# 如何科学的计数？

2015-02-14 宾狗 理论 概率与统计

## 0x00 前言

一直以为，概率与统计是一门非常有传奇色彩的学科。它由赌博业催生而来，但你确不能靠它在赌场发家致富。它不能告诉你抛一次硬币是正是反，但它能告诉你抛1万次硬币正反面大概各5000次。而现实生活中的概率并非简单的抛硬币，我们所看到的并非真的是随机，而那些真正的随机又常常难以理解。

最近知乎上有一篇很不错的科普，《伪随机的上位和真随机的逆袭》，推荐一看~

## 0x01 从π的奇葩算法谈起

圆周率π我们每个人都很熟悉，说起圆周率的计算，我们最熟悉的版本无非是祖冲之的“割圆法”，当然随着科学的不断进步，数学界的牛人们不断刷新着π的精度，什么贝利 - 波尔温 - 普劳夫公式（BBP公式）、高斯-勒让德算法（GLA）、快速傅里叶变换、牛顿迭代法……

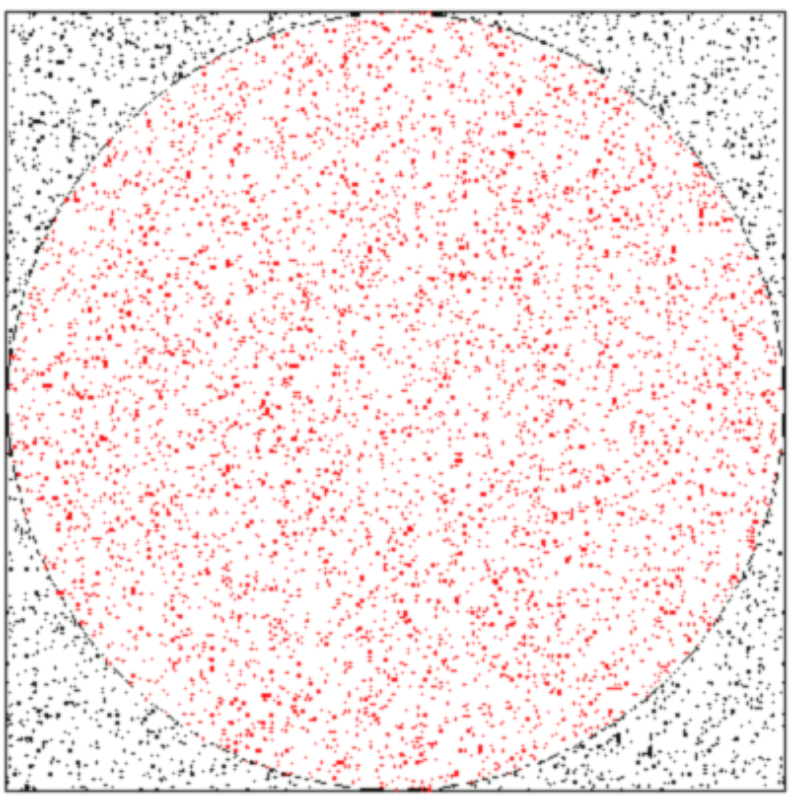
你如果对自己的数学充分自信的话可以看看《一起来算圆周率》，如果你和我一样已经完全抓瞎了，那么还是继续往下看吧ヽ(￣▽￣)ﾉ

有没有正常人可以理解同时也非常简单粗暴的方法来算π呢？当然有，这就是传说中的蒙特卡罗方法(Monte Carlo method)，俗称统计模拟，也可以称之为熔火之心(MC)ヽ(￣▽￣)ﾉ

说白了，这是一种用随机数或伪随机数来解决计算问题的方法，这话听上去非常矛盾，我们认知中的计算是非常精确的，怎么可以通过随机数来计算呢？别急，我们先来看看怎么用MC方法来计算π

## 内切圆

在平面上画一个正方形和一个内切圆，然后**随机**向正方形区域内撒$n$个点，统计落在圆内点的个数$m$，如下图所示



可以认为我们随机撒的点服从均匀分布，因此点的个数与平面区域的面积有如下关系

$$\frac{S_{circle}}{S_{square}} = \frac{m}{n}$$

$$\frac{\pi(\frac{d}{2})^2}{d^2} = \frac{m}{n}$$

最终可以得到

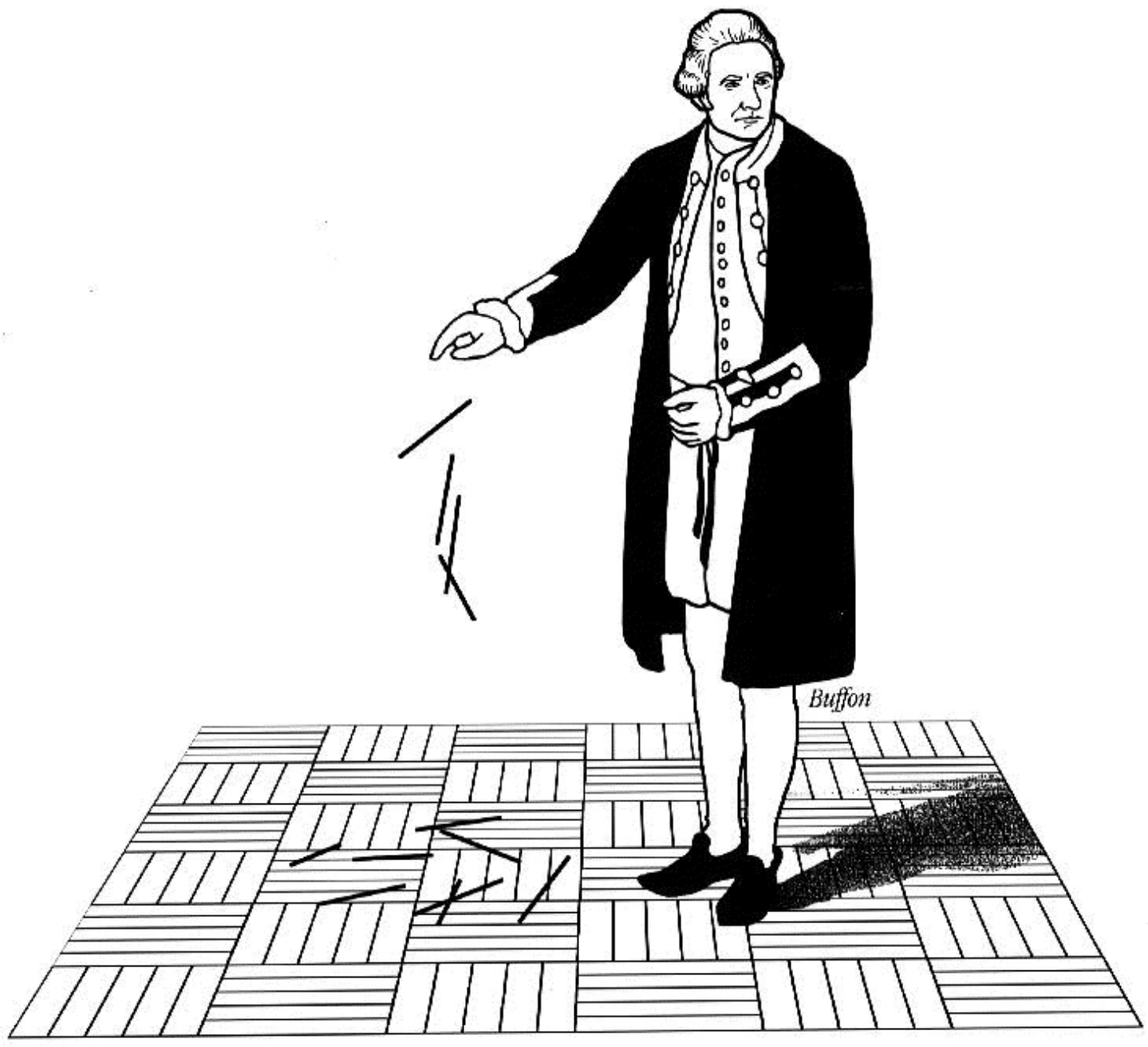
$$\pi = \frac{4m}{n}$$

你看，我们并没有用到什么高深的数学知识，只需要统计出区域内点的个数，利用上面的式子即可计算出π，而且我们撒出的点越多，计算出的π就越精确。用程序实现起来也非常方便

```
from random import random
n=10**6
print sum(1 if random()**2 + random()**2 < 1 else 0 for i in range(n))*4.0/n
```

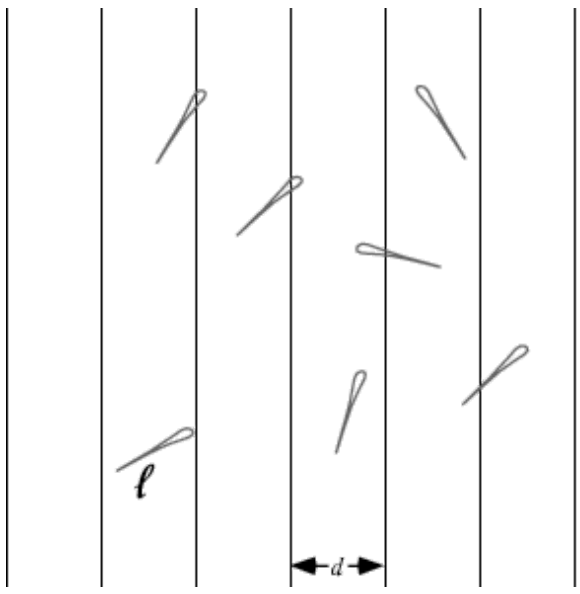
## 蒲丰投针

再来看第二种利用统计模拟计算π的方法，这就是大名鼎鼎的蒲丰投针问题(Buffon's needle)




在平面上画上无数条平行线，间距为$d$， 取一根长度为$l(l < d)$的针，随机向该平面投掷$n$次

然后统计与平行线相交的次数$m$，如下图所示



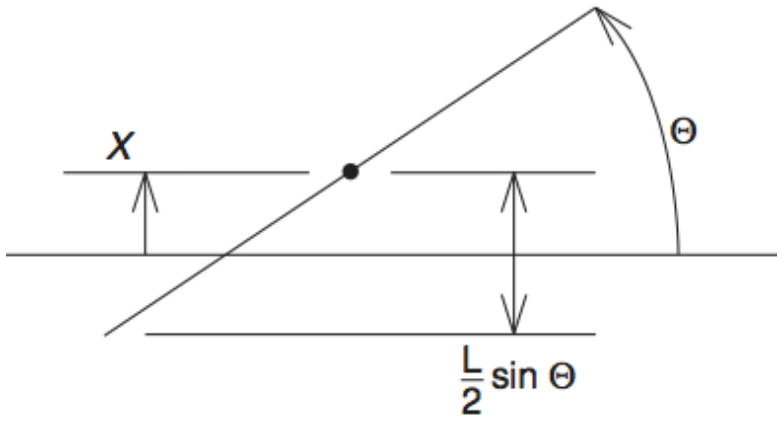

那么π可以由下式算出

$$\pi = \frac{2l}{d} \cdot \frac{n}{m}$$

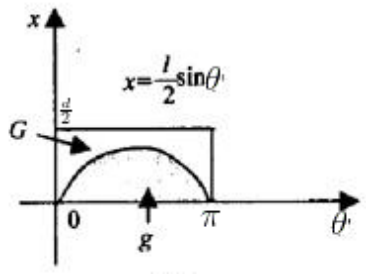
方法是很简单，但是为什么呢？下面我们来证明一下。

首先我们把问题简化一下，只考察**两条平行线之间的半区域**（其他区域都是等价的）。容易观察出针与平行线是否相交取决于两个因素，针的位置和旋转的角度，我们用$x(0 \leq x \leq \frac{d}{2})$表示针中心点到平行线的距离，用

$\theta(0 < \theta < \pi)$表示针与平行线的夹角，如下图所示




显然，当$x \leq \frac{l}{2} \cdot sin\theta$时，针与平行线相交。如下图所示




从图中可以看出，当$x$与$\theta$的取值落在区域$g$内时，上述条件即可满足，所以针与平行线相交的概率为

$$p = \frac{S_g}{S_G} = \frac{\int_0^{\pi} \frac{l}{2} \cdot sin\theta d\theta}{\frac{d}{2} \cdot \pi} = \frac{\frac{l}{2} \cdot (-cos\theta|_0^{\pi})}{\frac{d}{2} \cdot \pi} = \frac{2l}{d\pi}$$

而$p = \frac{m}{n}$，所以

$$\pi = \frac{2l}{d} \cdot \frac{n}{m}$$

python实现

```
from random import uniform
from math import pi, sin
#注意:为了便于计算,这里我们假定l=d=1
def direct_needle(N):
    N_hits=0
    for i in range(N):
        x_center=uniform(0.,0.5)
        phi=uniform(0,pi/2)
        x_tip=x_center - sin(phi)/2.
        if x_tip < 0: N_hits += 1
    return N_hits
n=10**6
print 2.0*n/direct_needle(n)
```

这里还有一个关于蒲丰投针证明的视频，讲的很生动形象（台湾腔的大叔讲话还是很有感觉的ㄟ(￣▽￣)ㄏ）

Error loading player:
No playable sources found

与MC方法相关的还有许多理论与算法，如马尔可夫链蒙特卡洛方法(Markov Chain Monte Carlo，简称MCMC)，吉布斯采样(Gibbs Sampling)……这些方法在物理、经济等诸多领域发挥着重要作用~当然本文的重点并不是介绍MC方法，只是想通过这个例子说明，在理论和事实之间有着某种神秘的联系（就像我们可以从大量统计数据中估算出π一样），而概率与统计就是我们认识和研究这种联系的利器。

**ps:后面的内容较长且涉及的数学知识较多，理解起来可能不太轻松，感兴趣的同学可以把本文存为书签，有时间的时候再看~**

# 0x02 基数计数(Cardinality Counting)

基数(cardinality，也译作势)，是指一个集合中不重复元素的个数。注意这里的集合和我们学过的严格定义的集合不同，允许存在重复元素，另外，本文所讨论的均为有限集。如给定这样的一个集合$\{1, 2, 3, 1, 2\}$，它有$5$个元素，但它的基数为$3$。

基数计数在很多领域都有应用，最为的典型同时也最为大家所熟知的就是网站统计了。如果把PV（Page View，即页面浏览量或点击量，用户每次刷新即被计算一次）看作一个集合，那么UV（Unique Visitor，访问该页面的一台电脑客户端为一个访客，也就是我们常说的独立访客）就是这个集合基数。显然UV是一个页面价值和影响力的一个重要评判指标，那么如何对其计数呢？

有人说这还不简单，定义一个包含独立访客标识，访问计数等信息的数据结构，最后用数组或者链表保存起来不就行了。每当有新的访问行为发生，则依据其访客标识进行查找，命中则说明该访客已访问过该链接，访问计数加一；若未命中，说明该访客是首次访问，则分配一个新空间保存该访客信息。

如果只是一个小型网站这么做是完全可以的，但是如果是一个每日PV在上亿甚至更高数量级的大型网站中，线性表的查询效率可想而知，而进行二分查找、B树等优化也会遇到各种各样的问题。这里我不想炒大数据这个概念，但在现实场景中，数据到达一定数量级之后这都是不得不考虑的问题。

## bitmap与Hash函数

bitmap是非常经典的海量数据处理数据结构，其本质是用bit数组的某一位表示某一数据，从而一个bit数组可以表示海量数据。用0表示某一元素不在集合中，用1表示某一元素在集合中，如`0100000011000000`可以用来表示集合$\{1,8,9\}$。

Hash函数，也称散列函数，原本是密码学领域的概念。说白了就是将一长串输入转换为固定长度的输出，即$H(M)=h$，其中$H$为Hash函数，$M$为密文，$h$为定长的hash值。同时要满足以下几个条件：

- 单向性
- 抗冲突性
- 映射分布均匀性和差分分布均匀性

更多细节内容就与本文无关了，这里不再赘述。但是在数据结构中我们用到的Hash函数并没有那么严格的要求，一般满足第二点就可以了，有的时候甚至简单的如求余函数也可作为Hash函数。那么为什么要使用Hash函数呢？很简单，单凭其时间复杂度为$O(1)$便足以傲视群雄了，在海量数据处理中，这个优点显得尤为突出。

怎样将两者结合起来进行基数计数呢？比如我们选取`md5`作为哈希函数，只取其结果的前8位，那么整个Hash函数的映射空间就有$16^8$（40多亿，几乎不会产生冲突）个值，取一个长度为$16^8$的bitmap（大约536MB），每一位对应Hash函数映射空间中的一个值，初始值全为0。每当有新访问产生，对该访客标识进行Hash，并映射到bitmap中的某一位上，若该位置为0，则置1；若为1，则不作处理。最后统计整个bitmap中1的个数即为基数。

整个过程没有特别耗时的查找操作，大大提高了效率。但这种方法占用了大量空间，我们这里只是统计某一个链接的数据，如果要统计全站的链接的话就不行了，即便换一个映射空间较小的Hash函数也不能从本质上解决这个问题。

当然这是属于比较精确的计数，相应的代价自然会比较高。在实际应用中我们可能并不需要知道太过精确的数据，10000000的访问量和10000100的访问量并没有太大差别，有没有一种方便快捷的估算方法呢？

又到了概率与统计出场的时候了~

## Linear Counting

这里我们先从简单的LC算法(Linear Counting)讲起，仔细分析上面的例子不难发现其空间占用较多是因为其过于追求Hash函数的抗冲突性，进而导致映射空间过大。LC算法正是大大降低了Hash函数的要求，并利用概率与统计的相关知识，最终给出基数的一个估计。

LC算法描述如下，设有一Hash函数H，其Hash映射空间有m个值（最小值0，最大值m-1），且服从均匀分布。取一个长度为m的bitmap，每一位与Hash函数映射空间中每个值一一对应，初始值全为0。设一个集合的基数为n，此集合所有元素通过Hash函数映射到bitmap中，如果某一个元素被Hash到第k个比特并且第k个比特为0，则将其置为1。当集合所有元素Hash完成后，设bitmap中还有u个bit为0。则：

$$\hat{n}=-m\log\frac{u}{m}$$

为n的一个估计，且为最大似然估计（MLE）。

乍一看，这个算法的描述和前面的差不多，其实最大的区别在于这里的Hash函数是允许冲突的，也就是说允许$H(M)=H(M')$的情况出现。

下面来看看如何进行证明。首先我们要明确一点，这里bitmap的最终值只与集合的基数有关，比如$A=\{k_1,k_2,\cdots,k_n\}$和$B=\{k_1,k_2,\cdots,k_n,x_1,x_2,\cdots,x_i\}$两个集合，其中$k_1,k_2,\cdots,k_n$为$n$个不同元素，$x_i\in\{k_1,k_2,\cdots,k_n\}$。显然，$A$和$B$的基数是相同的，经过Hash函数映射得到的bitmap也是一样的。

所以下面的证明我们就用集合$A$来进行辅助思考~

LC算法定义的Hash函数是满足映射均匀分布的，也就是说集合$A$中的**每一个元素映射到bitmap中的每一位都是等可能的**（这里有的同学可能会有点困惑，Hash函数不是确定的么，为什么又和等可能扯上关系了呢？可以这样理解，在未进行Hash运算时，你是不知道输入值将映射到哪个位置的，而由于Hash函数的均匀分布性，所以其映射到bitmap任何位置都是等可能的，但是相同的输入值一定会映射到相同的位置上）。显然，在一次映射中，

bitmap上的某一位被映射到的概率为$\frac{1}{m}$，不被映射到的概率为$1-\frac{1}{m}$，设事件$C_j$为“经过$n$次元素Hash后，bitmap上第$j$位为0的概率”，则有

$$C_j = (1-\frac{1}{m})^n$$

由于bitmap上每一位都是独立的，所以$u$的期望为

$$E(u) = \sum_{j=1}^{m} P(C_j) = m(1-\frac{1}{m})^n = m[(1+\frac{1}{-m})^{-m}]^{-\frac{n}{m}}$$

如果你高数还没忘光的话，一定还记得下面这个式子

$$\lim_{n\to+\infty}(1+\frac{1}{n})^n = e$$

自然数$e$就是这么定义的~

$$\begin{aligned}\lim_{m\to+\infty}(1+\frac{1}{-m})^{-m} &= \lim_{m\to+\infty}(1-\frac{1}{m})^{-m} \\ &= \lim_{m\to+\infty}(\frac{m-1}{m})^{-m} \\ &= \lim_{m\to+\infty}(\frac{m}{m-1})^{m} \\ &= \lim_{m\to+\infty}(1+\frac{1}{m-1})^{m} \\ &= \lim_{m\to+\infty}(1+\frac{1}{m-1})^{m-1} \\ &= \lim_{m\to+\infty}(1+\frac{1}{m})^{m} \\ &= e\end{aligned}$$

所以，当$m$和$n$都趋于无穷时有，

$$E(u) = me^{-\frac{n}{m}}$$

$$n = -m\ln\frac{E(u)}{m}$$

bitmap上每一位的值服从参数相同0-1分布，因此u服从二项分布。由概率论知识可知，当n很大时，可以用正态分布逼近二项分布，因此可以认为当n和m趋于无穷大时u服从正态分布。

u的概率密度函数

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

由于我们观察到的0的个数u是从正态分布中随机抽取的一个样本，因此它就是$\mu$的最大似然估计（正态分布的期望的最大似然估计是样本均值）。

又由如下定理：

$f(x)$是可逆函数，$\hat{x}$是$x$的最大似然估计，则$f(\hat{x})$是$f(x)$的一个最大似然估计。

$f(x) = -m\ln\frac{x}{m}$是可逆函数，$u$为$E(u)$的最大似然估计，所以$\hat{n} = -m\log\frac{u}{m}$为$n = -m\log\frac{E(u)}{m}$的一个最大似然估计。

这里我们也要注意到，m的取值不能太小，不然很有可能bitmap上所有位都被映射到了，这样u就为0了，整个算法就失去了意义。原论文作者给出了一张表

Table II. The Size of the Map Required to Assure that $m > \beta(e^t - t - 1)$ when $a = \sqrt{5}$

| | Map size $m$ epsilon | | | Map size $m$ epsilon | |
|---|---|---|---|---|---|
| $n$ | .01 | .10 | $n$ | .01 | .10 |
| 100 | 5034 | 80 | 80000 | 23029 | 10458 |
| 200 | 5067 | 106 | 90000 | 24897 | 11608 |
| 300 | 5100 | 129 | 100000 | 26729 | 12744 |
| 400 | 5133 | 151 | 200000 | 43710 | 23633 |
| 500 | 5166 | 172 | 300000 | 59264 | 33992 |
| 600 | 5199 | 192 | 400000 | 73999 | 44032 |
| 700 | 5231 | 212 | 500000 | 88175 | 53848 |
| 800 | 5264 | 231 | 600000 | 101932 | 63492 |
| 900 | 5296 | 249 | 700000 | 115359 | 72997 |
| 1000 | 5329 | 268 | 800000 | 128514 | 82387 |
| 2000 | 5647 | 441 | 900000 | 141441 | 91677 |
| 3000 | 5957 | 618 | 1000000 | 154171 | 100880 |
| 4000 | 6260 | 786 | 2000000 | 274328 | 189682 |
| 5000 | 6556 | 948 | 3000000 | 386798 | 274857 |
| 6000 | 6847 | 1106 | 4000000 | 494794 | 357829 |
| 7000 | 7132 | 1261 | 5000000 | 599692 | 439233 |
| 8000 | 7412 | 1412 | 6000000 | 702246 | 519429 |
| 9000 | 7688 | 1562 | 7000000 | 802931 | 598645 |
| 10000 | 7960 | 1709 | 8000000 | 902069 | 677040 |
| 20000 | 10506 | 3105 | 9000000 | 999894 | 754732 |
| 30000 | 12839 | 4417 | 10000000 | 1096582 | 831809 |
| 40000 | 15036 | 5680 | 50000000 | 4584297 | 3699768 |
| 50000 | 17134 | 6909 | 100000000 | 8571013 | 7061760 |
| 60000 | 19156 | 8112 | 120000000 | 10112529 | 8373376 |
| 70000 | 21117 | 9294 | | | |

可以看出m大约为n的十分之一左右。

## LogLog Counting

从前文可以看出LC算法虽然有一些提升，但毕竟只是线性的减少了空间占用，如果面对上亿级别的基数计数，这个方法依然要占用大量空间。但这小节将要介绍的LogLog Counting的空间复杂度只有$O(\log_2(\log_2(N)))$，是的你没看错，取了两次$\log$。

在开始介绍LLC算法前，要提前说明的是这里的Hash函数又是要求比较严格的，要保证碰撞几率极小，同时映射空间是近似均匀分布的。同时，LLC算法中不使用bitmap这个数据结构。

设a为待估集合中的一个元素，$h = H(a)$，这里把h表示为长度为L的比特串（如h为`098950fc`，写为`00001001100010010101000011111100`），将这L个比特串从左至右依次编号为1、2、……、L。因为Hash函数是均匀分布的，所以这L个比特服从如下分布且相互独立

$$P(x = k) = \begin{cases} 0.5 (k=0) \\ 0.5 (k=1) \end{cases}$$

其实也就是说h中每一个比特位出现0或者1都是等可能且互相独立的，设$\rho(h)$为比特串h中第一个1出现的位置（例如h为`098950fc`，则$\rho(h) = 5$）。现在对集合中的每个元素都作Hash运算，对每个生成的h都计算其$\rho(h)$，取$\rho_{max}$为所有$\rho(h)$中的最大值。此时，我们可以估计集合的基数为

$$\hat{n} = 2^{\rho_{max}}$$

为什么可以这样估计呢？

我们可以把上述寻找比特串中第一个1的过程看作一个投硬币试验：当硬币为反面时，记为0；当硬币为正面时，记为1，试验停止，记录投掷次数。设n次试验中，最大投掷数为k。

现在考虑如下两个事件：

A：进行n次试验，每次投掷次数都不大于k

B：进行n次试验，至少有一次投掷次数大于等于k

在一次试验中，投掷次数大于k的概率为$(\frac{1}{2})^k$，相当于只要连续投掷出k个反面，投掷次数肯定大于k，那么在一次试验中，投掷次数不大于k的概率为$1 - \frac{1}{2^k}$，所以

$$P(A) = (1 - \frac{1}{2^k})^n$$

B的对立事件为“进行n次试验，每次投掷次数都不大于k-1”，所以

$$P(B) = 1 - (1 - \frac{1}{2^{k-1}})^n$$

注意到，当$n \gg 2^k$时，$P(A) \to 0$，当$n \ll 2^k$时，$P(B) \to 0$，转换为自然语言描述就是，当n远远大于$2^k$时，每次试验投掷次数都不大于k的概率几乎为0，当n远远小于$2^k$时，至少有一次试验投掷次数大于等于k的概率也为0。而这些均与我们观察到的试验结果不符，即**存在至少一次试验的投掷次数等于k，且不存在比k更多的投掷次数**。所以唯一合理的推断是$n \approx 2^k$。

现在回到我们的初始问题，自然有

$$\hat{n} = 2^{\rho_{max}}$$

当然在实际使用中，直接这样估计仍会存在较大误差，所以LLC算法还采用了分桶平均的方法（类似物理实验的多次测量求平均值），这里不再深入。

# 0x03 总结

当然这里介绍的基数计数方法都是一些基本的算法和思想，除此之外还有Adaptive Counting、HyperLogLog Counting等方法，相关链接会在参考文献中列出。

概率与统计的神奇之处还不仅限于这些，在**统计学习**中有更多让人着迷的应用。最后，引用《统计与真理》这本书的里的话作为结语~

> All knowledge is, in final anlysis, history.
>
> All sciences are, in the abstract, mathematics.
>
> All judgements are, in their rationale, statistics.

---