程序设计实习（实验班-2023春）
随机算法及其实现

授课教师：姜少峰
助教：张宇博 楼家宁

Email: shaofeng.jiang@pku.edu.cn

随机算法：
计算过程中利用随机变量的算法

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

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

δ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δ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δ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δ）
  - 如果要运行q次且每次失败概率是δ，那么存在一次失败的概率至多是$q\delta$

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

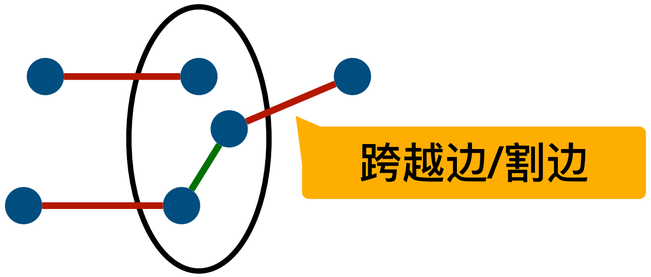
随机算法的设计

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

- 我们预先设定δ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

- 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cut(S)



- 求S使得|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

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

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$\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$

- 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

恰好等于的概率很小

- 因此必须要先设计一个“无偏估计”，即$\mathbb{E}[\mathrm{ALG}]$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：$\mathbb{E}[\,|\mathrm{cut}(S)|\,] \geq 0.5 \cdot |E|$ （期望是2-近似的）

实际上这里更强：cut(S)有图中至少一半的边，这必然大于0.5 OPT

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

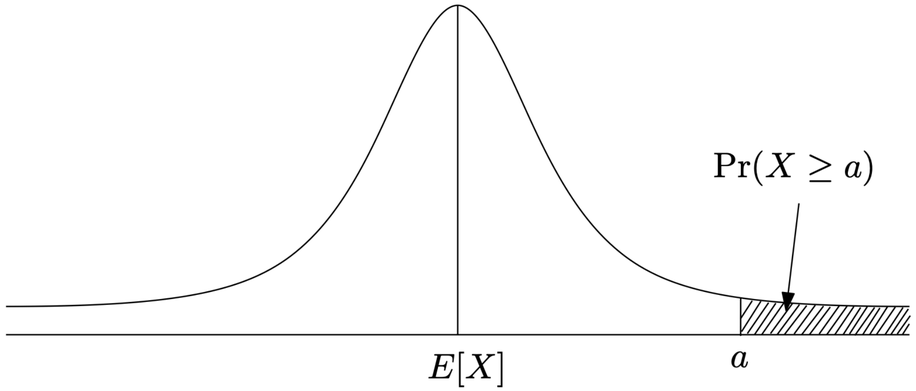
第二步：Pr[ALG is correct] ≥ 常数

（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

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

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



不在割里的边数

- 考虑X = |E| - |cut(S)|，则$\mathbb{E}[X] = |E| - |\mathrm{cut}(S)| \leq 0.5|E|$
- $\Pr[|\mathrm{cut}(S)| \leq (0.5-\epsilon)|E|] = \Pr[X \geq (0.5+\epsilon)\cdot|E|] \leq \frac{1}{1+2\epsilon}$

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$\Pr[\text{ALG is correct}] \geq$ 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：
  - $\Pr[$存在一次ALG is correct$] \geq 1 - 0.9^T$
  - 如果要追求δ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δ失败概率，$0.5-\epsilon$近似比：

  for i =1, …, 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$\arg\max_{S_i} |\text{cut}(S_i)|$

问题：如果不是0.5会怎样?

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

编程实现

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ϵ, δ反推T，而是直接指定具体数值

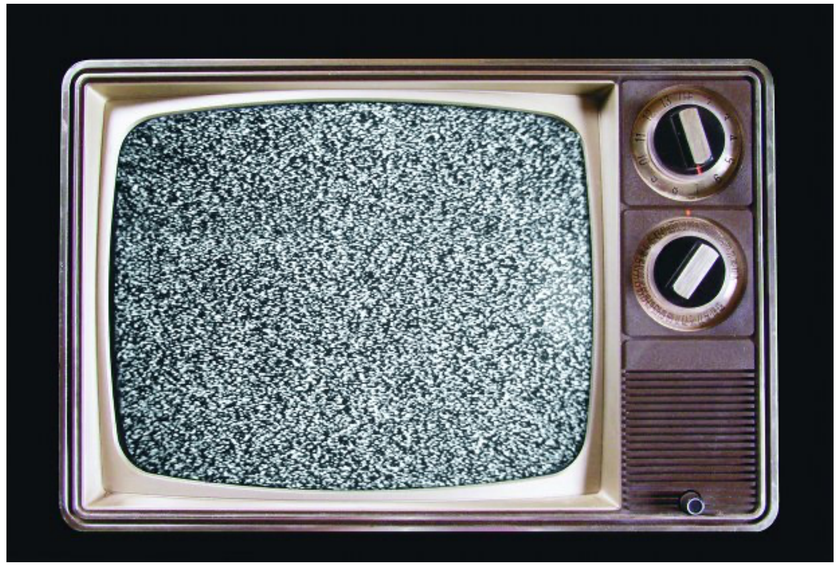
比如，计算资源受限（1s内要运行完），那么T能取的最大值基本是固定的（比如T = 500）

编程实现需要考虑的问题

- C++应该怎样/用什么工具实现随机性
- 如何生成各种分布
- 如何处理实际与理论的gap

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$f' \mod m$，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$X_{n+1} = (aX_n + c) \mod m$
  - 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: $X_n = (X_{n-m} + X_{n-l} + c) \mod m$
  - 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

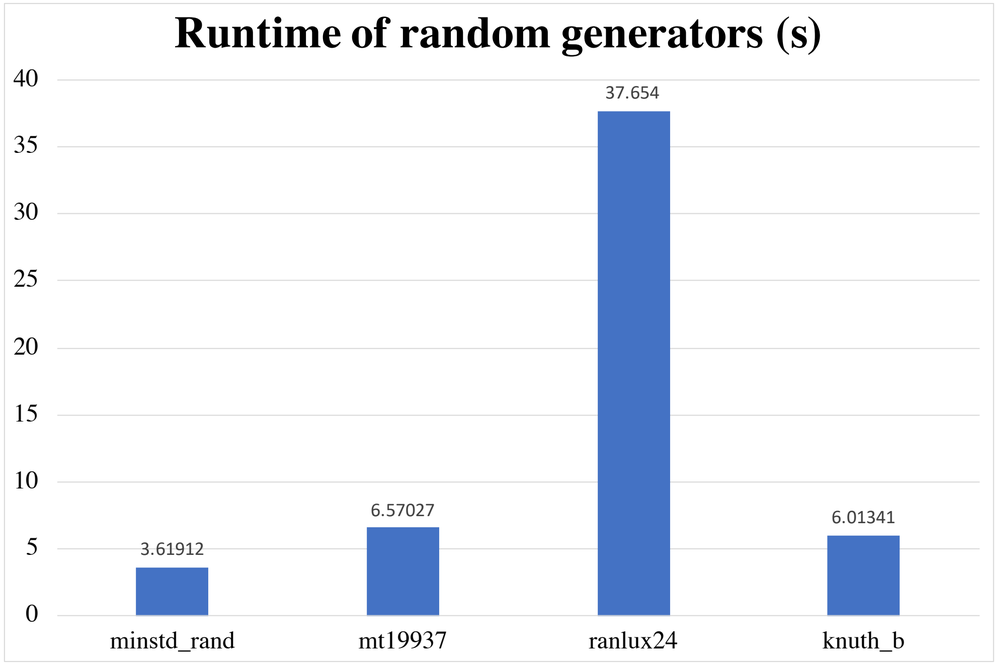
Line	a	m	ν_2^2	ν_3^2	ν_4^2	ν_5^2	ν_6^2
19	16807	$2^{31}-1$	282475250	408197	21682	4439	895
20	48271	$2^{31}-1$	1990735345	1433881	47418	4404	1402
28	$2^{-24\cdot 389}$	$\approx 2^{576}$	1.8×10^{173}	3.5×10^{115}	4.4×10^{86}	2×10^{69}	5×10^{57}

ranlux

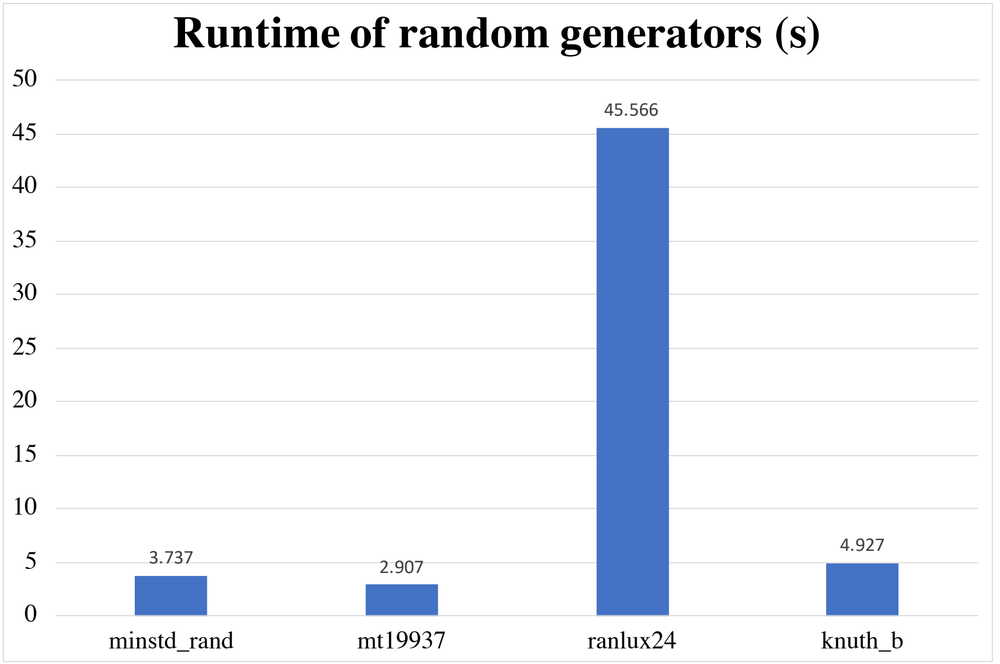
- ν_2^2至ν_6^2列都是越大随机性越好，代表的是2-6长度的子列之间的独立性程度

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

```cpp
#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

`uniform_int_distribution` (C++11)	produces integer values evenly distributed across a range (class template)
`uniform_real_distribution` (C++11)	produces real values evenly distributed across a range (class template)

- 其他重要分布

Bernoulli distributions

- `bernoulli_distribution` (C++11)
- `binomial_distribution` (C++11)
- `negative_binomial_distribution` (C++11)
- `geometric_distribution` (C++11)

Normal distributions

- `normal_distribution` (C++11)
- `lognormal_distribution` (C++11)
- `chi_squared_distribution` (C++11)
- `cauchy_distribution` (C++11)
- `fisher_f_distribution` (C++11)
- `student_t_distribution` (C++11)

Poisson distributions

- `poisson_distribution` (C++11)
- `exponential_distribution` (C++11)
- `gamma_distribution` (C++11)
- `weibull_distribution` (C++11)
- `extreme_value_distribution` (C++11)

编程实现

举例：Bernoulli分布

```cpp
#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)

```cpp
#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;
}
```

* 如何实现这个算法?

- [0, 1]均匀随机数 + 二分查找

- 如果是给定概率密度函数，要求采样连续型随机变量呢?

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）

```cpp
#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genreator, 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$A_1, \ldots, A_n$是输入数组
- 算法：for $j = n \to 1$, pick random $k \in [1,j]$, swap(A_j, A_k)

一个常犯错误：取$k \in [1,n]$

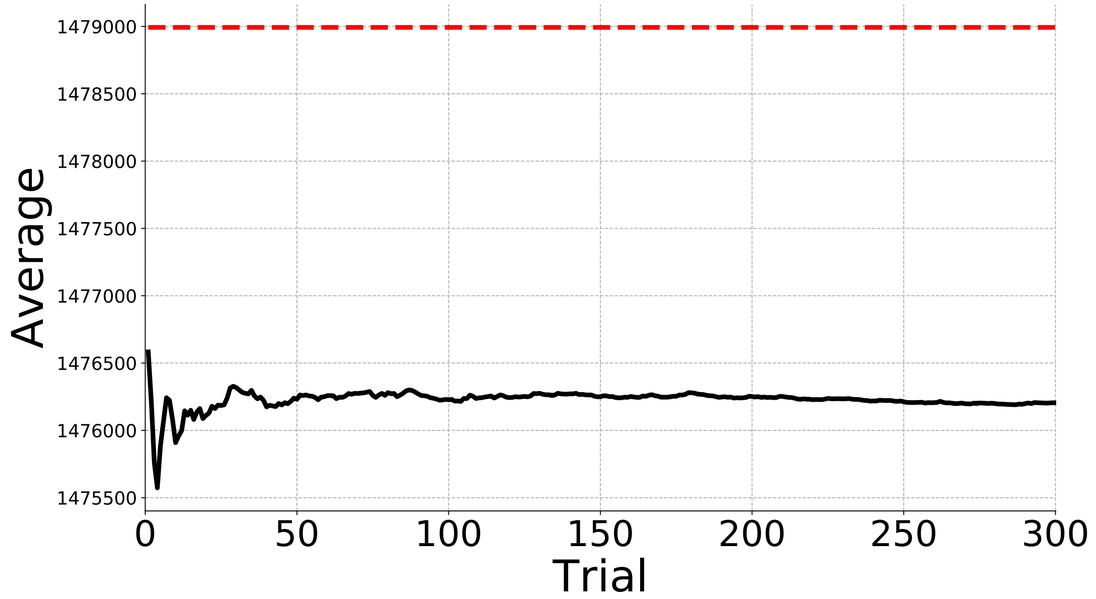
- 验证充分必要条件：每种排列出现的概率都是$1/n!$
- 如果用错误的$k \in [1,n]$取法为什么不是均匀的排列?

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

- 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

更多典型随机算法

快速测试矩阵相乘结果是否正确

- 给出三个n x 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
- 如何选择判据?
- 直觉上：重复T次，如果有No就回答No
- 概率：如果$p \leq 0.5$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$T = \log 1/\delta$

作业二：测试矩阵相乘是否正确

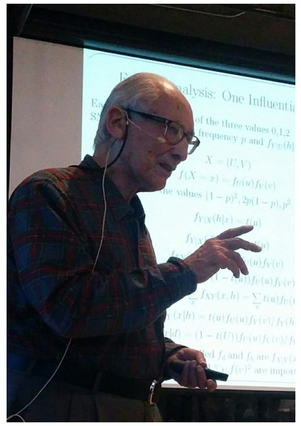
- 实现课上讲的随机检测方法
- 在openjudge评测，只有所有测试用例都通过才算通过
- 标程使用固定重复次数在设定时间内可以通过
- http://cssyb.openjudge.cn/2023hw2/

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



Herman Chernoff
1923 -

Chernoff bound. 设$X_1, \ldots, X_n \in [0,1]$是独立随机变量。

令$X = X_1 + \ldots + X_n$。则对$t \in [0,1]$有

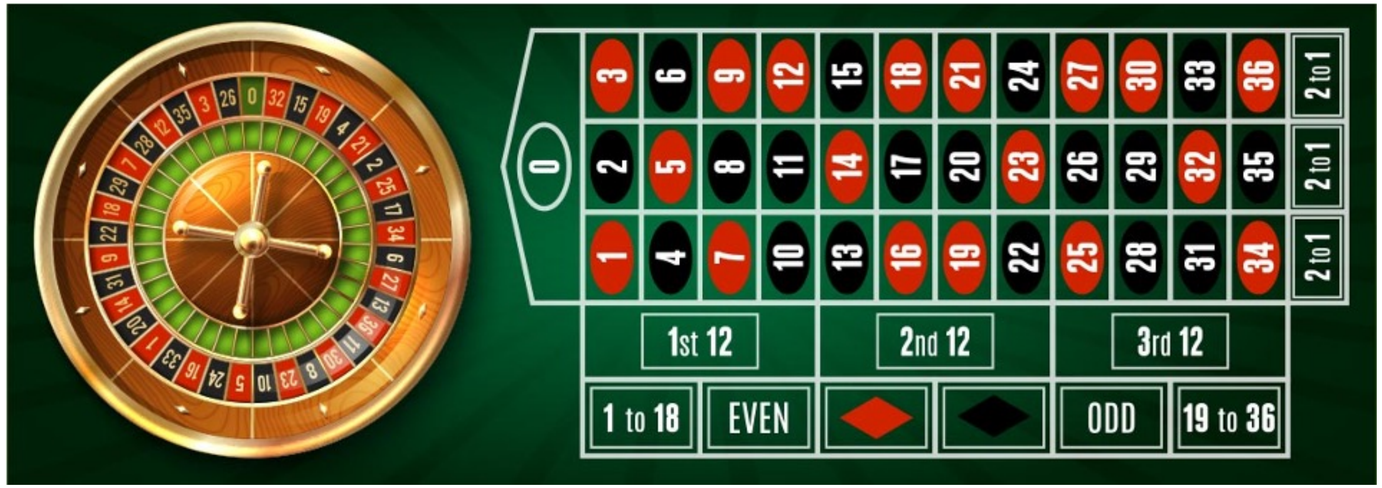
$$\Pr[\,|X - \mu| \geq t\mu] \leq 2\exp(-t^2\mu/3)$$

令t = 0.1, $2\exp(-t^2\mu/3) = \delta$
得到$0.5n = \mu = O(\log(1/\delta))$

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

* 独立性和有界性是必要的

- 赌场的例子：
  - 如果赌场的随机试验不独立会怎样？赌徒如何利用？
  - 赌场一般要求每局赌资有一个上界，为何？



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$1-\delta$?
- 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$X_i \in \{\text{Yes, No}\}$代表第i次重复的结果
- Chernoff告诉我们$T = O(\log 1/\delta)$足够

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

再论失败概率δ

- 在之前的slides，我们讨论了如何选取δ
- 事实上：常数δ就通常“不失一般性”
  - 经过$O(\log n)$次重复，可以达到$1/\text{poly}(n)$的失败率

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

- 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）

* Rejection Sampling

一种实现条件概率的方法

- 用于生成条件概率对应的随机变量
- 举例：假设我们有一个均匀硬币，利用该硬币生成一个1/3概率的两点分布
- 考虑抛两次的情况，HH TH HT TT四种可能，以不出现TT为条件，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）

思考题：如果要生成任意实数p概率 ($p \in (0,1)$)的两点分布呢?

* 蒙特卡洛估计积分

- 现在有一个函数$f:[0,1]\to[0,1]$，并且算法可以任取x询问$f(x)$的值
- 假设每次询问需要花单位时间，算法需要估计$M:=\int_0^1 f(x)\ dx$
- 设计随机算法，对于给定$0<\epsilon,\delta<1$，使得估计值$\widehat{M}$满足

$$\Pr[|\widehat{M}-M|\le\epsilon]\ge 1-\delta$$

随机算法

- 设$X \in [0,1]$是一个$[0,1]$上的均匀随机数
- 那么$\mathbb{E}[f(X)] = \int_0^1 f(x)\ dx$
- 多次试验取平均值：$X_1, \ldots, X_T$，并设$Z := \frac{1}{T}\sum_{i=1}^{T} X_i$
- ~~Chernoff~~告诉我们$T = O(\epsilon^{-2} \log 1/\delta)$就可以达到目标

Chernoff是关于相对误差的；

$\Pr[|\widehat{M} - M| \leq \epsilon] \geq 1 - \delta$ 这个地方需要绝对误差

绝对误差版本的Chernoff Bound

Hoeffding inequality. 设独立随机变量$X_1, \ldots X_n \in [a, b]$。令$X := \sum_i X_i$，则

$$\Pr[X - \mathbb{E}[X] \geq t] \leq 2 \exp\left(-\frac{2t^2}{n(b-a)^2}\right)$$

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

确定性算法必有0.5的误差（即使运行任意长时间）

注意区分“任意”和“无限”；运行无限长的不能叫做“算法”！
类似注意：“任意”和“随机”的区别?

- 确定性算法的具体定义：对于任意确定的输入，必然得到确定的输出
- 具体要证的命题：对任何确定性算法，存在函数f，使得误差 ≥ 0.5
- 设计f: 不论算法问什么x，都设置值$f(x) = 0$；算法不问的点都设置成z (待定)
- 设置z：如果算法最后输出 ≥ 0.5，那么$z = 0$；否则$z = 1$
- 由于算法必须在有限步结束，所以只有有限个特殊0点，故 $\int_0^1 f(x)\ dx = z$

随机算法 vs 确定性算法

- 随机算法就可以作弊：即使对于固定的随机算法，也无法设计出一个确定的函数f让算法总是表现很坏
- 或者说：确定性算法总会遇到很坏的输入，而随机算法可以通过失败概率来“绕过去”
- 即，随机算法也能碰到很坏的输入并得到很坏的输出，但可以把这些“忽略”，当作“失败”来看