01 绪论

数据结构和算法

作者：小甲鱼

让编程改变世界

Change the world by program

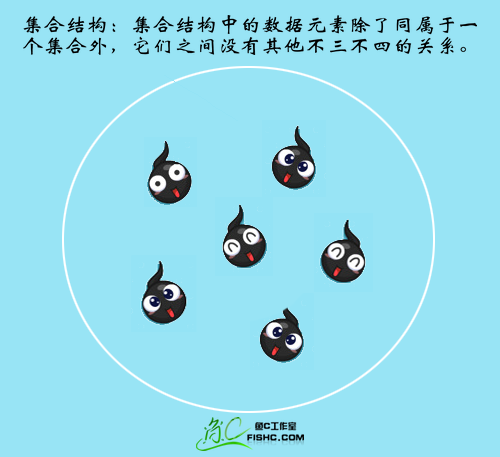
数据结构和算法绪论

* 数据结构和算法这门计算机必修课历来无论在哪个学校，都是无比乏味和催人入睡的。
* 但是，小甲鱼决定要投入大量的精力来将这门课程打造成有屎以来最为华丽的，最为欢乐地，最为图文并茂的课程！
* 因为，在中国，有一句古训：No picture you say a J8 a !
* 鉴于本节目是向上的，积极地，不被和谐的，所以小甲鱼就不会说出中文版本了。
* 数据结构和算法这是一门不太容易学好的课程，因为这门课程比较搞脑子，所以建议每天只听一集视频，并准备好鸡汤等营养品。
* 虽然这门课程不太好学，但如果你想让自己的编程能力有质的飞跃，不再停留于调用现成的东西而是追求更完美的实现，那么这是你的必修课！
* 如果你的目的是为了考计算机、软件方面的研究生，那么这门必考课现在就值得你开始准备。因为很多时候，考研玩的不是智商，其实就是一个人投入的时间而已。
* 什么是数据结构？
* 数据结构是一门研究非数值计算的程序设计问题中的操作对象，以及它们之间的关系和操作等相关问题的学科。
* 这样的官方陈词不是小甲鱼的风格哈，如果是小甲鱼，会告诉你数据结构事实上就是这样子：
  + 程序设计 = 数据结构 + 算法
  + 再简单的来说数据结构就是关系，没错，就是数据元素相互之间存在的一种或多种特定关系的集合。

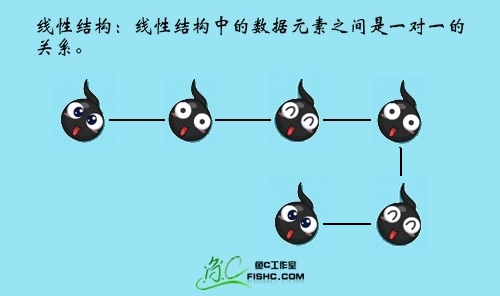
逻辑结构和物理结构

* 传统上，我们把数据结构分为逻辑结构和物理结构。
* 逻辑结构：是指数据对象中数据元素之间的相互关系，也是我们今后最需要关注和讨论的问题。
* 物理结构：是指数据的逻辑结构在计算机中的存储形式。
* 好，那让小甲鱼图文并茂地给大家介绍下四大逻辑结构吧：（此处可以有掌声）

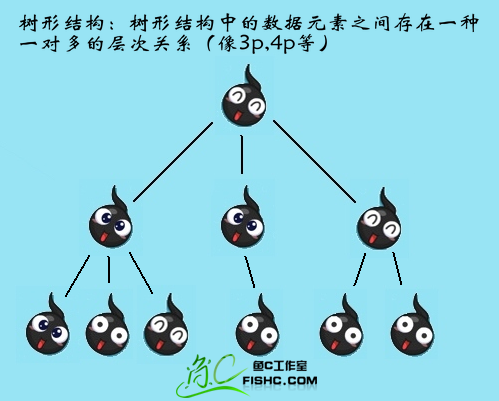
集合结构



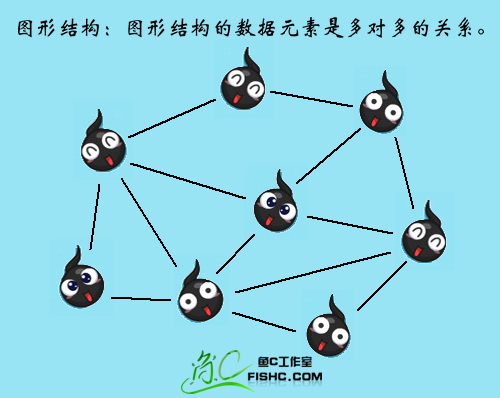
线性结构



树形结构



图形结构

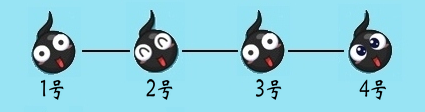


物理结构

* 说完逻辑结构，我们再来说说数据的物理结构。
* 根据物理结构的定义，我们实际上研究的的就是如何把数据元素存储到计算机的存储器中。
* 存储器主要是针对内存而言的，像硬盘、软盘、光盘等外部存储器的数据组织通常用文件结构来描述。
* 数据元素的存储结构形式有两种：顺序存储和链式存储。

顺序存储结构

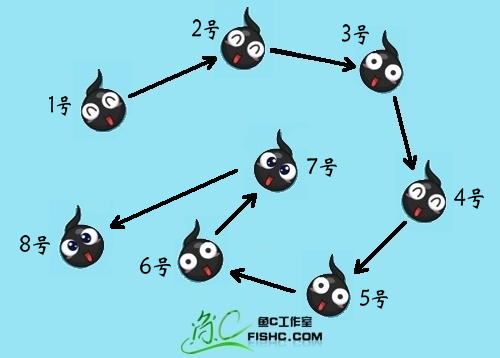
* 顺序存储结构：是把数据元素存放在地址连续的存储单元里，其数据间的逻辑关系和物理关系是一致的。
* 例如我们编程语言的数组结构就是这样滴。



链式存储结构

* 从顺序存储结构我们想到了日常生活中我们的排队，有木有？但现实生活中，我们发觉也并不完全如此。
* 例如有人排着排着她内急，她要被迫离开队伍去上洗手间，还有人不遵守基本道德规范他插队，这些情况会打破存储结构的基本原则。
* 面对这样时常要变化的结构，顺序存储是不科学的，那么就该让链式存储结构露面了。
* 现在如银行、医院等地方，都设置了排队系统。也就是每个人去了，先领一个号，等着叫号，叫到你的时候就可以去存一百块给小甲鱼或看病。
* 而在等待的时候，你爱在哪在哪，可以坐着、站着或者四处看看美眉，只要你及时回来就行。
* 这些情况下，你关注的是前一个号有没有被叫到，叫到了，下一个就该轮到你了。
* 链式存储结构就是这样的原理，相比起顺序存储结构就灵活多了。
* 链式存储结构：是把数据元素存放在任意的存储单元里，这组存储单元可以是连续的，也可以是不连续的。
* 很显然，这样说的话链式存储结构的数据元素存储关系并不能反映其逻辑关系，因此需要用一个指针存放数据元素的地址，这样子通过地址就可以找到相关联数据元素的位置。

No picture you say a j8 …



02 谈谈算法

谈谈算法

* 我们这门课程叫“数据结构和算法”，有鱼油可能会问这不是两门课程呢？为什么整在一起讲解呢？不是徒增我们的思想负担吗？
* 矮油，这看来小甲鱼是有必要跟大家解释一下数据结构和算法的关系啦。
* 打个比方，其实数据结构和算法的关系就比好基友是一辈子的关系。
* 他们患难见真情，他们生死不相弃，他们荣辱与共，他们一生情一辈子……



* 事实上，数据结构和算法也有类似的关系。只谈数据结构，我们可以在很短的时间内就把几种重要的数据结构介绍完。
* 不过听完后，你可能没啥感觉，不知道这些数据结构有啥用处。
* 但如果我们把相应的算法结合起来讲一讲，演示一下，你就会发现，甚至开始感慨:O，原来小甲鱼以及计算机界的前辈们，的确是一些很牛很牛的人，他们的工作使很多看似很难解决的问题变得如此美妙和神奇A。

算法初体验

* 小学学过珠算的鱼油应该很有印象，每天加法运算敲得手指都快断了就算那1+2+…+99+100。
* 这会儿，小甲鱼就给大家介绍一个有关也是有关从1加到100的小故事作为开端吧！
* 很久很久很久以前……
* 有部分淘气的鱼油可能不屑一顾，切，我们刚开始学习小甲鱼你的《零基础入门学习C语言》的时候，早就教过我们用C来写1加到100的代码咯。那时候你还说咱计算机的速度是何其快啊！

int i, sum = 0, n = 100;

for(i=1; i <= n; i++)

{

sum = sum + i;

}

printf(“%d”, sum);

* 对比下，用高斯先生的算法，我们可以这么写：

int i, sum = 0, n = 100;

sum = (1+n)\*n/2;

printf(“%d”, sum);

* 可能以计算机的神速，两个算法都可以秒杀解决掉！但是，如果我们把条件换成1加到1千万，或者1加到1千亿，差距就可想而知了，甚至人脑都可以比电脑计算得快了。
* 那么什么是算法呢？
  + 算法是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作。
  + 懵了吧？用小甲鱼的话来讲，算法就是你泡妞儿的技巧和方式。
* 从刚才的例子中我们看到，对于给定的问题，是可以有多种算法来解决的。
* 这就像追女孩子，总不可能每个人追女孩子的方式都一样吧？举个例子，小甲鱼常常在街上看到很多美眉，但旁边都是挽着条件十分一般的男朋友。这时候，小甲鱼就会由衷的佩服该男同胞一定是用了特别牛掰的算法追到美眉的！
* 就像没有药可以包治百病一样，一个问题可以由多个算法解决，一个算法也不可能具有通解所有问题的能力。

算法的特性

* 考虑到大部分学习小甲鱼《数据结构和算法》的鱼油都是学生，都要对付各种考试和考核。
* 注意，小甲鱼这里用了“对付”而不是“应付”，虽然是填鸭式教育，但是，如果我们能从中学到有用的知识并且可以对付考试，是最好的！
* 所以小甲鱼这个系列的视频教程也是针对性的要把经常考试的概念提一提说一说谈一谈侃一侃。
* 嗯，算法具有五个基本特征：输入、输出、有穷性、确定性和可行性。
* 输入
  + 算法具有零个或多个输入。
  + 尽管对于绝大多数算法来说，输入参数都是必要的。但是有些时候，像打印“I love fishc.com”，就不需要啥参数啦。

void print()

{

printf(“I love fishc.com\n”);

}

* 输出
  + 算法至少有一个或多个输出。
  + 算法是一定要输出的，不需要它输出，那你要这个算法来干啥？输出的形式可以是打印形式输出，也可以是返回一个值或多个值等。
* 有穷性
  + 指算法在执行有限的步骤之后，自动结束而不会出现无限循环，并且每一个步骤在可接受的时间内完成。一个永远都不会结束的算法，我们还要他来干啥？程序可以为无穷性。
* 确定性
  + 算法的每一个步骤都具有确定的含义，不会出现二义性。
  + 算法在一定条件下，只有一条执行路径，相同的输入只能有唯一的输出结果。
  + 算法的每个步骤都应该被精确定义而无歧义。
* 可行性
  + 算法的每一步都必须是可行的，也就是说，每一步都能够通过执行有限次数完成。

算法设计的要求

* 刚才我们谈到了，算法并不是唯一的。也就是说同一个问题，可以有多种解决问题的算法。
* 这可能让那些常年只做有标准答案题目的童鞋失望了，他们多么希望存在标准答案，因为只有一个正确的，把它背下来万事大吉！
* 但是咱的算法是变幻无穷的，还记得我们刚才讲过的高斯童鞋吗？世界上要多几个这样的鱼油，那又会有多几种牛掰的算法哈。
* 尽管算法不唯一，但我们要学习掌握一些好的算法，对我们解决问题很有帮助！
* 正确性
  + 算法的正确性是指算法至少应该具有输入、输出和加工处理无歧义性、能正确反映问题的需求、能够得到问题的正确答案。
  + 大体分为以下四个层次：
    - 算法程序没有语法错误。
    - 算法程序对于合法输入能够产生满足要求的输出。
    - 算法程序对于非法输入能够产生满足规格的说明。
    - 算法程序对于故意刁难的测试输入都有满足要求的输出结果。
* 可读性
  + 算法设计另一目的是为了便于阅读、理解和交流。
  + 我们写代码的目的，一方面是为了让计算机执行，但还有一个重要的目的是为了便于他人阅读和自己日后阅读修改。
* 健壮性
  + 当输入数据不合法时，算法也能做出相关处理，而不是产生异常、崩溃或莫名其妙的结果。
* 时间效率高和存储量低
  + 生活中，每个男人都希望找一个贤惠的老婆，她们温柔又体贴，美丽又大方，还会做着一手的好菜。
  + 好算法就犹如好老婆，应该具备时间效率高和存储量低（时间复杂度和空间复杂度）的特点。所以在设计算法的时候我们应该尽量思考这两方面的问题！

03时间复杂度和空间复杂度1

算法效率的度量方法

* 上一讲中我们提到设计算法要尽量的提高效率，这里效率高一般指的是算法的执行时间。
* 那么我们如何来度量一个算法的执行时间呢？
* 所谓“是骡子是马拉出来遛遛”，比较容易想到的方法就是我们把算法跑若干次，然后拿个“计时器”在旁边计时。
* 这种事后统计方法看上去的确不错，并且也并非真的要你拿个计算器在那里计算，因为计算机都有计时功能。
* 事后统计方法：这种方法主要是通过设计好的测试程序和数据，利用计算机计时器对不同算法编制的程序的运行时间进行比较，从而确定算法效率的高低。
* 但这种方法显然是有很大缺陷的：
  + 必须依据算法事先编制好测试程序，通常需要花费大量时间和精力，完了发觉测试的是糟糕的算法，那不是功亏一篑？赔了娘子又折兵？
  + 不同测试环境差别不是一般的大！
* 我们把刚刚的估算方法称为事后诸葛亮。我们的计算机前辈们也不一定知道诸葛亮是谁，为了对算法的评判更为科学和便捷，他们研究出事前分析估算的方法。
* 事前分析估算方法：在计算机程序编写前，依据统计方法对算法进行估算。
* 经过总结，我们发现一个高级语言编写的程序在计算机上运行时所消耗的时间取决于下列因素：
* 算法效率的度量方法
  + 1. 算法采用的策略，方案
  + 2. 编译产生的代码质量
  + 3. 问题的输入规模
  + 4. 机器执行指令的速度
* 由此可见，抛开这些与计算机硬件、软件有关的因素，一个程序的运行时间依赖于算法的好坏和问题的输入规模。（所谓的问题输入规模是指输入量的多少）
* 我们搬回高斯先生的那个算法来跟大家谈谈：

第一种算法：

int i, sum = 0, n = 100; // 执行1次

for( i=1; i <= n; i++ ) // 执行了n+1次

{

sum = sum + i; // 执行n次

}

第二种算法：

int sum = 0, n = 100; // 执行1次

sum = (1+n)\*n/2; // 执行1次

第一种算法执行了1+(n+1)+n=2n+2次。

* 第二种算法，是1+1=2次
* 如果我们把循环看做一个整体，忽略头尾判断的开销，那么这两个算法其实就是n和1的差距。
* 有些喜欢跟真理死磕的朋友可能对小甲鱼这观点意见不是一般的大！
* 因为循环判断在算法1里边执行了n+1次，看起来是个不小的数量，凭什么说忽略就能忽略？

淡定，请接着继续看延伸的例子：

int i, j, x=0, sum=0, n=100;

for( i=1; i <= n; i++ )

{

for( j=1; j <= n; j++ )

{

x++;

sum = sum + x;

}

}

这个例子中，循环条件i从1到100，每次都要让j循环100次，如果非常较真的研究总共精确执行次数，那是非常累的。

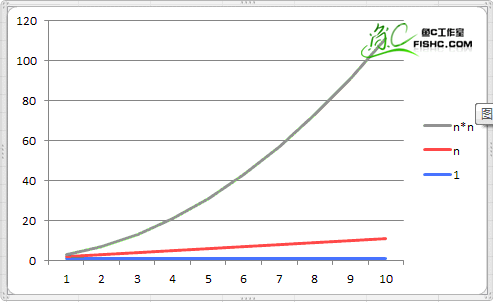
另一方面，我们研究算法的复杂度，侧重的是研究算法随着输入规模扩大增长量的一个抽象，而不是精确地定位需要执行多少次，因为如果这样的话，我们就又得考虑回编译器优化等问题，然后，然后就永远也没有然后了！

所以，对于刚才例子的算法，我们可以果断判定需要执行100^2次。

我们不关心编写程序所用的语言是什么，也不关心这些程序将跑在什么样的计算机上，我们只关心它所实现的算法。

这样，不计那些循环索引的递增和循环终止条件、变量声明、打印结果等操作。最终，在分析程序的运行时间时，最重要的是把程序看成是独立于程序设计语言的算法或一系列步骤。

我们在分析一个算法的运行时间时，重要的是把基本操作的数量和输入模式关联起来。



函数的渐近增长

小甲鱼给大家做一个测试：判断以下两个算法A和B哪个更好？

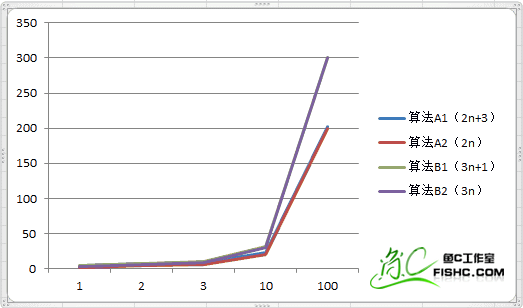
假设两个算法的输入规模都是n，算法A要做2n+3次操作，你可以这么理解：先执行n次的循环，执行完成后再有一个n次的循环，最后有3次运算。

算法B要做3n+1次操作，理解同上，你觉得它们哪一个更快些呢？

在给大家解答问题之前，小甲鱼先给大家做个图表参考：

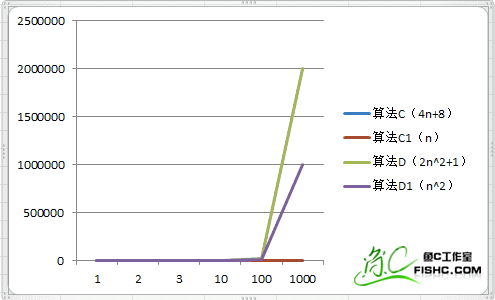
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **规模** | **算法A1（2n+3）** | **算法A2（2n）** | **算法B1（3n+1）** | **算法B2（3n）** |
| n=1 | 5 | 2 | 4 | 3 |
| n=2 | 7 | 4 | 7 | 6 |
| n=3 | 9 | 6 | 10 | 9 |
| n=10 | 23 | 20 | 31 | 30 |
| n=100 | 203 | 200 | 301 | 300 |

* 函数的渐近增长
* 当n=1时，算法A1效率不如算法B1，当n=2时，两者效率相同；当n>2时，算法A1就开始优于算法B1了，随着n的继续增加，算法A1比算法B1逐步拉大差距。所以总体上算法A1比算法B1优秀。



* 函数的渐近增长：给定两个函数f(n)和g(n)，如果存在一个整数N，使得对于所有的n>N，f(n)总是比g(n)大，那么，我们说f(n)的增长渐近快于g(n)。
* 从刚才的对比中我们还发现，随着n的增大，后面的+3和+1其实是不影响最终的算法变化曲线的。
* 例如算法A2，B2，在图中他们压根儿被覆盖了。所以，我们可以忽略这些加法常数。
* 后边我们给大家举多几个例子，会更明显。
* 函数的渐近增长
* 第二个测试，算法C是4n+8，算法D是2n^2+1。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **次数** | **算法C1（4n+8）** | **算法C2（n）** | **算法D1（2n^2+1）** | **算法D2（n^2）** |
| n=1 | 12 | 1 | 3 | 1 |
| n=2 | 16 | 2 | 9 | 4 |
| n=3 | 20 | 3 | 19 | 9 |
| n=10 | 48 | 10 | 201 | 100 |
| n=100 | 408 | 100 | 20001 | 10000 |
| n=1000 | 4008 | 1000 | 2000001 | 1000000 |

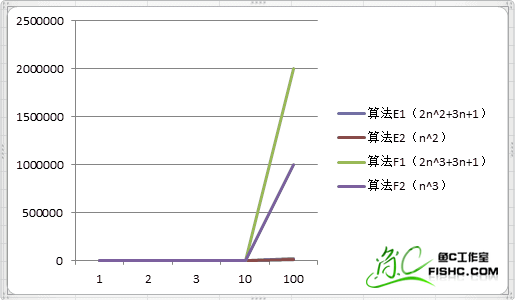
* 再来看一下线性图。
* 

函数的渐近增长

* 我们观察发现，哪怕去掉与n相乘的常数，两者的结果还是没有改变，算法C2的次数随着n的增长，还是远小于算法D2。
* 也就是说，与最高次项相乘的常数并不重要，也可以忽略。
* 函数的渐近增长
* 我们再来看第三个测试，算法E是2n^2+3n+1，算法F是2n^3+3n+1。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **次数** | **算法E1（2n^2+3n+1）** | **算法E2（n^2）** | **算法F1（2n^3+3n+1）** | **算法F2（n^3）** |
| n=1 | 6 | 1 | 6 | 1 |
| n=2 | 15 | 4 | 23 | 8 |
| n=3 | 28 | 9 | 64 | 27 |
| n=10 | 231 | 100 | 2031 | 1000 |
| n=100 | 20301 | 10000 | 2000301 | 1000000 |

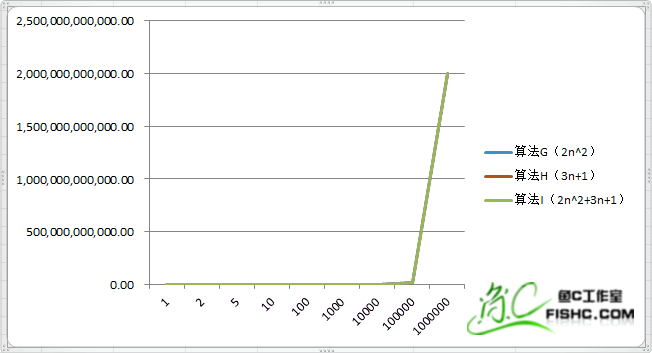
再来看一下线性图。



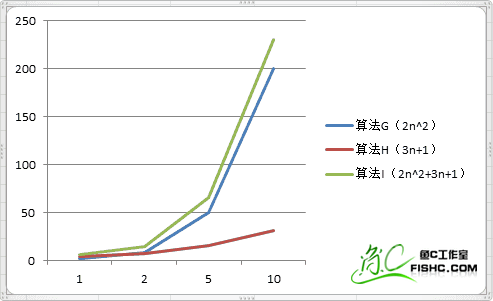
函数的渐近增长

* 这次我们又发现什么呢？
* 小甲鱼没有小鸡鸡？
* 不是的，我们通过观察又发现，最高次项的指数大的，函数随着n的增长，结果也会变得增长特别快。
* 恩，我们进行最后一个小测试，把这些概念都总结起来吧！
* 算法G是2n^2，算法H是3n+1，算法I是 2n^+3n+1。

|  |  |  |  |
| --- | --- | --- | --- |
| **次数** | **算法G（2n^2）** | **算法H（3n+1）** | **算法I（2n^2+3n+1）** |
| n=1 | 2 | 4 | 6 |
| n=2 | 8 | 7 | 15 |
| n=5 | 50 | 16 | 66 |
| n=10 | 200 | 31 | 231 |
| n=100 | 2000 | 301 | 20301 |
| n=1000 | 2000000 | 3001 | 200301 |
| n=10000 | 200000000 | 30001 | 200030001 |
| n=100000 | 20000000000 | 300001 | 20000300001 |
| n=1000000 | 2000000000000 | 3000001 | 2000003000001 |

函数的渐近增长

* 看出啥？一条直线？当他们数据很小的时候是这样的：



这组数据我们看得很清楚，当n的值变得非常大的时候，3n+1已经没法和2n^2的结果相比较，最终几乎可以忽略不计。而算法G在跟算法I基本已经重合了。

于是我们可以得到这样一个结论，判断一个算法的效率时，函数中的常数和其他次要项常常可以忽略，而更应该关注主项（最高项）的阶数。

注意，判断一个算法好不好，我们只通过少量的数据是不能做出准确判断的，很容易以偏概全。

04 时间复杂度和空间复杂度2

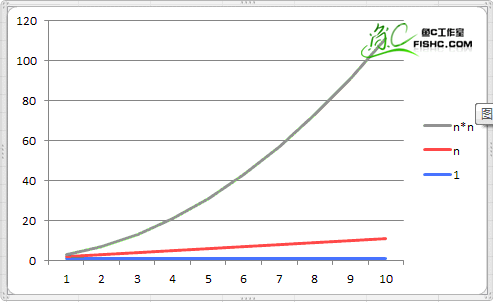
算法时间复杂度

* 我们说好的时间复杂度和空间复杂度呢？
* 历来大学老师在讲解这两个概念，都是直接登堂入室，导致八成学生对概念理解不深刻，或者说只是硬背起来而已。
* 为了让大家能够更好地接受这两个比较重要的概念，我们有了上一讲的准备环节。
* 这一讲我们直接切入正题，介绍计算复杂度的攻略，然后通过一系列例子和大家一起分析总结规律。

算法时间复杂度的定义：在进行算法分析时，语句总的执行次数T(n)是关于问题规模n的函数，进而分析T(n)随n的变化情况并确定T(n)的数量级。算法的时间复杂度，也就是算法的时间量度，记作：T(n)= O(f(n))。它表示随问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，称作算法的渐近时间复杂度，简称为时间复杂度。其中f(n)是问题规模n的某个函数。

好长好长，没想到定义这个概念的老家伙比小甲鱼还罗嗦。（关键需要知道执行次数==时间，cpu执行次数消耗一定量的时间）

* 这样用大写O()来体现算法时间复杂度的记法，我们称之为大O记法。
* 一般情况下，随着输入规模n的增大，T(n)增长最慢的算法为最优算法。
* 显然，由此算法时间复杂度的定义可知，我们的三个求和算法的时间复杂度分别为O(1)，O(n)，O(n^2)。
* 三个求和算法？哪有？忘了？
* 好吧，看看以下这张图能不能勾起点回忆？



推导大O阶方法

* 那么如何分析一个算法的时间复杂度呢？即如何推导大O阶呢？我们给大家整理了以下攻略：
  + 用常数1取代运行时间中的所有加法常数。
  + 在修改后的运行次数函数中，只保留最高阶项。
  + 如果最高阶项存在且不是1，则去除与这个项相乘的常数。
  + 得到的最后结果就是大O阶。
* 世界上的东西就是这么简单，老头儿们把它讲复杂，那么它就复杂了，举几个例子：

常数阶

int sum = 0, n = 100;

printf(“I love fishc.com\n”);

printf(“I love Fishc.com\n”);

printf(“I love fishC.com\n”);

printf(“I love fIshc.com\n”);

printf(“I love FishC.com\n”);

printf(“I love fishc.com\n”);

sum = (1+n)\*n/2;

* 大家觉得这段代码的大O是多少？
* O(8)？这是初学者常常犯的错误，总认为有多少条语句就有多少。
* 分析下，按照我们的概念“T(n)是关于问题规模n的函数”来说，这里大家表示对鱼C的爱固然是好的，要支持的，要鼓励的，要大力表彰的。但是，跟问题规模有关系吗？没有，跟问题规模的表亲戚都没关系！，所以我们记作O(1)就可以。
* 另外，如果按照攻略来，那就更简单了，攻略第一条就说明了所有加法常数给他个O(1)即可。

线性阶

* 一般含有非嵌套循环涉及线性阶，线性阶就是随着问题规模n的扩大，对应计算次数呈直线增长。

int i , n = 100, sum = 0;

for( i=0; i < n; i++ )

{

sum = sum + i;

}

* 上面这段代码，它的循环的时间复杂度为O(n)，因为循环体中的代码需要执行n次。

平方阶

* 刚才是单个循环结构，那么嵌套呢？

int i, j, n = 100;

for( i=0; i < n; i++ )

{

for( j=0; j < n; j++ )

{

printf(“I love FishC.com\n”);

}

}

* n等于100，也就是说外层循环每执行一次，内层循环就执行100次，那总共程序想要从这两个循环出来，需要执行100\*100次，也就是n的平方。所以这段代码的时间复杂度为O(n^2)。
* 那如果有三个这样的嵌套循环呢？
* 没错，那就是n^3啦。所以我们很容易总结得出，循环的时间复杂度等于循环体的复杂度乘以该循环运行的次数。
* 刚刚我们每个循环的次数都是一样的，如果：

int i, j, n = 100;

for( i=0; i < n; i++ )

{

for( j=i; j < n; j++ )

{

printf(“I love FishC.com\n”);

}

}

* 惨了，老办法好像在这里套不上了，咋整？！
* 分析下，由于当i=0时，内循环执行了n次，当i=1时，内循环则执行n-1次……当i=n-1时，内循环执行1次，所以总的执行次数应该是：
  + n+(n-1)+(n-2)+…+1 = n(n+1)/2
* 大家还记得这个公式吧？恩恩，没错啦，就是高斯先生发明的算法丫。
* 那咱理解后可以继续，n(n+1)/2 = n^2/2+n/2
* 用我们推导大O的攻略，第一条忽略，因为没有常数相加。第二条只保留最高项，所以n/2这项去掉。第三条，去除与最高项相乘的常数，最终得O(n^2)。

对数阶

* 对数，属于高中数学内容啦，对于有些鱼油可能对这玩意不大理解，或者忘记了，也没事，咱分析的是程序为主，而不是数学为主，不怕。
* 我们看下这个程序：

int i = 1, n = 100;

while( i < n )

{

i = i \* 2;

}

* 由于每次i\*2之后，就举例n更近一步，假设有x个2相乘后大于或等于n，则会退出循环。
* 于是由2^x = n得到x = log(2)n，所以这个循环的时间复杂度为O(logn)。
* 其实理解大O推导不算难，难的是对数列的一些相关运算，这更多的是考察你的数学知识和能力。
* 所以这里小甲鱼要分两类来说下，对于想考研的朋友，需要强化一下你的数学尤其是数列方面的知识。对于想增长自己编程能力的朋友，大概知道规律即可，不要在高等数学的概念上死磕！

05 时间复杂度和空间复杂度3

函数调用的时间复杂度分析

* 如果我们把问题再实际化一点，大家是否能自己正确的分析出来呢？
* 我们来看下边这个例子：

int i, j;

for(i=0; i < n; i++) {

function(i);

}

void function(int count) {

printf(“%d”, count);

}

* 函数体是打印这个参数，这很好理解。function函数的时间复杂度是O(1)，所以整体的时间复杂度就是循环的次数O(n)。
* 假如function是下面这样，又该如何呢：

void function(int count) {

int j;

for(j=count; j < n; j++) {

printf(“%d”, j);

}

}

* 事实上，这和之前我们讲解平方阶的时候举的第二个例子一样：function内部的循环次数随count的增加(接近n)而减少，所以根据游戏攻略算法的时间复杂度为O(n^2)。
* 接着使出杀手锏，给鱼油们一个挑战的机会！
* 尝试自己分析以下程序的时间复杂度：

n++; -----------1

function(n); ------------n^2

for(i=0; i < n; i++) { ------n^2

function(i);

}

for(i=0; i < n; i++) {-------n^2

for(j=i; j < n; j++) {

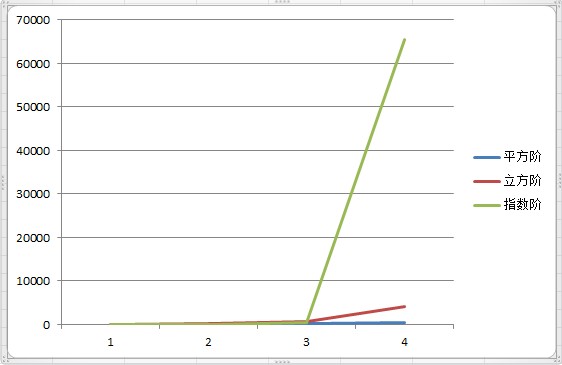
printf(“%d”, j);

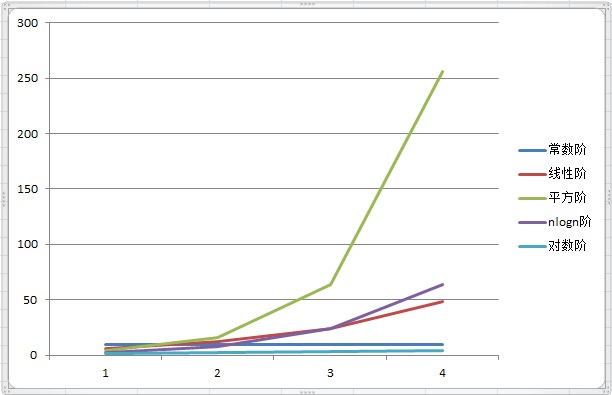
}

}

常见的时间复杂度

|  |  |  |
| --- | --- | --- |
| **例子** | **时间复杂度** | **装逼术语** |
| 5201314 | O(1) | 常数阶 |
| 3n+4 | O(n) | 线性阶 |
| 3n^2+4n+5 | O(n^2) | 平方阶 |
| 3log(2)n+4 | O(logn) | 对数阶 |
| 2n+3nlog(2)n+14 | O(nlogn) | nlogn阶 |
| n^3+2n^2+4n+6 | O(n^3) | 立方阶 |
| 2^n | O(2^n) | 指数阶 |





* 常用的时间复杂度所耗费的时间从小到大依次是：O(1) < O(logn) < (n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)
* O(1),O(logn),O(n),O(n^2)我们前边已经给大家举例谈过了，至于O(nlogn)我们将会在今后的课程中介绍。
* 而像O(n^3)之后的这些，由于n值的增大都会使得结果大得难以想象，我们没必要去讨论它们。

最坏情况与平均情况

* 从心理学角度讲，每个人对将来要发生的事情都会有一个预期。譬如看半杯水，有人会说：哇哦，还有半杯哦！但有人就会失望的说：天，只有半杯了。
* 一般人常出于一种对未来失败的担忧，而在预期的时候趋向做最坏打算。这样，即使最糟糕的结果出现，当事人也有了心理准备，比较容易接受结果，假如结局并未出现最坏的状况，这也会使人更加快乐，瞧，事情发展的还不错嘛！嗯，这是典型的自慰手法。
* 算法的分析也是类似，我们查找一个有n个随机数字数组中的某个数字，最好的情况是第一个数字就是，那么算法的时间复杂度为O(1)，但也有可能这个数字就在最后一个位置，那么时间复杂度为O(n)。
* 平均运行时间是期望的运行时间。
* 最坏运行时间是一种保证。在应用中，这是一种最重要的需求，通常除非特别指定，我们提到的运行时间都是最坏情况的运行时间。

算法的空间复杂度

* 我们在写代码时，完全可以用空间来换取时间。
* 举个例子说，要判断某年是不是闰年，你可能会花一点心思来写一个算法，每给一个年份，就可以通过这个算法计算得到是否闰年的结果。
* 另外一种方法是，事先建立一个有2050个元素的数组，然后把所有的年份按下标的数字对应，如果是闰年，则此数组元素的值是1，如果不是元素的值则为0。这样，所谓的判断某一年是否为闰年就变成了查找这个数组某一个元素的值的问题。
* 第一种方法相比起第二种来说很明显非常节省空间，但每一次查询都需要经过一系列的计算才能知道是否为闰年。第二种方法虽然需要在内存里存储2050个元素的数组，但是每次查询只需要一次索引判断即可。
* 这就是通过空间上的开销来换取计算时间开销的小技巧。到底哪一种方法好？其实还是要看你用在什么地方。
* 算法的空间复杂度通过计算算法所需的存储空间实现，算法的空间复杂度的计算公式记作：S(n)=O(f(n))，其中，n为问题的规模，f(n)为语句关于n所占存储空间的函数。
* 通常，我们都是用“时间复杂度”来指运行时间的需求，是用“空间复杂度”指空间需求。
* 当直接要让我们求“复杂度”时，通常指的是时间复杂度。
* 显然对时间复杂度的追求更是属于算法的潮流！