

# 程序设计实习（实验班-2024春）

## 随机算法概述

授课教师：姜少峰

助教：冯施源 吴天意

Email: [shaofeng.jiang@pku.edu.cn](mailto:shaofeng.jiang@pku.edu.cn)

**随机算法：**  
**计算过程中利用随机性的算法**

# 随机算法的特点/性能衡量

- 确定性算法：对于确定的输入，**一定产生确定的输出**
- 随机算法：对于确定的输入，产生的输出是随机的，随机性来源于算法本身
- 可以有概率输出“错误”结果

Correct可以有很多含义，具体场景具体分析：  
比如算法达到某近似比，达到某时间复杂度

$$\Pr[\text{ALG is correct}] \geq 1 - \delta$$

$\delta$ 是失败概率

# 失败概率如何取？

- 如果算法只需要运行一次，那么 $\delta$ 取一个常数一般就足够

比如 $\delta = 10^{-3}$

- 但是如果算法需要多次运行，要保证每次成功，就需要让 $\delta$ 与运行次数有关
  - 例如算法需要支持某种查询，而每次查询都需要运行随机算法（失败概率 $\delta$ ）
  - 如果要运行 $q$ 次且每次失败概率是 $\delta$ ，那么存在一次失败的概率至多是 $q\delta$

union bound:  $\Pr \left[ \bigcup_i X_i \right] \leq \sum_i \Pr[X_i]$

# 随机算法的一般设计目标

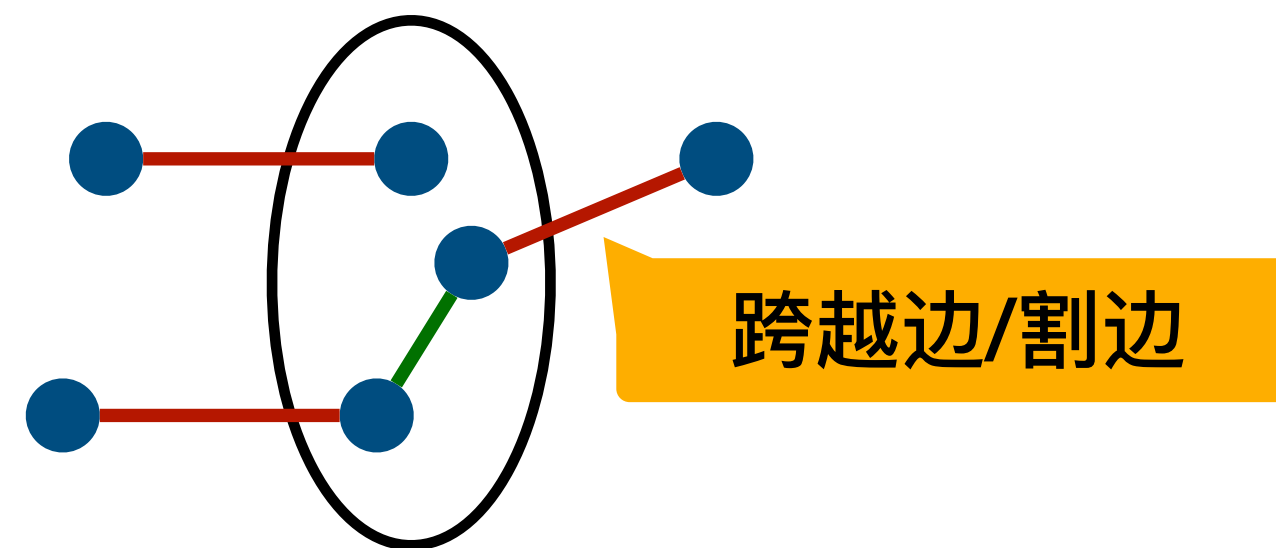
$$\Pr[\text{ALG is correct}] \geq 1 - \delta$$

- 一般目标：预先设定 $\delta$ 的值以及“correct”的含义，然后设计算法满足上述条件
- 下面以一个优化问题为例，讨论随机算法一般设计思路

# 最大割问题

## Max-Cut

- 输入一个无向无权图 $G(V, E)$ ，寻找最大割
- 割：对于一个点集 $S \subseteq V$ ， $S$ 定义的割是跨越 $S$ 的边集，记为 $\text{cut}(S)$



- 求 $S$ 使得 $|\text{cut}(S)|$ 最大

$S$ 定义的割的边数

# 第一步：数学期望

## 最重要的量：数学期望

基本性质

$$\mathbb{E} \left[ \sum_i X_i \right] = \sum_i \mathbb{E}[X_i]$$

• 如果X和Y独立，那么 $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$

$$\mathbb{E}[X] = \sum_x \Pr[X = x] \cdot x$$

- 数学期望是X的**典型**行为：我们可以“期望”看到X的值“大概”在 $\mathbb{E}[X]$ 附近
- 因此必须要先设计一个“无偏估计”，即 $\mathbb{E}[\text{ALG}]$ 是“correct”的
  - 这是一个必要条件，否则算法在典型情况都会误差很大

# 最大割的随机近似算法

## 数学期望分析

- 算法：对每个点  $u \in V$ ，以0.5概率独立放入  $S$
- 结论： $\mathbb{E}[|\text{cut}(S)|] = 0.5 \cdot |E| \geq 0.5 \cdot \text{Max-Cut}$ （期望是2-近似的）

cut的大小至多是边数，因此这里说明cut(S)期望是至少0.5倍的最大割

- 对于任何一条边  $(u, v) \in E$ ，有0.5概率  $(u, v)$  也在  $\text{cut}(S)$  里
- $\mathbb{E}[|\text{cut}(S)|] \geq 0.5 |E|$



## 第二步: $\Pr[\text{ALG is correct}] \geq \text{常数}$

(只是分析, 非算法步骤)

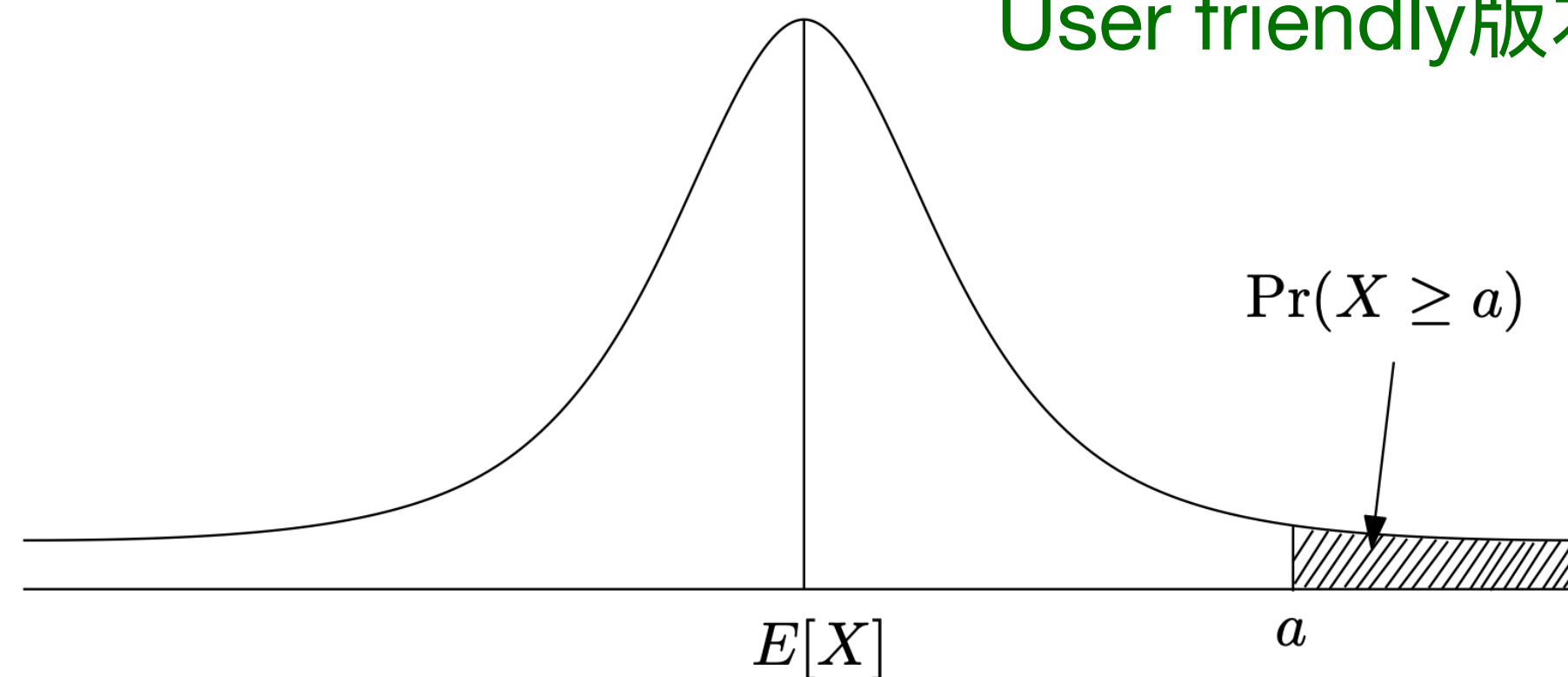
是“数学期望典型性”的自然结果  
这里是一个严格定量分析



Markov inequality: 若  $X \geq 0$ , 有  $\Pr[X \geq a] \leq \mathbb{E}[X]/a$

User friendly版本:  $\Pr[X \geq t\mathbb{E}[X]] \leq 1/t$

我们的好朋友: Andrey Markov  
1856 - 1922; “马尔可夫链”



不在割里的边数

• 考虑  $X = |E| - |\text{cut}(S)|$ , 则  $\mathbb{E}[X] = |E| - \mathbb{E}[|\text{cut}(S)|] \leq 0.5|E|$

•  $\Pr[|\text{cut}(S)| \leq (0.5 - \epsilon)|E|] = \Pr[X \geq (0.5 + \epsilon) \cdot |E|] \leq \frac{1}{1 + 2\epsilon}$

在割里的边很少

不在割里的边很多

## 第三步：通过多次重复来降低失败概率

一旦有了 $\Pr[\text{ALG is correct}] \geq \text{常数}$ ，就可以通过多次重复来降低失败概率

- 例如有 $\Pr[\text{ALG is correct}] \geq 0.1$ ，那么独立重复算法 $T$ 次后：

- $\Pr[\text{存在一次ALG is correct}] \geq 1 - 0.9^T$

- 如果要追求 $\delta$ 失败概率，那么 $T \approx \log(1/\delta)$

- **算法上**：对于最大化问题，取 $T$ 次重复后的**最大值**

# 完整的最大割算法

为达到 $\delta$ 失败概率,  $0.5 - \epsilon$ 近似比:

for  $i = 1, \dots, T$

形成一个 $S_i$ : 对每个 $u \in V$ 以0.5概率独立决定是否放入 $S_i$

返回 $\arg \max_{S_i} |\text{cut}(S_i)|$

$T$ 应该取 $O\left(\frac{\log(1/\delta)}{\epsilon}\right)$

# 编程实现与理论之间的gap

我们从理论上推导出T应该取 $O\left(\frac{\log(1/\delta)}{\epsilon}\right)$

- 这只是一个upper bound，会不会太松？
- 就算这个bound是紧的，那也只是最坏情况，实际中可以远小于这个bound

更重要的是应用上的习惯不同：

- 应用中经常不是指定 $\epsilon, \delta$ 反推T，而是直接指定具体数值

比如，计算资源受限（1s内要运行完），那么T能取的最大值基本是固定的（比如 $T = 500$ ）

# 实际中如何选取重复次数T?

如果有严格资源限制，比如时间1s

- 根据此限制最大限度设置T

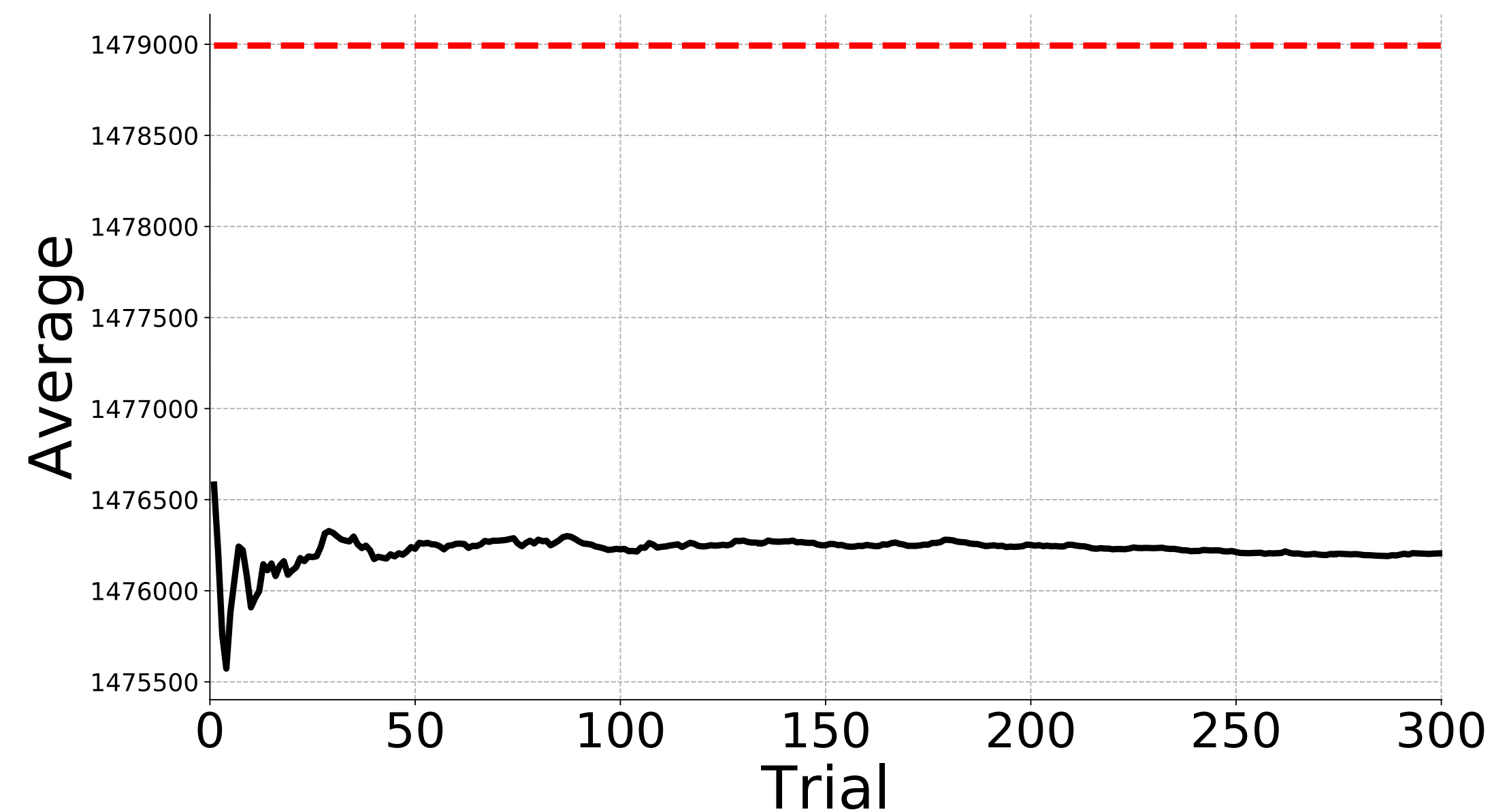
如果无硬性资源限制，可以做dynamic averaging，然后看变化趋势

- dynamic averaging：横轴是重复到第j次，纵轴是前j次目标函数的平均
- 当看到曲线逐步平稳，即可停止
- 不看曲线：算法中可设置一个阈值，当变化量连续3轮小于该阈值就可停止

# 作业一：最大割随机近似算法实现、调优

分值：2分

- 作业一是一个实验报告（作业要求在教学网和课程网站）
- 给定测试数据，实现课上讲的最大割算法，并绘制重复次数-平均代价图



建议用python的matplotlib画图  
作业题目里给出了画图的样例代码

# 作业二： 最大割

分值： 1分

- 实现最大割算法，使得在多组数据上测试都必须输出0.45近似的割
- 在openjudge评测，只有所有测试用例都通过才算通过
- 标程使用固定重复次数在设定时间内可以通过
- <http://cssyb.openjudge.cn/24hw2/>

**Median Trick:**

**一种一般的提升成功概率的办法**



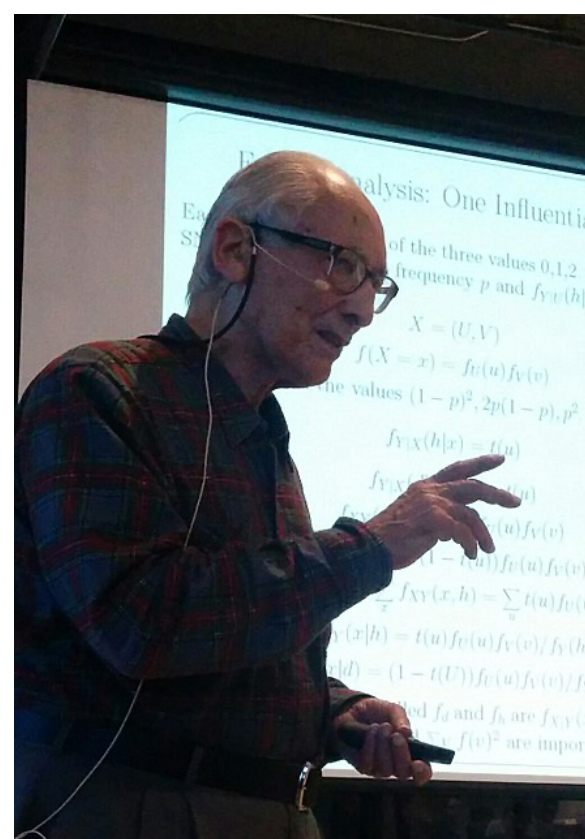
# Median Trick

- 现在有一个黑盒能以  $p > 0.5$  概率正确回答某个 Yes/No 问题的答案
- 如何将该概率强化成任意的  $1 - \delta$ ?
- 期望看到：若重复  $T$  次，正确答案会占多数，即超过  $pT$  个
- 算法：重复  $T$  次，选取占多数（或者处于中位）的答案
- $T$  选多大才够？

# Chernoff Bound

- 我们的新朋友Chernoff

User-friendly版本



Herman Chernoff  
1923 -

**Chernoff bound.** 设 $X_1, \dots, X_n \in [0,1]$ 是独立、同期望 $\mu$ 随机变量

令 $X := (X_1 + \dots + X_n)/n$ 。对任何失败概率 $\delta \in (0,1)$ ，有

$$\Pr \left[ |X - \mu| \geq \sqrt{\frac{\log(1/\delta)}{n}} \mu \right] \leq \delta$$

$n = O(\log(1/\delta))$ 就够了

- 抛多少次硬币，可以使得有 $1 - \delta$ 概率有45% - 55%正面朝上？

# 回到Median Trick

为什么得是严格大于0.5? 等于呢?  
可能小于吗?

- 现在有一个黑盒能以 $p > 0.5$ 概率正确回答某个Yes/No问题的答案

- 如何将该概率强化成任意的 $1 - \delta$ ?

即使对于优化问题，若不保证误差是单侧的（有时高估，有时低估），那么也需要median trick

- 期望看到：若重复 $T$ 次，正确答案会占多数，即超过 $pT$ 个

- 算法：重复 $T$ 次，选取占多数（或者处于中位）的答案

- 分析：比如 $p = 0.51$ ，那么设 $X_i \in \{0,1\}$ 代表第 $i$ 次重复的Yes/No结果

- Chernoff告诉我们 $T = O(\log 1/\delta)$ 足够

# 再论失败概率 $\delta$

- 在之前的slides, 我们讨论了如何选取 $\delta$
- 事实上: 常数 $\delta$ 就通常“不失一般性”
  - 对于某个 $n$ , 经过 $O(\log n)$ 次重复, 可以达到 $1/\text{poly}(n)$ 的失败率
- 多大的“常数”?
  - 如max-cut等可以取最大来放大成功率的: 可以是任何常数
  - 如果需要用median trick, 则必须是  $> 0.5$  (注意严格不等号)

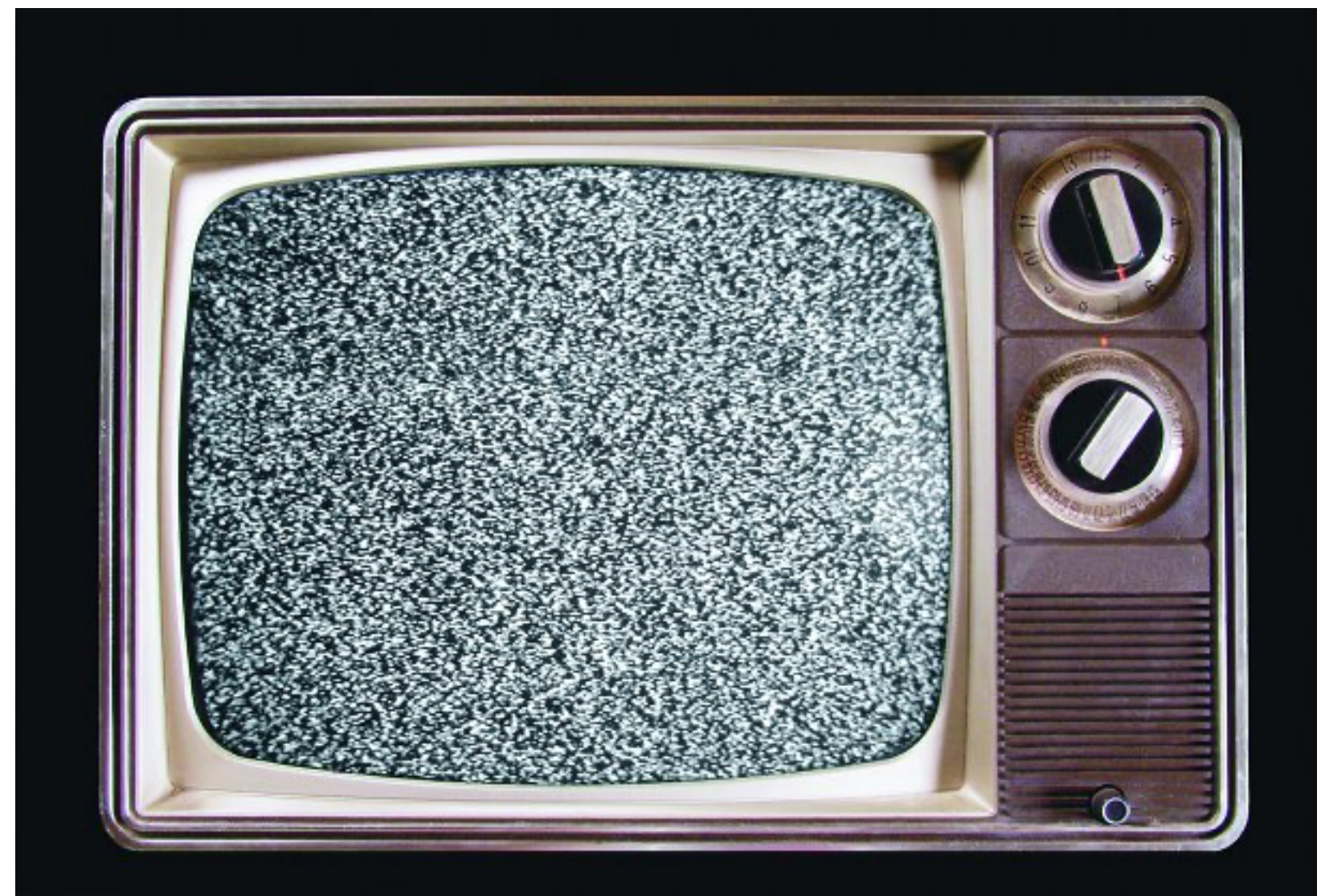
这对于多次运行也通常足够了, 毕竟一般只需要运行 $\text{poly}(n)$ 次

# C++的随机库



# 计算机程序中的随机性

- 真随机：需要从某个物理过程去产生



例如收集宇宙微波背景辐射产生的噪声

# /dev/random

- 摘抄自Linux内核文档：

比如用户的键盘鼠标输入；比如网络/  
硬盘的数据传输

The random number generator gathers **environmental noise** from device drivers and other sources into an entropy pool. The generator also keeps an estimate of the number of bits of noise in the entropy pool. From this entropy pool random numbers are created.

random.org



# 我们需要真随机吗？

- 真随机自然是好，但是采集起来比较麻烦/费时，完全依赖真随机会限制程序的效率
- 对于实际运行随机算法设计来说，实验证明并不十分需要
- 但很多密码学应用需要真随机来确保安全性
- 数值模拟（物理过程）时很多时候也需要真随机来确保良好的效果
- 因此，很多时候可以使用伪随机数

# 伪随机数

- 伪随机数：给定初值 $X_0$ ，通过某个确定性的函数 $f$ 来生成 $X_{n+1} := f(X_n)$
- 一旦初值 $X_0$ （又称种子）确定之后，那么整个序列也随之确定
- $X_0$ 可以使用生成代价较大的真随机数

对随机算法来说这不必须，而重要的是 $f$   
看上去足够随机

- 只需要设计输出 $[0,1]$ 上的均匀随机就可以，其他分布通过这个转化得到

# 伪随机数生成器




- 一般而言，f都是形如 $f' \bmod m$ ，且m取不大于机器word size (32位 $m \leq 2^{32}$ )
- 输出的随机数 $f / m$ 来达到在 $[0,1]$
- 挑战在于需要尽量“像”均匀分布；一个必要条件：
  - 不能（很快）重复，比如1 3 4 2 1 3 4 2
  - 也就是周期要长

一般来说只有周期长度的“前半部分”随机性表现更好，因此实际应用如果使用生成器n次那么尽量让周期远大于n

# 经典伪随机数生成器

- 线性同余生成器  $X_{n+1} = (aX_n + c) \bmod m$ 
  - 经过精心选取a、c和m，可以做到周期 =  $m - 1$
  - 也有考虑平方同余
- Lagged Fibonacci:  $X_n = (X_{n-m} + X_{n-l} + c) \bmod m$ 
  - 可以达到更大的周期，例如一些典型选取可以达到  $2^{50} \cdot m$  的周期

# 复合已有伪随机生成器

- 给定一个（或多个）基础生成器（如线性同余），可通过复合得到新的生成器
- Discarding：每隔 $j$ 个数输出一个数  会降低周期，但是如果周期本来很长，使用这种技术可以提高随机性的表现
- Shuffling：利用自身生成的随机数当作下标来挑选下一个输出序列中的哪个数
- 聚合：将若干个输出数拼接成一个输出数  经常用于线性同余生成器来提升随机性  
 要特别小心，拼接后未必有更好的效果

# C++的随机库 (C++ 11)

以上提到的都在C++标准库有体现

```
#include <random>
```

<https://en.cppreference.com/w/cpp/numeric/random>

# 真随机数生成器： Random Device

Defined in header `<random>`

---

```
class random_device;    (since C++11)
```

---

- random\_device会尝试使用系统提供的真随机数例如/dev/random
- 主要用来为其他伪随机数生成器提供种子

# 基本伪随机数生成器

Defined in header `<random>`

线性同余

**linear\_congruential\_engine** (C++11)

**mersenne\_twister\_engine** (C++11)

一种较新的生成器，周期很长

**subtract\_with\_carry\_engine** (C++11)

Lagged Fibonacci

不要直接用这几个基本生成器，应该用后面提到的包装后的生成器



# 伪随机数发生器的复合

Defined in header `<random>`

**discard\_block\_engine** (C++11)

每隔若干元素后只保留一部分元素

拼接/截短生成的随机数

**independent\_bits\_engine** (C++11)

**shuffle\_order\_engine** (C++11)

用随机序列本身生成下标来取对应下标的随机数

这几个也无法直接用，应该用后面提到的包装后的生成器

# 预设的标准随机数生成器

- 基于线性同余的

---

`minstd_rand0 (C++11)`

---

`minstd_rand (C++11)`

---

周期都大约为 $2^{32}$ ，且只能生成int32

---

`knuth_b (C++11)`

---

shuffle版的minstd\_rand0

- minstd\_rand系列性能接近，minstd\_rand随机性略好于minstd\_rand0
- knuth\_b的shuffle有一定计算代价，但有更好的随机性
- 这些算法“随机性”对于实现随机算法来说一般足够

# 预设的标准随机数生成器

- 基于Mersenne Twister的

`mt19937(C++11)`

生成int32

mt19937\_64版本可以生成int64

- 相比线性同余，该方法一般更快
- 虽然周期很长，多数情况下随机性保证不错，但有些测试无法通过，随机性要求高的场合会有问题
- 实际应用十分广泛，是很多语言/工具的默认实现

# 预设的标准随机数生成器

- Lagged Fibonacci

ranlux24(C++11)

---

ranlux48(C++11)

分别可以生成int32和int64

- 是对应的base版本的加强，修复随机性漏洞
- 有很好的随机性保证，但运行效率较低，对实现随机算法不推荐

# 如何评价生成器的优劣？

- 运行效率：对于实现随机算法来说至关重要
- “随机性”：看上去有多么像是随机的？
- 只是周期长还远远不够，还需要符合均匀分布的各种统计性质
  - 例如，每单个元素、每个元素pair/triple...出现都比较独立且频率类似
- 金标准之一：Spectral test

... an especially important way to check the quality of linear congruential random number generators. **Not only do all good generators pass this test, all generators now known to be bad actually fail it.** Thus it is by far the most powerful test known

# 关于Spectral Test

- Spectral test有一个参数 $t \geq 2$ ，测试的是整个随机序列每连续 $t$ 个数构成的 $t$ 元组之间的“独立性”
  - 如果是真随机的，那么不论 $t$ 取多少，都是完全独立的
  - 但是伪随机会随着 $t$ 增大，独立性变差
  - 具体如何衡量这种独立性比较复杂，这里略去

细节见Section 3.3.4, TAOCP II (the art of computer programming), by Donald Knuth

# 一些Spectral Test结果

19、20对应  
minstd

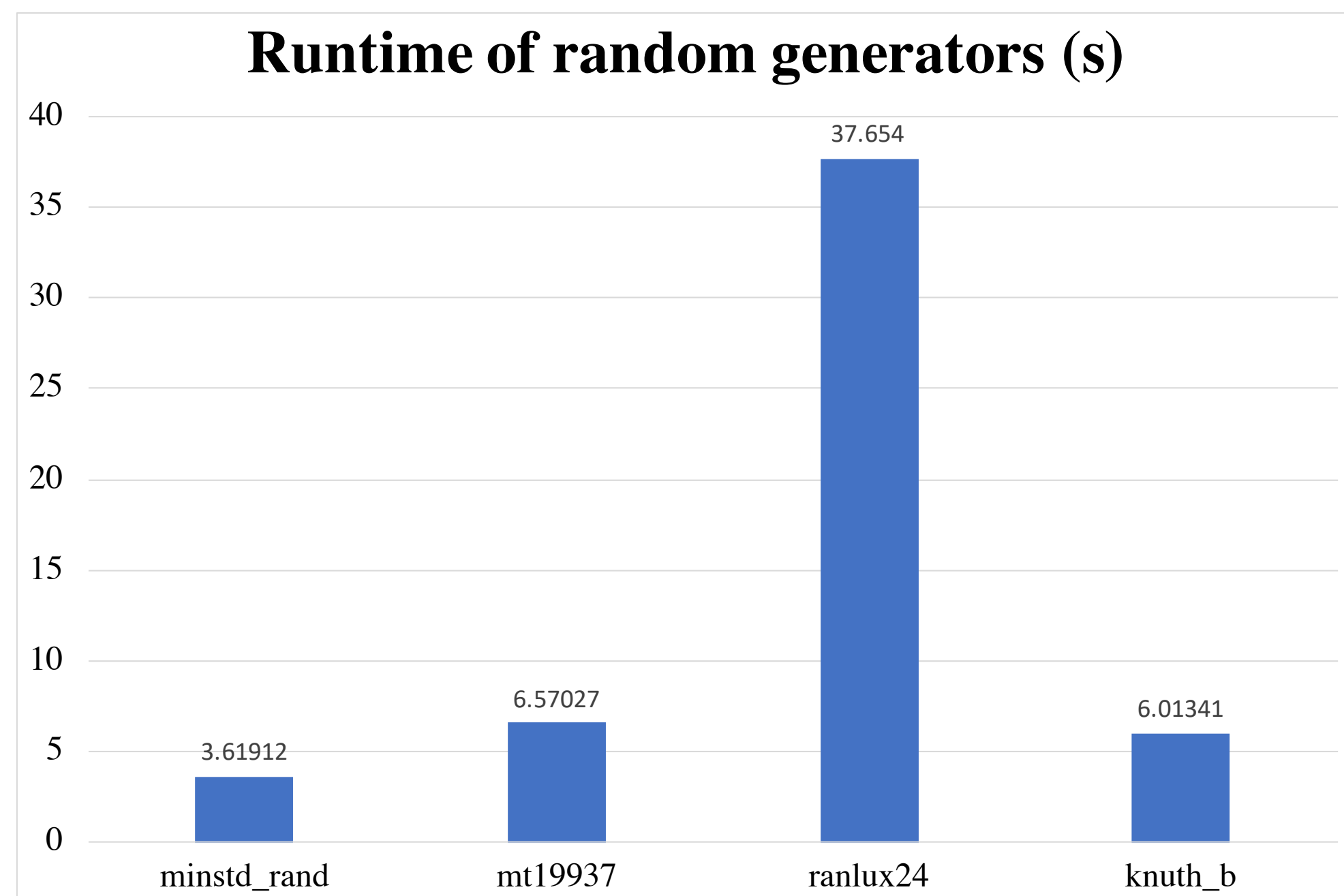
Line	$a$	$m$	$\nu_2^2$	$\nu_3^2$	$\nu_4^2$	$\nu_5^2$	$\nu_6^2$
19	16807	$2^{31}-1$	282475250	408197	21682	4439	895
20	48271	$2^{31}-1$	1990735345	1433881	47418	4404	1402
28	$2^{-24.389}$	$\approx 2^{576}$	$1.8 \times 10^{173}$	$3.5 \times 10^{115}$	$4.4 \times 10^{86}$	$2 \times 10^{69}$	$5 \times 10^{57}$

ranlux

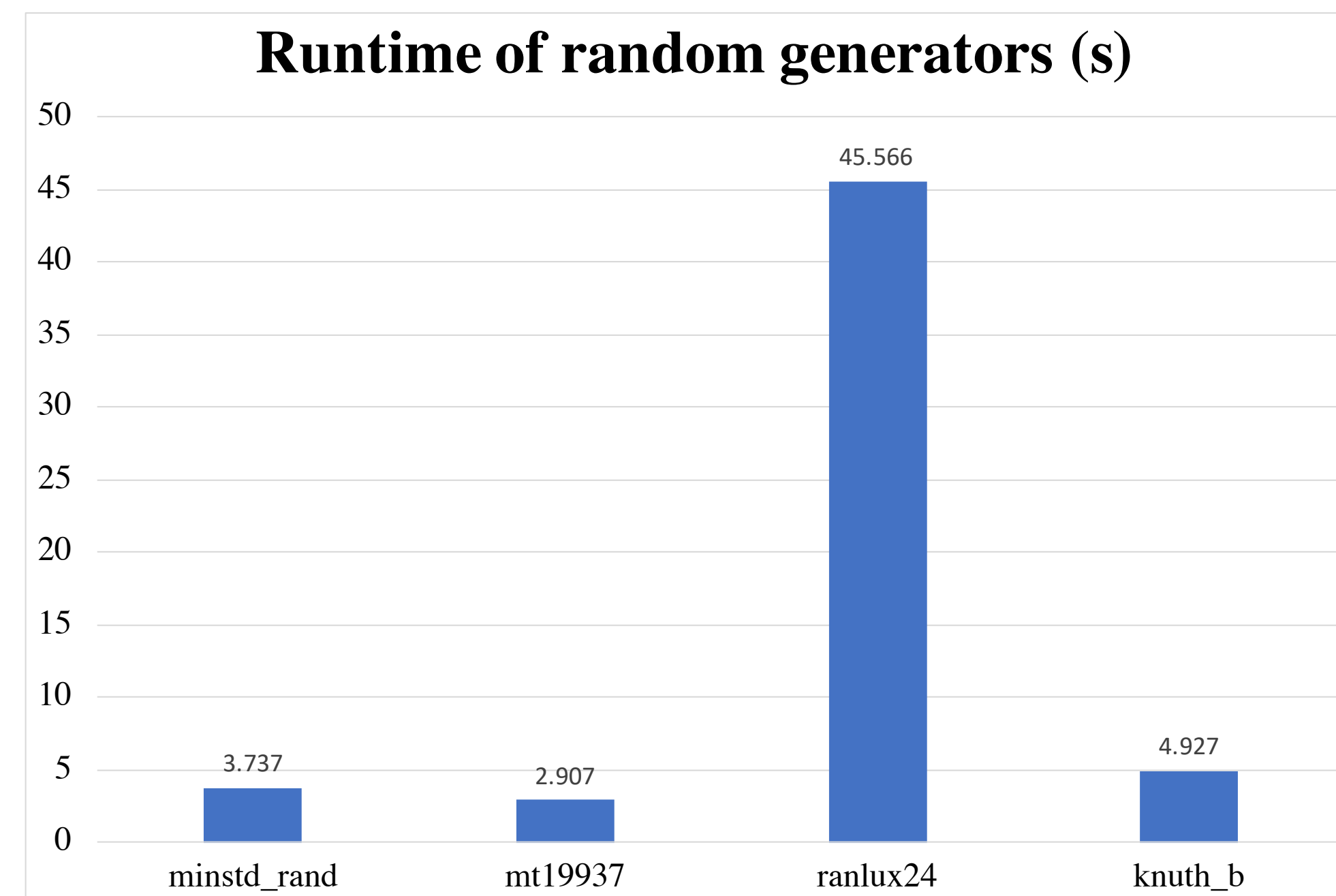
- $\nu_2^2$ 至 $\nu_6^2$ 列都是越大随机性越好，代表的是2-6长度的子列之间的独立性程度

# 伪随机数：时间测试 & 推荐

i7 Mac, clang++ -O2



M2 Mac, clang++ -O2



- 随机性上各个生成器都胜任本课程应用，因此**本课程推荐**最简单的minstd\_rand
- **脱离本课程，今后实际应用时应该再做仔细评估（尤其是对随机性的要求）**



# 程序示例

```
#include <iostream>
#include <random>

int main()
{
    // 如果需要真随机种子, 则使用random_device来初始化
    // std::random_device rd;
    // std::minstd_rand gen(rd());

    // 默认不需要真随机种子
    std::minstd_rand gen;
    for (int i = 0; i < 20; i++)
    {
        std::cout << gen() / (gen.max() + 0.0) << std::endl;
    }
    return 0;
}
```

除以gen.max()来得到[0, 1]随机数

# 从特殊分布采样

- 均匀分布

## Uniform distributions

<code>uniform_int_distribution</code> (C++11)	produces integer values evenly distributed across a range (class template)
<code>uniform_real_distribution</code> (C++11)	produces real values evenly distributed across a range (class template)

- 其他重要分布

## Bernoulli distributions

<code>bernoulli_distribution</code> (C++11)
<code>binomial_distribution</code> (C++11)
<code>negative_binomial_distribution</code> (C++11)
<code>geometric_distribution</code> (C++11)

## Normal distributions

<code>normal_distribution</code> (C++11)
<code>lognormal_distribution</code> (C++11)
<code>chi_squared_distribution</code> (C++11)
<code>cauchy_distribution</code> (C++11)
<code>fisher_f_distribution</code> (C++11)
<code>student_t_distribution</code> (C++11)

## Poisson distributions

<code>poisson_distribution</code> (C++11)
<code>exponential_distribution</code> (C++11)
<code>gamma_distribution</code> (C++11)
<code>weibull_distribution</code> (C++11)
<code>extreme_value_distribution</code> (C++11)

# 编程实现

## 举例：Bernoulli分布

```
#include <iostream>
#include <random>

int main()
{
    std::minstd_rand gen;
    std::bernoulli_distribution d(0.25);
    for (int i = 0; i < 20; i++)
    {
        std::cout << d(gen) << std::endl;
    }
    return 0;
}
```

# 从一般离散分布采样

## Sampling distributions

### `discrete_distribution` (C++11)

```
#include <iostream>
#include <random>

int main()
{
    std::minstd_rand gen;
    // 初始化列表是一个浮点数组A, 设A有n个元素, 则1 <= i <= n将以A[i] / \sum_j A[j]概率被采样
    std::discrete_distribution<> d({1.5, 1.5, 1.5, 3.0});
    for (int i = 0; i < 20; i++)
    {
        std::cout << d(gen) << std::endl;
    }

    // 输出对应的概率值
    std::cout << std::endl;
    for (double i : d.probabilities())
    {
        std::cout << i << std::endl;
    }
    return 0;
}
```

# 注意

- 随机数生成器的随机性保证比较特殊，很多操作可能会导致随机性变差
  - 一个典型例子：用`gen() % 10000`生成`[0, 9999]`随机数
  - 这是有问题的，`% 10000`之后周期可能变短 (因为本来需要mod一个合适的m)
  - 一般没问题的方法是`gen() / gen().max() * 10000`再取整
- 尽可能使用标准库自带的生成器来避免这些坑！

# 其他常见随机结构的生成

## 随机排列Shuffling

- 随机排列（算法复杂度线性）

```
#include <iostream>
#include <random>
#include <algorithm>

int main()
{
    std::minstd_rand gen;
    int A[] = {1, 2, 3, 4, 5};
    // shuffle函数接受begin和end以及一个generator, 就地将[start, end)里面的元素进行random shuffle
    std::shuffle(A, A + 5, gen);
    for (int i = 0; i < 5; i++)
    {
        std::cout << A[i] << std::endl;
    }
    return 0;
}
```

- 不要使用较老的random\_shuffle，因为无法使用新的随机库

# 如何实现随机排列?

- 设 $A_1, \dots, A_n$ 是输入数组

一个常犯错误：取 $k \in [1, n]$

- 算法：for  $j = n \rightarrow 1$ , pick random  $k \in [1, j]$ , swap( $A_j, A_k$ )
- 验证充分必要条件：每种排列出现的概率都是 $1/n!$
- 如果用错误的 $k \in [1, n]$ 取法为什么不是均匀的排列?

# 典型随机算法选讲



# 快速测试矩阵相乘结果是否正确

- 给出三个  $n \times n$  实矩阵  $A$   $B$   $C$ ，测试是否  $AB = C$
- 确定性/暴力算法：  $O(n^\omega)$  时间，现在  $\omega \approx 2.37188$
- $O(n^2)$  时间随机算法：如果  $AB = C$  那么一定返回 Yes；但是  $AB \neq C$  时可能答错
  - 随机选取一个  $n$  维随机  $\{0, 1\}$  向量
  - 测试是否  $ABx = Cx$ ：是则输出 YES 否则 NO

达到  $\omega = 2$  是重大 open question

可以在  $n^2$  时间计算  $ABx$ ：先算  $y = Bx$ ，再算  $Ay$

# 正确性

- 如果 $AB = C$ :  $ABx$ 一定等于 $Cx$
- 如果 $AB \neq C$ 
  - 考虑简单情况 $n = 1$ , 也就是 $A$   $B$   $C$ 都是实数（而不是矩阵）
  - 比如 $AB = 1$ ,  $C = 2$ , 那么不超过0.5概率会判断为相等

尝试分析 $n = 2$ ?

# 如何提高成功率？

- 只运行一次的话，相等时一定成功，不相等时成功率只有0.5
- 可以考虑多次运行；但如何汇总结果呢？
- 看到这个序列，应该得出何种结果：YES YES YES YES NO
- 典型情况下，我们“期望”看到什么序列呢？

# 典型情况

- 如果相等：那么只会看到Yes Yes Yes...
- 如果不相等：至多一半是Yes, 至少一半是No, 但是实际上可以略微浮动
- 判据：重复 $T$ 次，如果有No就回答No，否则回答Yes
  - 分析：如果有 $p \leq 0.5$ 概率输出Yes, 那么没有No的概率是 $p^T$
  - 要想达到 $1 - \delta$ 成功率，只需要 $T = \log 1/\delta$

# 作业三： 测试矩阵相乘是否正确

分值： 1分

- 实现课上讲的随机检测方法
- 在openjudge评测， 只有所有测试用例都通过才算通过
- 标程使用固定重复次数在设定时间内可以通过
- <http://cssyb.openjudge.cn/24hw3/>

# Rejection Sampling

## 一个抛硬币问题

- 假设我们有一个均匀硬币，利用该硬币生成一个 $1/4$ 概率的 $\{0, 1\}$ 两点分布
  - 算法是什么？需要抛多少次？
- 如果要求是 $1/3$ 呢？
  - 注意到：抛掷有限次然后根据抛掷结果输出 $\{0, 1\}$ 是不可能得到 $1/3$ 的！

# Rejection Sampling

## 1/3的解法

- 考虑抛两次的情况，HH TH HT TT四种可能，以不出现TT为条件，则剩下每种出现的概率就是1/3的
- 算法：尝试连续抛掷两次硬币，若TT则重新抛，否则当HH时返回1其他返回0
- 需要抛掷多少次？（需要分析数学期望；最坏情况可以抛任意多次）

TT被reject掉了

本质上：rejection sampling实现了条件概率，但如果reject太多那么会影响性能

比如只有1%的概率不reject，那就大约采样100次才能停下来

# 作业四： Rejection Sampling

分值2分

- 题目：推广刚刚的rejection sampling，对于一般的一个目标概率 $p$ 生成两点分布
  - 请注意效率！
- 本题为代码填空/交互题，需要使用我们提供的随机性（即 $\{0, 1\}$ 均匀分布）
  - 请不要自己在函数中利用其他随机数发生器
- <http://cssyb.openjudge.cn/24hw4/>



# 一个亚线性算法问题：求中位数

- 中位数问题：输入 $n$ 个整数 $A = \{a_1, \dots, a_n\}$ ，求这 $n$ 个数的中位数
  - 中位数：将这些数排序后，居于最中间的，也就是第 $\lceil n/2 \rceil$ 位的数
- 传统上：
  - 可以排序然后直接统计（但需要 $O(n \log n)$ 复杂度）
  - 另外存在一个线性 $O(n)$ 时间的分治算法
  - 不论哪种方法，至少都需要存储、访问整个数据集

# 亚线性设定

- 然而，如果不允许访问所有数据，仅可以随机访问某些元素呢？
  - 例如，这些数分布式存储在非常多的机器、硬盘上，无法载入内存
  - 但可以用较小的代价去访问指定的某第 $i$ 个元素
  - 即，算法可以使用一种查询，该查询每次提交一个 $i$ ，并得到 $a_i$ 的值
- 是否能用显著小于 $O(n)$ 次的访问来得到对中位数的估计呢？例如 $O(\log n)$ ？

# 一个基于采样的亚线性算法

只需要 $O(1/\epsilon^2)$ 次采样就能得到 $\pm \epsilon n$ 位误差的估计

- 给定一个很小的误差限 $\epsilon$ （例如0.01）
- 算法：均匀独立采样 $T := O(1/\epsilon^2)$ 次，找这个采样上的中位数并返回
- Claim：以大常数概率，算法返回的数排在 $(0.5 \pm 2\epsilon)n$ 位上

注意到 $0.5n$ 位是“精确解”，这里有一个 $2\epsilon n$ 位次的误差

# 简要分析

**Chernoff bound.** 设 $X_1, \dots, X_n \in [0,1]$ 是独立、同期望 $\mu$ 随机变量

令 $X := (X_1 + \dots + X_n)/n$ 。对任何失败概率 $\delta \in (0,1)$ ，有

$$\Pr \left[ |X - \mu| \geq \sqrt{\frac{\log(1/\delta)}{n}} \mu \right] \leq \delta$$

- 设 $\alpha = 0.5 - 2\epsilon$ ,  $\beta = 0.5 + 2\epsilon$
- 考虑三个子集:  $A_1$ 为排序后在 $[1, \alpha n]$ 位次的数,  $A_2 = [\alpha n, \beta n]$ ,  $A_3 = [\beta n, n]$
- 现在考虑一个均匀采样 $i$ 
  - 各有 $0.5 - 2\epsilon$ 概率 $i \in A_1$ 和 $i \in A_3$
- Chernoff bound说明大概率 $A_1$ 和 $A_3$ 各被采到至多 $(0.5 - \epsilon)T$ 个点
- 因此有大概率样本中位数落在 $A_2$ 上, 即某个排位在 $(0.5 \pm 2\epsilon)T$ 的数

典型情况下,  $T$ 次采样各有 $(0.5 - 2\epsilon)T$ 个点落在 $A_1$ 和 $A_3$

问题:  $T$ 应该取多少?

# 作业五： 亚线性时间估算分位点

分值： 2分

- 作业题推广到一半的 $p$ 分位点（中位数是 $p = 0.5$ 的特例）
- 本题为代码填空/交互题，需要用我们提供的查询器来访问数据
- 最后的性能指标不（只）看运行时间，主要看查询次数
- 在标程查询次数的一定范围内都可以通过
- <http://cssyb.openjudge.cn/24hw5/>

# 非均匀采样构造数据摘要问题

## 从均匀采样的局限性说起

- 考虑随机采样的时候，**均匀采样**是第一个该考虑的
  - 算法设计的一般原则：有简单的就不用复杂的；简单的不够再考虑复杂的
- 均匀采样的好处：一般确实可以得到**无偏估计**，即 期望 $E[X]$  是正确的
- 不足：未必可以做到 **常数概率** 落在期望附近！

奥卡姆剃刀

# 均匀采样何时不适用？

- 例如：考虑  $a_1 = a_2 = \dots = a_{n-1} = 0, a_n = 1$
- 通过均匀采样  $a_i$  来估计这列数的和：需要  $\Omega(n)$  次采样，即使容易构造无偏估计

即使想要以常数概率采到一次非0的  $a_n$   
都需要  $\Omega(n)$  次采样！

# 一个例题：1-median的数据摘要

- 输入为n个整数  $A := \{a_1, \dots, a_n\}$
- 要求预处理之后高效回答查询：给定整数  $q$ ，返回

$$\text{cost}(A, q) := \sum_{i=1}^n |q - a_i|$$

这个叫做1-median查询， $q$ 就是一个候选的median点

- 本题虽然存在基于平衡树等结构的做法，但这里介绍一种采样做法



# 摘要数据结构

我们的算法要找到一个 $S \subseteq A$ 以及对应的权重 $w(x)$ ，使得

$$\forall q, \quad \sum_{a \in S} w(a) \cdot |q - a| \in (1 \pm \epsilon) \cdot \text{cost}(A, q)$$

即 $S$ 是 $A$ 的一个数据摘要，使得任何查询点 $q$ 上的1-median值都能得到近似

# 如何构造S?

- 刚刚提到，均匀采样是不行的，会有 $a_1 = \dots = a_{n-1} = 0, a_n = 1$ 的数据
  - 这种数据要求我们要对 $a_n$ 有更高的采样概率
- 思路：
  - 首先考虑一种均匀采样适用的子集（并在上面做均匀采样构造摘要）
  - 然后将数据集划分成满足上述性质的若干子集，分别构造摘要

# 特例：“环”形子集上的摘要

设这个数据集叫做 $P$

- 考虑一种特殊情况：所有数据点都到某个固定的点 $c$ 距离在 $[r, 2r]$ 之间 ( $r > 0$ )
- 考虑一个均匀采样算法：

几何上这是一个圆环

$m$ 为待定参数

  - 均匀选取 $P$ 中的 $m$ 个点，每个点设置权重 $|P|/m$ ，设该随机集合为 $S$
- 因此给一个查询 $q$ ，我们的估计量就是

$$\sum_{a \in S} w(a) \cdot |q - a|$$

# 期望分析：无偏估计

结论：  $\forall q,$   $\mathbb{E} \left[ \sum_{a \in S} w(a) \cdot |q - a| \right] = \text{cost}(P, q)$

计算过程：

$$\mathbb{E} \left[ \sum_{a \in S} w(a) \cdot |q - a| \right] = \frac{|P|}{m} \cdot m \cdot \frac{1}{|P|} \cdot \sum_{a \in P} |q - a| = \text{cost}(P, q)$$

# “环”的性质

对  $a \in S$  设  $X_a := |q - a|$ ，那么我们的估计量就可以写成  $X := \frac{|P|}{m} \cdot \sum_{a \in S} X_a$

注意到  $S$  是独立采样，所以任意两项  $X_a, X_b$  是独立的

另外由于“环”的性质：任意两项数值上差别不大：

$$X_a - X_b \leq |q - a| - |q - b| \leq |a - b| \leq 4r$$

利用了绝对值不等式（事实上是三角形不等式）  
此性质恒成立，与随机性无关

# 利用相加误差版本的Chernoff Bound

**Hoeffding inequality.** 设独立随机变量  $X_1, \dots, X_m \in [s, t]$ 。令  $X := \sum_i X_i$ ，则

$$\Pr[X - \mathbb{E}[X] \geq z] \leq 2 \exp \left( -\frac{2z^2}{m(t-s)^2} \right)$$

Chernoff bound 这里是  $t \cdot \mathbb{E}[X]$

经过计算可得  $\sum_{a \in S} w(a) |q - a| \in \text{cost}(P, q) \pm \epsilon |P| r$

说明均匀采样大概率产生相加误差

$\epsilon |P| r$

需要  $1 - \delta$  成功率时应当取  $m = O(1/\epsilon^2 \cdot \log(1/\delta))$

# 如何划分成环及处理相加误差 $\epsilon |P| r$

总结一下之前的：

- $P$ 中的点都到某个 $c$ 距离在 $[r, 2r]$ 之间
- 在 $P$ 上运行均匀采样得到的 $S$ 对任何查询 $q$ 有相加误差 $\epsilon |P| r$
- 解决相加误差：选 $c$ 为**最优的1-median**，即 $c$ 最小化 $\text{cost}(A, c)$
- 此时，选 $r_0 := \epsilon \cdot \text{cost}(A, c)/n$ ， $r_i := r_0 \cdot 2^i$ ，定义 $P_i$ 为 $A$ 中到 $c$ 距离 $[r_i, 2r_i]$ 的点
- 设 $S_i$ 为在 $P_i$ 均匀采样的点集

结论：c取A的中位数即可

把A划分成环！

$P_0$ ，即 $[0, r_0]$ 一段的处理：任选 $P_0$ 中一点作为 $S_0$ ，权重设为 $|P_0|$ ，即“缩点”

# 如何处理相加误差 $\epsilon \mid P \mid r$

此时：每个  $P_i$  上采样的  $S_i$  有相加误差  $\epsilon \mid P_i \mid r_i$

$$\text{总误差} \epsilon \sum_i \mid P_i \mid r_i \leq \epsilon \cdot \sum_i \text{cost}(P_i, c) = \epsilon \cdot \text{cost}(A, c)$$

不等号利用了环的性质：  $P_i$  每个点到  $c$  距离都至少是  $r$

而因为  $c$  最小化了  $\text{cost}(A, \cdot)$ ，因此对任何查询  $q$  有  $\text{cost}(A, c) \leq \text{cost}(A, q)$

所以对任何查询  $q$ ，我们的总误差都是  $\epsilon \cdot \text{cost}(A, q)$  以内



# 总结：完整算法、结论和参数选取

- 算法：

问题：这个算法得到的 $S$ 的大小是多少？

  - 先求使 $\text{cost}(A, c)$ 最小化的 $c$ （可以找中位数）
  - 定义环 $P_i$ 并在每个 $P_i$ 上依照之前进行 $m$ 次均匀采样得到 $S_i$ ，并赋予合适权重
  - 将所有 $S_i$ 求并集得到 $S$ 返回
- 保证：对任何 $q$ ，以概率 $1 - \delta$ 有

可以证明一共有 $O(\log n)$ 个环  
要对所有环用union bound保证同时成功，需要取  
 $m = O(1/\epsilon^2 \log(\log n/\delta))$

$$\text{cost}(S, q) \in (1 \pm \epsilon) \cdot \text{cost}(A, q)$$

# 推广到高维

- 刚刚讲的算法是对于一维输入的，但容易推广到高维
    - 除了 $c$ 的选取，其他地方的算法都完全不需要改动
  - 如何在高维有效选取 $c$ ?
- 可从刚才的推导得出
- 首先：常数近似的 $c$ 只会让 $S$ 的大小增加相应的常数，因此不必找最优的 $c$
  - 算法：均匀随机选取一个数据点作为 $c$ ，则 $\mathbb{E}[\text{cost}(A, c)] \leq 2 \min_{c'} \text{cost}(A, c')$

即期望上是2-近似；可以利用三角形不等式验证

# 作业六： d维1-median查询

分值： 3分

- 作业为维针对 $d = 3$ 维的输入的近似查询
- <http://cssyb.openjudge.cn/24hw6/>