1.1.算法

算法，自古已有，并不是什么高深莫测的东西。

在古代，算法已经被大量研究和使用，比如绘图师对地图进行着色、商人计算成本和收益之间的关系，再比如计算平方根、筛选素数等等......这些都是很古老的问题了，有的甚至出现在古希腊时代。

包括我国著名的「曹冲称象」，也是一种算法。这些算法起源于瓦砾，在各个时代，经过一代一代学者们的改良，最终被沿用至今。

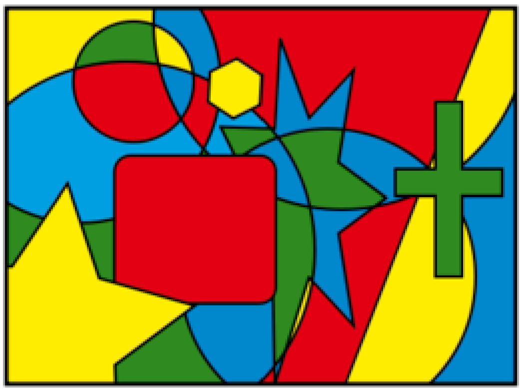

图1.0.1 四色理论，让相邻区域的颜色不冲突，最少需要4种颜色

我们今天的生活已经离不开算法——搜索引擎使用page rank算法对网页进行排序，电商网站用算法对用户进行产品推荐，导航系统用算法计算用时最短的路径，显卡用算法计算渲染结果，我们在手机上写字用到的手写识别，拼音输入用到的联想功能......尤其在近年来伴随着计算和存储能力的发展，算法在大数据和人工智能领域大放异彩——智能投资、医疗助手、无人工厂、自动驾驶、智能客服......可能这些还不够成熟，但在未来的10年中，这些技术会最终走向成熟，并成为我们生活的一部分。谷歌的首席执行官说，人工智能将比电和火为人类做出更大的贡献，这句话在未来也许不是空谈。而人工智能，并不是近年来才被发明的技术，比如「神经网络」在20世纪40年代就有理论讲解的图书问世了。

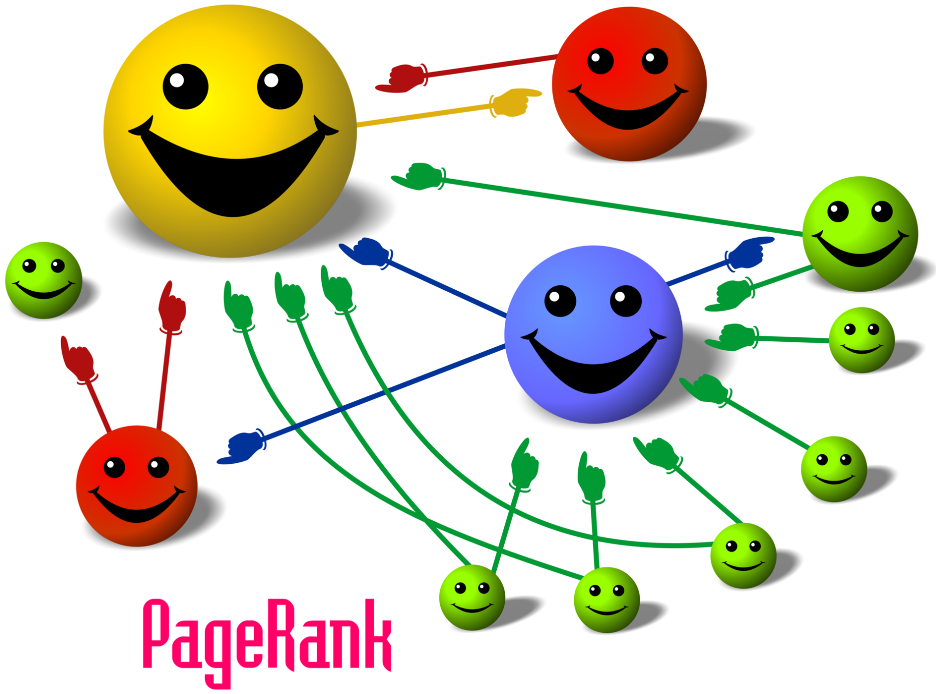

图1.0.2 维基百科上看到的一个卡通描述，每个笑脸代表一张网页，笑脸越大代表网页权重越高，被指向的网站越多权重越大

1.0.1.学习算法的意义

对大多数人来说，学习算法，并不是为了创造算法。很多人学习算法是为了在特定领域「应用」算法。就特定的算法而言，比如「计算机视觉」，往往在开源领域已经有了比较好的实现。如果学习原理，那么就可以更好的运用这些算法，让他们能够为自己服务。

对于更多的人来说，并不是为了直接应用某种算法而学习算法，他们是为了更好的「思考」，将学习算法所得的知识通过某种方式迁移到自己的日常工作中，从而提升工作效率。比如「衡量」算法好坏用到的「渐进记号」，这个可以用来衡量和分析日常工作中程序的执行时间，从而发现性能问题；比如，运用学习算法过程中学到的「循环不变式」这一概念，可以把复杂问题简单化，思考更加透彻；比如「哨兵」，可以有效减少程序中对边界条件的判断；比如，「贪心」算法，可以帮助程序员在解决优化问题时，先去找一种可以能的最优解；再比如，学习「散列」的思想，应用在系统的缓存设计上，可以加速缓存的访问。

当然还有一部分追求卓越的程序员，学习算法是为了「创造」。这样的程序员和算法的研究工作者不同，他们不是理论的研究者，他们将理论和实践结合。比如V8团队的创始人Lars Bak（丹麦程序员），使用隐藏类优化对象成员的访问；再比如React团队，最早使用Virtual DOM来加速网页的动态渲染。这些算法，固然有迹可循，但通常是某些基础算法的一个混合体（或者组合拳），或者某种算法的一个变化。发明这样的算法，然后对行业进步直

接作出贡献，对于大多数程序员而言，是至高的荣誉。 所以有很多优秀的程序员，将锻炼算法看做必修课，孜孜不倦的学习和努力。

1.0.1.算法的定义

算法，是明确定义了步骤的的计算过程。

所谓「明确」，举个例子，魔方很多人都接触过。对于大多数人，在没有学习专门的「还原算法」之前，还原一个3*3的魔方是非常困难的事情。甚至有一些运气成分，今天还原了，第二次不一定可以再还原。很多人认为还原魔方是一件凭借天赋的事情，其实魔方是有算法的。最快的算法目前可以确保在20步之内还原魔方。还原魔方的算法，必须让还原魔方的挑战者，能明确知道自己下一步应该做什么——何种情况下，应该转动哪一面？顺时针还是逆时针？任何时候，算法的执行者都知道下一步应该如何去做。

算法领域还存在着大量基于模糊数学或者基于随机的算法，这些算法的描述也需要足够明确。比如从数组中随机取一个数字，算法就要规定随机数如何生成等等。

其次，算法，是对经验的总结和概括。我们平时写程序，比如完成一些商业计算指标，或者对数据进行一定的处理，比如我们使用`Array.sort`对数据进行排序。但这些程序本身，只是解决了我们自己遇到的特定问题，而没有将他们上升到某一个高度去解决「某一类」问题，所以它们不是算法。如果我们能够把他们抽象出来，作为某类问题的解，那么也就成为了我们说的「算法」。

当然，算法往往是巧妙的。算法追求更快、更节省内存空间。比如一个3*3的魔方，总共有43,252,003,274,489,856,000种可能的状态。如果使用穷举法，那么需要很多年。但魔方自1970年被发明出来后，就有人提出了22步内还原魔方的解。在这之后，后人一直在优化还原它的算法。2010年，Google的一个组，还找到了20步还原魔方的方法。

1.0.1.数量级

对于10进制的数字而言，「数量级」是一个以10为底的近似描述。比如1500的数量级是3，因为$1500 = 1.5 \times 10^3$。

比如微信团队要对他们的所有用户根据某个数据指标排序，而微信月活10亿左右。那么微信团队的用户月活数量级是10。

我们也通常这样说，微信用户月活是「十亿级」，而不说是10，但其实是一个意思。因为，我们就是要一种模糊的描述而已。再比如2017年淘宝双11订

单8.12亿，我们要对这些订单进行分析，那么分析的数据规模是「亿」级。

1.0.1.输入和输出

算法的输入是数据，输出也是数据。

淘宝团队对双11订单进行分析，输入是双11的订单——这是一个「亿」级规模的输入。比如我们按日进行总额的统计，那么产出的结果是一个数组，这是算法的「输出」。

1.1.算法执行的环境

我们今天随便使用的一部便宜的智能手机的运算效率，都是当年第一部计算机ENIAC的万倍不止。（ENIAC每秒可以进行5000次计算）。

我们使用的大数据集群，可以将千万台服务器联合在一起形成计算资源，那是「万」级的CPU在同时进行计算，自然又是智能手机的「万」倍。所以从计算机发展至今，实际的计算能力已经前进了「亿级」级，甚至更多。

到了科学家们在研究量子比特，开始利用平行世界的力量进行计算的时代，计算能力又天翻地覆。

如果是一台50量子比特的量子计算机，据说每秒的计算次数可以达到1125亿亿次，那么又是今天极限计算速度的「亿级」倍数。如果，将来某天，量子计算机装进的手表里呢？ 当然这对我而言是无法想象的事情。只有现在的科幻小说作者们，才能想象出，有这样一天。

在如今学习算法，尤其经典算法，我们讨论的区间，要限定在我们现在计算机的计算能力之内讨论。那么就需要了解：计算机是如何执行程序的？

1.0.1.抽象

比如我们需要执行一条JS程序：

```
const sum = 5000 * 0.2
```

在编译阶段，5000和0.2都会存储到内存中去，不仅仅如此，这段代码本身也会转换成另种形态，存储到内存中去。

内存，和数组非常相似。每一个位置都有一个序号，我们称作「内存地址」。每个地址都是一个字节（8位）。

javascript中存储一个数字需要4个字节。比如说，我们将数字5000存放在内存地址1000。那么1000-1003这4个内存地址，都被5000占用了，因为数字需要4个字节。同理，0.2也需要4个字节。

另一方面，程序也会被存储到内存中。所以内存会划分成两个区域：数据空间用于存储数据，指令空间用于存储程序。

这里存储的程序，并不是原来的样子，而是编译后的语句，类似：

```
1. LOAD A, 1000   # 从地址1000读取数据到CPU的寄存器A
2. LOAD B, 1004   # 地址1004读取数据到CPU的寄存器B
3. Multiply C, A, B # 将寄存器A和寄存器B的值相乘，然后
4. Store C, 1008 # 将寄存器C的值存入地址1008
```

1.0.1.指令空间和数据空间

5000和0.2这样的数据，会被存放到「数据空间」。比如地址是1000和1004。而上述的4条指令，会被存放到「指令空间」。无论是指令空间还是数据空间，都是内存的一部分。

1.0.1.计算过程

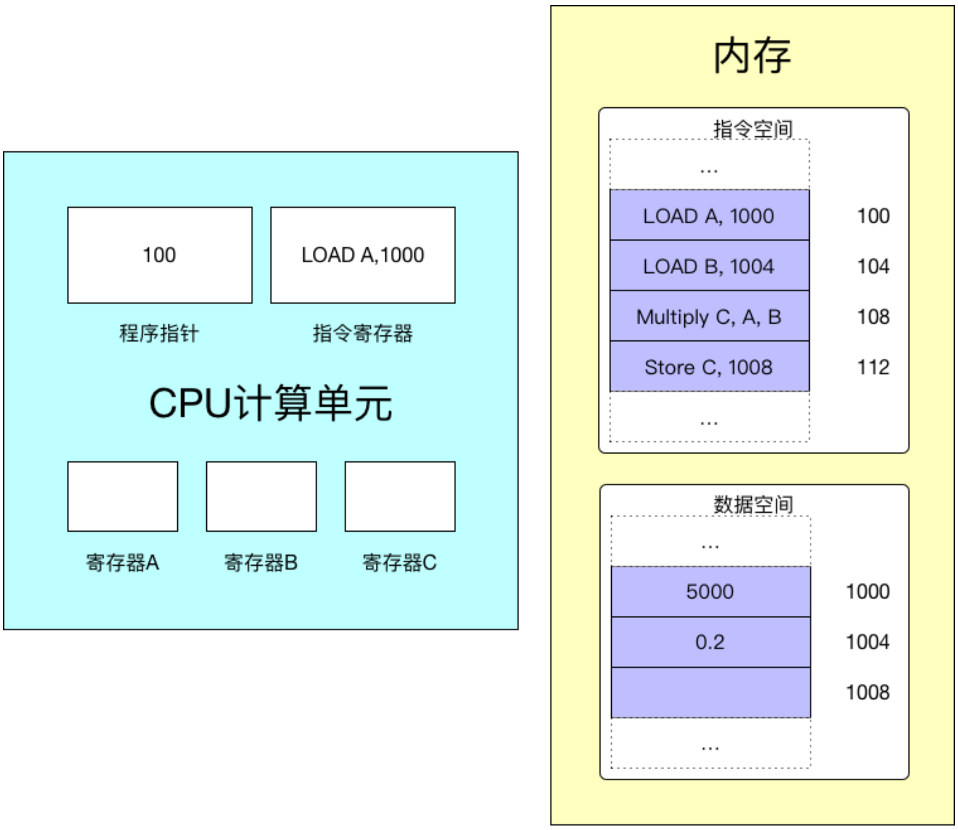

图1.0.3 CPU的计算模型，上图是刚好CPU从内存中读入了第一条指令，还没有执行这条指令

内存分成了「指令空间」和「数据空间」，作用是存储。

CPU的作用是计算，分成「寄存器」、「程序指针」和「计算单元」。首先，CPU不能直接从内存中读取数据进行计算。数据要进行计算，必须先从内存中读取到「寄存器」。

「寄存器」是数据在进行最后计算之前，存储的位置。 所以为了执行5000*0.2，第一条指令 `LOAD A, 1000` 是将数据5000从地址1000读取到寄存器A。第二条指令 `LOAD B, 1004` 是将数据0.2从地址1004读取到寄存器B。第三条指令，将寄存器A和寄存器B的值相乘，然后存入寄存器C。最后一条指令将寄存器C的值存入地址1008。

而指令也被存放在内存的「指令空间」中，CPU执行那一条指令，实际上是由CPU中的「程序指针」控制的。 程序指针，先指向地址100， 也就是第一条指令。每次执行完一条指令，程序指针，就指向下一条指令。同理，指令也需要从「内存」中读入寄存器， 有专门的指令寄存器。所以寄存器也分成指令寄存器和数据寄存器。

1.0.1.CPU指令的执行

执行一条指令，首先CPU要通过程序指针，找到从内存中对应的指令，读入指令寄存器；然后，CPU执行指令寄存器中的指令。 所以计算5000*0.2，总共消耗了4个这样的步骤。 每一个这样的步骤，需要消耗的时间是相对稳定的。

通常的，我们说CPU的主频是1GHZ，其实我们说的是CPU的时钟频率是每秒1G($10^9 = 1024 \times 1024 \times 1024$)个周期。通常，执行一条指令需要的周期是固定的，这个数值不会很大，比如2T、4T或8T（T指的就是一个周期)。

对于主频是1GHZ的CPU，每一个指令需要4个周期，那么每秒可以执行的指令数为:

$$(1024 \times 1024 \times 1024)/4 = 268,435,456$$

1.0.1.javascript的执行时间

5000*0.2只需要4个指令，是理论的最快速度。但javascript程序是解释执行过程，需要耗费额外的指令。另外，赋值语句也没有如此简单，比如说，const语句要消耗时间去检查变量名称之前有没有声明过。

所以这种理论分析是为了知道大致的时间开销，而并不是真正的执行时间。但是研究这种时间，可以让我们了解一个大致的必要开销，是算法分析的基础知识。

1.1.查找和二分查找的执行时间

从一列数字中查找是否存在目标数字，如果存在返回目标数字的序号，如果不存在，返回-1。如果该列数字没有规律，那么至少需要将这列数字遍历一遍，也就是一个一个的看一遍。当然有可能需要的值在第一个位置，但通常情况下，这个算法需要的时间会随着输入规模上升而上升。当这列数字是一个有序的数字，我们可以利用算法去优化，比如「二分查找」。

1.0.1.查找

数组如果没有顺序，从中找到值，需要一个一个去比对，这个方法我们称作「遍历」。通常是利用循环：

```
function search(arr, x){
  for(let i = 0; i < arr.length; i++){
    if(arr[i] === x){return i}
  }
  return -1
}
```

大家会发现这个算法的执行时间和规模成正比，因为for循环会执行数组的长度次。当然这种思考方法，是一种粗略估计。

1.0.1.逐行分析法

最简单的方法就是一行一行去标注和分析。需要分成「最坏情况」和「最好情」况。

对于最坏情况for循环中代码执行n次，最终没有找到或者最后一个元素是目标值。对于最好情况，第一个元素是目标值，for循环体执行1次。

最坏情况

最坏情况，是遍历完整个数组没有找到：

- `let i=0`执行1次，时间为c_1
- `i<arr.length`执行n+1次，时间为$c_2(n+1)$
- `i++`执行n次，时间为c_3n
- for循环体中`if`语句执行n次，时间为c_4n
- 最后一行`return null`执行1次，时间为c_5

具体一条javascript语句占用的时间，不好评估。有时候，操作系统也会开点小差。在上一节，我们知道，如 `let i=0` 这样的语句，至少需要操作若干次CPU寄存器，不需要很多时间。因此我们用常数c_1来替代。同理，但凡这类简单运算，我们都用常数替代。后续我们会看到，这样替代并不影响我们整体的评估和判断。

于是我们得到遍历查找在最坏情况下的时间：

$$T_{max} = c_1 + c_2(n+1) + c_3 n + c_4 n + c_5 = (c_2 + c_3 + c_4)n + c_1 + c_2 + c_5 = an + b$$

最终我们用$an + b$的形式来替代常数c_i。

从上述表达式可以得知，算法的执行时间T和规模n成正比，时间和规模的关系，画在纸上是一条直线。

最好情况

最好情况，第一次就找到了。for循环中 `let i=0` 执行1次， `i<arr.length` 执行1次， `if` 语句执行1次， `return` 执行1次。 所以：

$$T_{min} = c_1 + c_2 + c_4 + c_5 = C$$

最好情况，运行时间是一个常数。当然这个常数，每次执行都会有所变化。这是因为每次执行程序，计算的的情况是不同的，比如，CPU有可能在做其他事情。我们说它是常数，是因为时间和规模没有关系。无论规模多大，时间都是这么多。

1.0.1.二分查找

对于有序的情况，人会怎样查找呢？ 比如在一列按照拼音排好序的名单中查找「刘看山」，人会如何去查找呢？一种策略是先看名单中间的，比如发现是「秦始皇」，于是就不再查看「秦始皇」之后的人了，开始向前翻阅。

二分查找正是这样一种方法，与遍历不同，而是去猜测位置。先猜测最中间的数字，如果比目标大，那么说明在左半边；如果比目标小，那么说明在右边。

下图在2,10,15,18,33,55,78,129,210中查找129，虚线框代表数组的序号，下面的每一行代表依次查找，图中一共通过3次查找找到了对应的值：

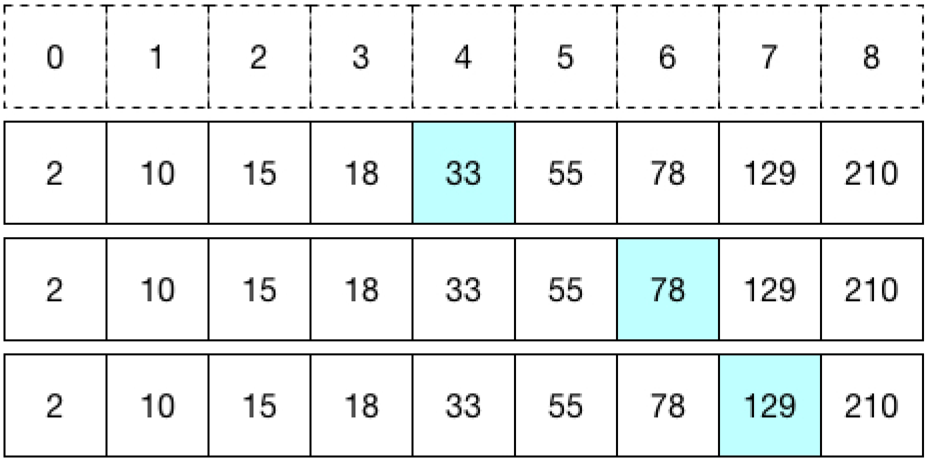

图1.0.4 先查找中间位置，发现33<129，于是去右边查找。然后发现右边的4个元素，中间位置有两个元素。这里可以设定一个规则，就是遇到相似情况，每次都取左边。于是找到78，发现78<129。于是去78的右边查找，取到129。

将上述方法写成代码，可以使用「递归」，或者使用循环：

```
1  function bsearch(A, x) {
2    let min = 0,
3        max = A.length,
4        guess
5
6    while(min <= max) {
7      guess = Math.floor( (min+max)/2 )
8      if(A[guess] === x) return guess
9      else if(A[guess] > x) max = guess - 1
10     else min = guess + 1
11   }
12   return -1
13 }
```

上述代码中的min-max代表猜测的范围。就好像说，在一份很长的名单中查找「刘看山」，第1次的范围是整份名单。而当我们从中间看到「秦始皇」之后，我们的查找范围就缩小了一半。

所以第6行的while循环在不断缩小查找范围，上述代码中使用了3个临时变量:

1. min指向猜测的下界
2. max指向猜测的上界
3. guess是当前猜测

每次while循环，都缩小了猜测的范围——要么min增加，要么max缩小，而且目标值，总是在min-max之间(除非根本不存在)。

这里，每次执行while循环开始前，min-max代表当前的搜索范围；每次while循环结束，min-max代表缩小后的范围。像这样，描述循环开始和结束条件的方式，我们称作「循环不变式」，这个概念我们会在后续深入讨论。

1.0.1.二分查找的时间开销

二分查找中的while循环究竟执行了多少次，不太好直观的获得。当然可以自己在纸上画带有箭头的图示，比如说数组总共有128个元素。二分查找的while循环，每次执行相当于将搜索范围减半。 所以搜索范围在按照这样的数量在递减:

128->64->32->16->8->4->2->1

所以如果是从128个元素中查找需要的元素，最差的情况，128(2^7)，一共需要执行7次while循环。

再比如，对于100个元素的数组，也就是规模在2^6和2^7之间，也会有如下的递减链条：

100->50->25->12->6->3->2->1

需要7次（数箭头的数量）。

归纳

归纳是人的本能，看到3天的日夜交替，所以推知每天都是这样。

看到上述的规律，我们也会推知一般的规律。那就是对于规模是n的数组，最坏情况需要几次搜索?

显然，如果n介于2^{m-1}和2^m之间，那么循环会执行m次。事实上，$m = \lceil \lg n \rceil$。

注:$\lceil\rceil$代表向上取整，log是幂运算的逆运算，比如$a^b = n$，那么$log_a n = b$

逐行分析

还是用之前的分析方法

- `let min=0,max=A.length,guess` 执行1次，时间我们用c_1代替。
- while循环在最差情况执行了$\lceil \lg n \rceil$次，其中需要执行若干次比较和赋值操作，执行时间我们用c_2代替

那么总的时间：

$T_{max} = c_1 + \lceil log_2 n \rceil c_2$

1.0.1.执行时间的比较

于是我们得到两种方法的执行时间，「遍历」和二分查找：

- 遍历：$T_1 = c_1 + c_2(n-1) + c_3 + c_4 n + c5 = (c_2 + c_4)n + c_1 + c_3 + c_5 - c_2$
- 二分查找: $T_2 = c_1 + \lceil \lg n \rceil c_2$

上面两个表达式，谁更快呢？ 我们可以用常数来替代试一试：

我们用常数做个替代，比如说「遍历」$T_1 = 10n + 4$，「二分查找」$T_2 = 10\lceil \lg n \rceil + 3$

当输入规模n=1,000时，$T_1 = 10004$，$T_2 = 103$

当输入规模n=1,000,000时，$T_1 = 10000004$，$T_2 = 203$

可见，随着时间增加，T_1和T_2的差距越来越大。

1.1.渐进记号

上一节，我们发现时间和规模之间可以用函数表示，比如：

$T_2 = 10\lceil \lg n \rceil + 3$

$T = 4n + 4$

$T = 0.1n^2 + 2n + 5$

$T = \lg n$+n^3$

$T = n!$

当然也有这种情况，算法的执行时间是一个常数，比如：

$T = 5$

这样就引出了形形色色的表达式。

1.0.1.复杂度

我们研究算法，是研究算法需要的时间、空间随着规模上升的趋势。这个时间、空间随着规模上升的趋势，我们称作算法的复杂度。下面我们通过观

察，引入一个符号来描述算法的复杂度。

1.0.1.观察

上一节我们发现对于$T_1=\lg n$的算法和$T_2=n$的算法，执行时间的差异非常大。比如n=1,000,000，$T_1=19$，$T_2=1,000,000$

而且这种趋势，似乎随着n的增加，就越加明显。

即便$T_1=1000\lg n$，$T_2=0.0001n$

对n=1,000,000，$T_1\approx 19000$，$T_2=100$。

对n=1,000,000,000，$T_1\approx 29000, T_2=100000$

我们发现T_1随规模上升非常缓慢，相比之下，T_2就迅速很多。同时，常数的大小对算法的增长影响似乎并不是很大。随着n上升，T_2迅速超过T_1。

同理，对于一个时间表达式是:$T=0.1n^2-2n+10$，随着n的上升0.1,$-2n$和10变得越来越不重要，最重要的(或者说影响最大的）就是n^2。

1.0.1.使用BIG-O对算法进行分类

对于$T=an^2+bn+c$的形式，最重要的是n^2。

对于$T=a\lg n+c$的形式，最重要的是$\lg n$。

对于$T=an+b$的形式，最重要的是n。

通过上述观察结果，我们发现算法的执行时间可以根据时间表达式中最重要（影响最大的），进行分类。我们这里使用一个大写的O来描述这样的分类：

- 最坏情况下，二分查找的时间**复杂度**是O(logn)
- 最坏情况下，任何情况下，遍历查找的时间复杂度是O(n)
- 最坏情况下，插入排序的时间复杂度是O(n^2)，归并排序的时间复杂度是O($nlogn$)……*插入排序和归并排序见下一章*
- ……

这种分类方法，比一个一个算法去描述节省时间和精力。因为我们通常只需要知道一个大体的分类，就知道程序的性能。就好像当看到两个嵌套的for循环，我们就可以猜测算法的是O(n^2)。这样的分类，方便我们比较算法和确定算法随着规模增加大体的性能。

比如说对于一个O(n^3)的算法，我们大概可以判断，输入在「万」级别，算法需要至少「万亿」级别的CPU周期。 那么凡是被归类成为$O(n^3)$的算法，我们就知道在实践中应该避免使用。

1.0.1.O(1)

O(1)代表了那些开销固定不变的算法，也就是算法用时不随规模上升而变化的算法。javascript中绝大多数库函数都是O(1),具体我们会在下一节介绍。

1.0.1.BIG-O的数学意义

BIG-O在数学上，被称作*渐进记号*。原本是用来描述函数的参数趋向极限（某一个值或者无穷）时函数值的表现。在算法领域，我们用来分类算法。

比如T=O(n)的实际数学含义是，对于函数g(n)=n，总是存在常数c>0，使得$T < cg(n)$。而O(n)就是上述满足条件的函数的集合。比如说对于T=100n，那么存在c=101使得在n增加的时候，T<101n；而c=102,c=103……都满足上述条件。

从反例上思考，比如对于T=100n，那么存不存在常数c>0使得$T < c\lg n$呢？显然这样的常数C是不存在的，因为无论c多么大，100n随着规模的n的增长都会超过$c\lg n$。

所以O(n)实际上，是一组函数。这组函数，描述了一个增长的上界。随着算法的规模上升，总是有一个函数，可以写成cn的形式，而且比T增长快。

1.1.常见Javascript程序的时间复杂度

1.0.1.$O(1)$

比如常见的**赋值**和对象构造语句，基本上都是O(1)：

```
const x = 1
let y = 2
const obj = {
  soft : 1,
  hard : 2
}

```

一些数学操作，比如，也是O(1)

```
Math.floor(5.12)
Math.sign(1)
```

判断一个元素是否在一个对象（集合)中，是O(1):

```

// 判断元素2是否在集合中
new Set([1,2,3]).has(2)

// 判断键a是否在对象x中
if(x['a']) {
  //...
}
```

1.0.1. *O*(*n*)

数组的初始化，是O(n)

```
new Array(10000)
```

上述语句虽然只有一行代码，但是实际要分配10000个元素，每个元素都要初始化，所以理论上它是O(n)的。

但是要注意的是，如果不去初始化数组，直接赋值，在绝大多数javascript引擎中，都是O(1)的，比如:

```
const arr = []
arr[999999] = 1
```

上述语句在多数引擎中都是O(1)的，因为实际上js引擎（比如V8)在运行时，发现数组被这样使用（就是「跳」着使用了，元素序数离得很远），会创造一个占用内存空间更小的散列表来替代这个数组。

数组求最大值、查找、过滤等等是O(n)的。

```

const arr = [1,2,3,4,5,6,7,8,9]

// 求最大(小)值
const max = Math.max(...arr)
const min = Math.min(...arr)

// 寻找 =4 的值
const four = arr.find(x => x===4)
```

```

// 寻找 >8 的值
const greatThan8 = arr.filter(x => x > 8)

// 求和
const sum = arr.sum((x, y) => x + y)

```

上述这些语句，都**隐含**着，需要将数组进行一次遍历，所以实际上是O(n)的。

另外很多字符串的操作，是O(n)的，比如：

```
1 const alphabets = 'abcdefghijklmnopqrstuvwxyz'
2 const m = str.indexOf('hello')
```

第1行： 字符串在内存中的表达方式，和数组很像，初始化一个字符串，实际上需要的时间和字符串的长度相关,所以也是O(n)的。

第2行：字符串的匹配算法，通常也是O(n)的，比如KMP算法（后续介绍）。

1.0.1. $O(nlogn)$

$O(nlogn)$的算法非常常见，比如数组的排序，在多数javascript引擎中都是O(nlogn)。当然也有少数O(n)的实现。

```
const arr = [9, 2, 3, 8, 5, 4, 1, 7]
arr.sort((x, y) => x - y)
```

上述语句通常是O(nlogn)。

之所以会有少量O(n)的实现，是因为，有一些js引擎，在数据量小的情况下，使用了一些比较占用内存空间的算法来换取执行效率。

1.0.1. $O(n^2)$

上一节提到$O(n^2)$的算法，其实已经比较慢了，所以在javascript提供的库中，$O(n^2)$算法通常是没有的。但有时候，程序写错了，会写出一些$O(n^2)$的算法。

比如：求两个数组的公共元素

$O(n^2)$的解法

```
const a = [1,3,5,4,6,2]
const b = [2,3,4,8,10]
a.filter(x => b.find(y => y === x))
```

结果

```
[3, 4, 2]
```

因为a.filter和b.find都隐含了$O(n)$的复杂度，所以两次叠加，就变成了

$O(n) \times O(n) = O(n^2)$

正确的做法是，使用Set或者先对a、b进行排序。（见习题）

1.1.习题

1.0.1.阶乘

一个数字N的阶乘定义为所有小于等于N的整数的乘积，表示为$N!$。比如

$10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$。

通常的，我们会用三个省略号，表示这些中间过程，比如

$10! = 10 \times 9 \times ... \times 1$

另外，我们定义$0! = 1$

写一个函数fatorial，计算任意整数的阶乘。并指出这个函数的算法复杂度。

1.0.1.求前k大的值

在本书中大家接触到的第一个算法，是求数组的最大值。请完成一个函数，求一个数组中第k大的值。

1.0.1.素数

一个数是否是素数，是一个古老的命题。 素数就是只能被1和自己整除的数字。比如2、3、5、7、13......

写一个函数is_prime判断一个数字n是否是素数。

1.0.1.集合交集

在1.5.4中提到，求集合相交如果使用两个循环，那么复杂度是$O(n^2)$，请优化这个算法。

对于两个数组A和B，求A、B中的相同元素。 比如

```
const A = [1,3,5,4,6,2]
const B = [2,3,4,8,10]
```

那么他们的相同元素是3,4,2

写一个函数join(a, b)求数组a，b的相同元素

提示： 之前提到两种算法，一种是利用集合或者对象；另一种，是先对A、B进行排序。