心

计算机科学不是完全的工程学。

但也差不多。

计算机科学家终会发现他们和工程师在一起工作。计算机科学家和传统的工程师需要说相同的语言——一种扎根于实分析、线性代数、概率论与物理学的语言。

计算机科学家理应掌握物理学中的电磁学，但要达到这一点，他们还需掌握多元微积分，（外加学习微

分方程）。

在进行声音仿真时，精通概率论（通常还包括）线性代数是极有益处的。在说明计算结果时，对统计的牢固理解是无可替代的。

Unix 哲学

计算机科学家应当习惯并且熟练使用 Unix 哲学的处理。

Unix 哲学（不同于 Unix 本身）是一种注重语言学抽象和整合来达到预期处理的方法。

在实践中，这意味着要习惯于命令行形式处理、文本文件进行配置和轻型IDE的软件开发。

特别推荐

考虑到 Unix 系统的流行度，当今的计算机科学家应当熟练地掌握基本的 Unix 能力：

- 浏览和操作文件系统
- 使用管道进行组合操作
- 习惯于使用 `emacs` 和 `vim` 编辑文件
- 新建、修改和运行一个软件项目的 Makefile 文件
- 编写简单的 shell 脚本 学生在不理解 Unix 哲学强大能力时会抵制它。此时最好让学生尝试完成一些 Unix 有相对优势的有用的任务，比如：
- 找到指定目录下占用空间最大的5个文件夹
- 找到计算机中重复的 MP3 文件（相同的文件内容而不是文件名）
- 找到名字列表中姓名首字母是小写的名字，并调整大小写
- 找到第二个字母是 `x`，倒数第二个是 `n` 的英语单词
- 把你的手机的声音输入经由网络传送到另一台电脑的音响播放
- 把指定文件夹下的文件名中的空格替换为下划线
- 报告指定 IP 地址接入 web 服务器的最近十个错误连接

系统管理

一些计算机科学家嘲笑系统管理是一件“IT”任务。

他们的想法是可以自学技术人员能做得到的所有事。

这是正确的（嗯，理论上是）。

然而计算机科学家能够完全且安全地控制他们的系统和网络的态度是有些误导人的。

软件开发中很多任务不传给系统管理员来做是最高效的。

特别推荐

每个当代的计算机科学家应当能够：

- 安装和管理一个 Linux 发行版
- 配置和编译 Linux 内核
- 使用 `dig`、`ping` 和 `traceroute` 命令来排解故障
- 编译和配置 web 服务器，比如 apache
- 编译和配置 DNS 守护进程，比如 bind
- 使用文本编辑器维护一个站点

编程语言

编程语言有周期的兴起与衰落。

而一个程序员的职业不应如此。

尽管教授与获得工作相关的语言很重要，学生能够自学新的编程语言也同等重要。

学习怎样学习新的编程语言的最好方式是学习多种编程语言和编程范式。

学习第n个语言的难度是第(n - 1)个的一半。

然而，要想真正理解编程语言，应该自己实现一个。理想情况下，每个计算机科学系的学生都参加过编译的课程。至少，每个学生应该实现一个解释器。

一些语言

下面的编程语言涵盖了编程范式和实际应用：

- Racket
- C
- Javascript
- Squeak
- Java
- Standard ML
- Prolog
- Scala
- Haskell
- C++ 和
- 汇编

Racket

Racket，作为功能全面的 Lisp 的方言，有着极简单的语法。

对少部分的学生来说，这种语法是一种学习障碍。

不过坦率地讲，如若一个学生觉得即使是暂时接受一种相异的语法规则也是很大的脑力障碍的话，他缺乏从事计算机科学职业的灵巧心智。

Racket 丰富的宏系统和高阶编程组件彻底打破了数据和代码的分别。

如果教的合理，能够充分发挥 Lisp 的能力。

建议阅读

ANSI C

C 是对底层（硅）的简洁至极的抽象。

C 在嵌入式系统的编程中无可替代。

学习 C 能提供对冯·诺依曼体系的深入理解，其程度没有其他语言能匹拟。

考虑到差的 C 编码与普遍的缓冲区溢出安全隐患有着亲密的关系，程序员学习正确地编写 C 程序是很重要的。

建议阅读

Javascript

Javascript是动态、高级语言比如 Python、Ruby 和 Perl 的语义模型的很好的一个代表。

作为 web 原生语言，它的实用性优势是独一无二的。

建议阅读

Squeak

Squeak 是最纯正的面向对象语言 Smalltalk 的现代方言，它展现了“面向对象”的本质。

建议阅读

Java

Java 将保持流行久到无法将其忽略。

建议阅读

Standard ML

Standard ML 是 Hindley-Milner 系统的一个干净实现。

Hindley-Milner 类型系统是现代计算计算机领域最伟大（然而却是最不知名）的成就。

尽管有着指数级的复杂性，Hindley-Milner 的类型推断对于正常的程序来说是足够快的。

类型系统支持复杂的结构化不变量表达，事实上，它丰富到类型定义良好的程序经常是没有 bug 的。

建议阅读

Prolog

尽管在应用上占有一席之地，逻辑编程是计算思维的另一种范式。

在程序员需要在其他编程范式里模拟逻辑编程时，理解逻辑编程是值得的。

另一种值得学习的逻辑编程语言是miniKanren。miniKanren强调纯粹的逻辑编程。这个约束逐步形成了另一种风格的逻辑编程称为关系程序设计，并且它授予通常Prolog程序不支持的属性。

建议阅读

Scala

Scala 是定义良好的函数式与面向对象的融合语言。

Scala 是 Java 应该做到的样子。

建立于 Java 虚拟机之上，并兼容现存的 Java 代码库，Scala 最有可能成为 Java 的后继者。

建议阅读

Haskell

Haskell 是 Hindley-Milner 语言家族的王冠。

充分利用惰性求值，Haskell 是主流编程语言中最接近于纯数学的。

建议阅读

标准 C++

C++ 是无法避免的灾祸。

但是既然必须要教 C++，那就教全。

特别地，计算机科学系的学生毕业时应该掌握模板元编程.

建议阅读

汇编

任何汇编语言都行。

既然 x86 很流行，最好学它。

学习编译器的最好方式便是学习汇编，因为汇编直观地展示了将高级代码转化为低级代码。

特别推荐

计算机科学家应该理解产生式编程（宏编程）；词法（动态）范围；闭包；continuation；高阶函数；动态调度；子类型；模块和函子还有不同于其他特定语法的 monads 语义概念。

建议阅读

离散数学

计算机科学家必须要对形式逻辑及其证明有牢固的理解。代数操作和自然推理证明是处理例程任务的有力方法，归纳总结证明在构建递归函数时很有用处。

计算机科学家必须对形式数学记号很熟悉，并且对基本的离散数学结构－集合、元组、队列、方法和幂集能进行的严格推理。

建议阅读

对于计算机科学家，掌握这些理论很重要：

- 树；
- 图；
- 形式语言；和
- 自动机 学生应该学习足够多的数论知识来研究和实现基本的加密协议。

建议阅读

数据结构和算法

学生应该必须见过常见（或者罕见但异常有效的）数据结构和算法。但是，比起知道特定算法和数据结构（这些经常是很容易查阅到的），计算机科学家应该理解知道如何去设计算法（比如贪心、动态规划策略等）并且知道如何将理想中的算法真正实现。

特别推荐

对于想获得长期雇佣关系的计算机科学家来说至少要知道这些：

- 哈希表；
- 链表；
- 数；
- 二分查找树；和
- 有向、无向图 计算机科学家应该可以实现或者扩展操作这些数据结构的算法，包括增删改查特定元素。考虑到完备性，计算机科学家应该知道每个算法的指令式和函数式实现。

理论

理解理论是在研究生院进行研究的先决条件。当能提供了一个问题的hard boundaries（或者是提供转化为最初是hard boundaries的方法） 时理论是无价的。

计算复杂度可以说是所有计算机“科学”的真正的预测理论之一。

计算机科学家必须 知道易处理性和可计算性的程度，如果忽略了这些限制，最好的情况是有些挫折，最差的情况是导致失败。

特别推荐

在本科阶段，理论至少应涵盖计算模型和计算复杂度。

计算模型应该包括有限状态自动机、正则语言（和正则表达式）、下推自动机、上下文无关语言、形式文法、图灵机、lambda 演算和不可判定性。

在本科阶段，学生至少要学习足够复杂的知识来理解 P、NP、NP-Hard 和 NP-Complete 的区别。

为了防止留下错误的印象，学生应该通过将一些 NP 的问题规约到 SAT（Boolean satisfiability problem，布尔可满足性问题）并使用 SAT 求解程序求解。

架构

对软件架构有见识的理解是无可替代的。

计算机科学家应该从晶体管起理解一个计算机。

架构的理解包含一些标准的抽象：晶体管、逻辑门、加法器、多路复用器、触发器、算术逻辑单元、控制单元、缓存和随机存取存储器。

对高性能计算 GPU 模型的理解在可预知的未来是很重要的。

特别推荐

要想在现代系统上达到高性能对缓存、总线和物理内存管理的理解是很重要的。

要想理解机器架构，学生应该设计和仿真一个小的 CPU。

操作系统

任何足够大的程序最终都将成为一个操作系统。

正因如此，计算机科学家应该知道内核是如何处理系统调用、分页、调度、上下文切换、文件系统和内部资源管理的。

对操作系统的理解仅次于对编译器和实现高性能的架构的理解。

理解操作系统（我想当然也包括运行时的系统）在对嵌入式系统进行编程是非常重要。

特别推荐

学生必须在一个真正的操作系统上动手实践，在 Linux 和虚拟化技术的帮助下，这比之前容易些。想要对内核有很好的理解，学生应该：

- 在启动过程中输出 “hello world”；
- 设计他们自己的调度器；

- 修改分页策略；
- 创建他们自己的文件系统

网络

考虑到网络的普遍性，计算机科学家应该对网络栈和网络中的路由协议有坚实的理解。

对计算机科学家来说，在不可靠传输协议（比如 IP）的基础上构建可靠的传输协议（比如 TCP）的机制不应是不可思议的而应是核心知识。

他们应该理解在协议设计中的权衡—比如，什么时候选择 TCP，什么时候选择 UDP。（程序员需要知道在大型网络中有阻塞，他们也应更大规模地使用 UDP。）

特别推荐

考虑到当代程序员进行网络编程的频繁性，理解现存协议标准是有用的：

- 802.3 和802.11;
- IPv4 和 IPv6;
- DNS, SMTP 和HTTP. 计算机科学家应该理解包冲突时的指数回退和在拥塞控制中的加法增大和乘法减少机制。每个计算机科学家应该实现：
- 一个 HTTP 的客户端和守护进程；
- 一个 DNS 解析器和服务器；以及
- 一个命令行的 SMTP 的邮件程序 要想通过网络介绍课程，每个学生都应该使用wireshark来嗅探他们导师的谷歌搜索。

也许要求每个学生基于 IP 来从头实现一个可靠的传输协议是有些强人所难了，但可以说这是我学生时代的一个对我个人改变很大的经历。

安全

一个悲伤的事实是大多数安全漏洞都来源于粗心的编码，更悲哀的事实是很多学校在训练程序员编写安全代码上做的很差。

计算机科学家必须知道程序被攻破的方式。

他们需要形成防御型编码的意识——考虑他们自己的代码可能被攻击的方式。

安全最好在整个课程体系中分布开来进行训练：每个学科都应该提醒学生关于这个学科的原生漏洞。

特别推荐

每个计算机科学家至少应该了解：

- 社会工程；
- 缓冲区溢出；

- 整数溢出；
- 代码注入漏洞；
- 竞态条件；
- 权限混淆 一些读者指出计算机科学家也应知道基本的 IT 安全措施，比如选择合理的好密码和使用 iptables 配置防火墙。

密码学

密码学使得我们的大部分数字生活成为现实， 计算机科学家应该理解并能够实现下面的概念，并且知道实现这些的常见陷阱：

- 对称密码系统；
- 公钥密码系统；
- 安全哈希函数；
- 询问-响应认证；
- 数字签名算法；
- 门限密码系统 在实现这些密码系统时有个常见的错误——为手头工作获得 足够 随机的数，而这是每个计算机科学家应该知道的。
- 最后，如此多的数据泄露表明，计算机科学家应该知道如何在存储密码时进行加盐和哈希处理。

特别推荐

每个计算机科学家应该有使用手工统计工具来破解使用前现代加密系统的密文的乐趣。

RSA 是容易实现的 ，每个人都应试试。

每个学生都应创建他们自己的数字签名并在 apache 上建立 https 连接（做这个是出乎意料的费劲）。

学生还应该写一个使用 SSL 进行连接的 web 客户端。

作为实践，计算机科学家应该知道如何使用 GPG、ssh 的公钥认证、加密一个文件夹或者硬盘。

建议阅读

软件测试

软件测试必须贯穿整个课程体系。一个软件工程的课程可以涵盖基本的测试风格，但是只有练习才能掌握这项艺术。

应该根据学生上交的测试用例来给他们打分。

我使用学生上交来的测试用例来对其他学生进行测试。

学生看起来并不很在意防御性的测试用例，但是当向同学下手时却很是不客气。

用户体验设计

程序员大多是给其他程序员写程序，或者更糟糕，给他们自己写。

用户接口设计（更宽泛的讲，用户体验设计）可能是计算机科学最不受重视的方面。

即使是在专家之间也有这种误解，即用户体验是一种无法被教授的“软”技能。

在现实中，现代用户体验设计根植于人因工程学和工业设计中的人工经验。

如果没有别的办法，计算机科学家至少应知道接口执行任何任务的难易程度应该与任务的频率与重要性的乘积成比例。

为实用性考虑，每个程序员应该习惯于使用 HTML、CSS 和 Javascript 等设计可用的 web 接口。

可视化

好的可视化是可以将数据表现为人类可以感知的信息，而做到这点并不容易。

现代世界是数据的海洋，而开发人眼感知的局部最大值是理解这些信息的关键。

并行化

如今并行化比以往更落后、更丑陋。

不幸的是要掌握并行化需要对架构：多核、缓存、总线、GPU 等等有很深的理解。

并且需要练习，大量练习。

特别推荐

并行化的“终极”答案还不得而知，但是一些领域特定的解决方案已经给出。

当下学生应该学习 CUDA 和 OpenCL。

线程是脆弱的并行化抽象，特别是引入缓存和缓存一致性之后。但是，线程很流行且微妙，所以值得学习，Pthread 是一个合理的轻量库。

对于对大规模并行化感兴趣的人来说，MPI是首要条件。

在理论上，map-reduce 是经久不衰的。

软件工程

软件工程的原理改变地和编程语言一样快。

一个好的动手实践的团队软件开发练习能够展现出软件工程固有误区并提供关于这些误区的工作知识。一些读者建议说学生应该分为三人一组并且在不同的项目中轮流当作组长。

学习如何与现存大代码库打交道是每个程序员的必备技能，并且最好是在学校而不是在工作中掌握此项技能。

特别推荐

所有的学生都应知道集中版本控制系统如 svn 和分布式版本控制系统如 git。

对于调试工具如 gdb 和 valgrind 的使用很长时间后会有裨益。

形式化方法

随着对安全可靠软件的需求提高，形式化方法也许将是开发这种软件的唯一方法。

当前软件的形式化模型和证明还很有挑战性，但是这项领域的进程是稳健的：一年比一年容易。

也许在当前的计算机系学生的有生之年，形式化软件开发能成为一种预期技能。

每个计算机科学家应至少熟练使用一种定理证明器（我认为具体是哪一种并不重要）。

学习使用定理证明方法能够立刻影响代码风格。

比如说，一个人本能的不愿写无法覆盖所有可能性的 `match` 和 `switch` 语句，。

再比如当写递归函数时，使用理论证明方法的人有很强的欲望去消除 ill-foundedness。

图形仿真

没有学科比图更能体现“聪明”。

这个领域是由“足够好”驱动甚至由之定义的。

因此，没有比图形仿真更好的方式来教授巧妙的编程和进行性能优化。

我所学到的半数编码技巧都来自于对图的学习。

特别推荐

简单的光线追踪器可以在百行代码内实现。

实现从 3D wireframe engine 获取 3D 投影是费些脑力的。

类似于 BSP 的数据结构以及类似于 z-buffer 渲染的算法是巧妙的设计的例子。

在图形仿真领域，还有很多其他实例。

机器人

机器人是教授编程入门的最具吸引力的方式之一。

并且随着机器人的价格持续走低，哪一款将引发个人机器人浪潮成为了门槛。

对于会编程的人来说，个人机器自动化的伟大时代即将来临。

人工智能

仅是考虑到对早期计算历史的特大影响，计算机科学家也应学习人工智能。

即使人工智能的最初梦想还远未实现，人工智能在一些领域已有成效，比如机器学习、数据挖掘和自然语言处理等。

机器学习

除去出色的技术技术优点，对“relevance engineer”工作岗位的需求增大表示出每个计算机科学家都应该了解一下基本的机器学习。

机器学习也更加强调了理解概率论和统计的重要性。

数据库

数据库十分常见和有用以至于人们常常忽略它。

理解支撑数据库引擎的数据结构与算法是有用的，因为程序员经常需要在一个大的软件系统中实现一个数据库系统。

在sub-Turing 的计算模型的极大成功背后关系代数和关系计算起了极大的作用。

比起 UML 模型，ER 模型更适于可视化编码设计和约束的软件设计。

还有什么？

由于我自己也是知识盲点的，所以上面这些建议也是有局限的。

如果还有哪些应当包含但没有列出的东西，请大家在评论中补充。

更多频道

小组 - 好的话题、有启发的回复、值得信赖的圈子
头条 - 分享和发现有价值的内容与观点
相亲 - 为IT单身男女服务的征婚传播平台
资源 - 优秀的工具资源导航
翻译 - 翻译传播优秀的外文文章
文章 - 国内外的精选文章
设计 - UI,网页，交互和用户体验
iOS - 专注iOS技术分享
安卓 - 专注Android技术分享
前端 - JavaScript, HTML5, CSS
Java - 专注Java技术分享
Python - 专注Python技术分享