# 数据结构与算法I 实验1


姓名：于倬浩

学号：`2019201409`

学院：信息学院


日期：2020.9.28

# 一、题目

1. 实现无放回采样算法。输入为一个数组和正整数k，算法对数组进行操作，前k位为采样结果。需要对采样效率和无放回采样算法的选择进行分析。注：当数组长度较长时采样算法之间的效率差异会比较明显。
2. 实现计算pi的Monte Carlo算法。要使误差小于0.0001，大概需要多少次重复实验?

# 二、算法思路

## a.无放回采样

- 排序生成排列算法：时间效率较低的算法。对于每个元素，生成一个值域在`[1,n^3]`的随机权值，并将所有元素按照随机权值排序，由于每个元素的权值是等概率随机的，其排列后的顺序也是等概率随机的。生成n-排列并排序的时间效率是$\Theta(nlogn)$，空间复杂度$\Theta(n)$。该算法需要完整生成一个长度为n的随机排列，而后续算法只需要生成长度为k的随机排列，即该算法不仅时间复杂度更高，而且进行了更多没有必要的运算。

- 线性生成排列算法：可以参考`Random-In-Place`算法，利用其生成排列的思路。首先读入整个数组，接着，枚举第1~k个元素，每次交换`s[i]`和`s[rand(i,n)]`两个元素，执行结束后，数组的前k个元素即为等概率随机选择的结果。
  - 采样效率：读入所有数据时间复杂度$\Theta(n)$，空间复杂度$\Theta(n)$。生成随机排列的过程中，假设生成单个伪随机数的时间复杂度为$\Theta(1)$，那么生成随机排列的时间复杂度为$\Theta(k)$，额外的空间复杂度$\Theta(1)$。总时间复杂度$\Theta(n+k)$，空间复杂度$\Theta(n)$，n、k同阶时，时间复杂度可认为是$\Theta(n)$。
  - 正确性：该算法和`Random-In-Place`算法的正确性可以使用同样方法证明。首先，每个元素在下标为1的位置的概率相同，都是$\frac{1}{n}$。此后，每个元素在剩下n-1个位置的概率依旧相同，第二次随机选择一个放在下标为2的位置，因此每个元素在下标为2的概率相同。同理可以归纳出，每个元素在第`i`个位置的概率都相同，因此生成的k-排列依旧是等概率随机的。

- 优化算法：上面的朴素做法，时间上已经不再可能有优化，因为读入所有数据的时间复杂度已经是$\Theta(n)$，生成随机排列的时间复杂度为$\Theta(k)$，然而，空间复杂度上仍然有优化的空间。

  我们在读入数据之前，可以先生成无放回排列最终所取到的k个下标位置，将这k个下标从小到大排序，接着，可以使用类似数据流算法的思路，在线读入所有数据，维护一个指针，指向下一个要取到的下标位置，读数据时，如果读到指定的位置，则将该位置的元素保存至数组中，否则继续读取下一个数据。这样，不考虑生成下标的复杂度，算法的时间复杂度变为$\Theta(n+klogk)$，空间复杂度变为$O(k)$，如果n很大，超出了机器内存限制，但是k却比较小，则这种方法可以避开超出内存限制的风险。

  对于生成关键的k个下标，我们依旧沿用朴素算法的思路，假设我们现在有一个`[1,n]`的顺序排列，那么只需要枚举前k个位置和后面的哪些位置交换即可。为了保证空间复杂度和时间复杂度的平衡，我们可以维护一个哈希表，初始为空，从1枚举到k，每次交换即可。如果交换到的位置没有元素，则先在哈希表中插入对应位置的元素，然后再交换，这样减少了大量无谓的空间浪费。哈希表插入、访问的时间复杂度为$\Theta(1)$，插入一个元素所需空间复杂度均摊为$\Theta(1)$，则这部分的时空复杂度均为$\Theta(k)$。

  综上所述，这种算法的总时间复杂度为$\Theta(n+klogk)$，空间复杂度仅为$\Theta(k)$，在n很大，不能保存整个数组，但是k相对较小的情况下，可以使用这种算法。

## b.实现计算$\pi$的Monte Carlo算法

在一定范围`[0,L]`内，指定迭代次数`n`，生成n个坐标`(x,y)`，判断`(x,y)`是否在以原点为圆心，半径为L的圆内。统计圆内的点，即为落在四分之一圆内的点数，答案乘4再除以半径平方即为pi。

# 三、程序设计框架

## a. 无放回采样

我实现了三种算法的对应代码，其中排序生成排列算法对应`1a.cpp`，线性生成排列算法对应`1b.cpp`，优化线性生成排列算法对应`1c.cpp`。

前两者有接口`void random_selection(int s[], int n, int k)`，需要传入已经读入全部数据的数组s，数组长度n和选择的元素个数k，随机结果是`s[1]`到`s[k]`。

第三个程序提供了接口`vector<int> random_selection(int n, int k)`，可以传入数组总长度n和k，接着由该函数在线读取标准输入提供的数据，并返回一个包含了随机结果的`vector<int>`。

三个程序的随机数生成器均选用C++11提供的`std::mt19937`，使用梅森旋转算法生成伪随机数，并使用`std::uniform_int_distribution`生成随机数的均匀分布。

为了方便测试，在前两者的主函数中，我定义了两个长度固定的数组，并将数组元素置为1~n的顺序排列，避免了大量的文件IO操作，可以更好地观察算法消耗的时间。对于第三个算法，由于需要测试的数据n极大，进行磁盘读取操作亦会浪费大量的时间，故选择在代码内指定读入的内容为1-n的顺序排列。测试程序时，只需编译运行即可，无需读取附加文件。

如确需测试文件输入，请在源文件头部定义`#define MANUAL_INPUT`，并利用管道将输入重定向至标准输入流`stdin`。

如需测试程序输出，请在源文件中删除`cout`循环输出部分的注释。

# b. 实现计算π的Monte Carlo算法

实现的算法对应源文件`2a.cpp`。提供了接口`inline double calculate_pi(int iterations)`，可以传入所需迭代的次数，返回估算的pi结果。函数中迭代n次，生成n个随机的坐标，统计圆内的点数来近似表示圆的面积。除以半径平方再*4后即可得到该次估算出的pi的值。

在生成随机数方面，考虑到浮点数本身的有效位数并不多，样本空间的大小也不如整数能表示得多，因此选择生成整数坐标，可以得到更为灵活的随机范围。

# 四、实验结果说明

实验环境：WSL Ubuntu 18.04 x64, Intel Core i7-9750H

```
$ g++ --version
g++ (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
```

# a.无放回采样

对于正确性检验，由于本次作业的随机性，仅从（较小规模的）输出结果很难观察到算法的正确性，在此放过多输出造成篇幅过长，因此这里着重检验不同数据规模下的算法运行效率。

为了防止编译器过度优化或者输出过多信息导致难以观察，程序会输出取到的k个元素的异或和。

- 首先对于第一组数据，选择n=1e7，k=5e6，本组数据主要区分log算法和线性算法的运行效率差异。分别运行三个程序，得到如下输出结果：

```
➜  week4 g++ 1a.cpp -Ofast -o 1a
➜  week4 ./1a
10000000 5000000
[1a] Time elapsed: 921.875ms
Token=15067965
➜  week4 g++ 1b.cpp -Ofast -o 1b
➜  week4 ./1b
10000000 5000000
[1b] Time elapsed: 218.75ms
Token=2155621
```

```
→  week4 g++ 1c.cpp -Ofast -o 1c
→  week4 ./1c
10000000 5000000
[1c] Time elapsed: 3562.5ms
Token=2155621
```

可见，线性算法确实比log算法快很多，但是第三个优化算法，较依赖于k的大小（因为要对大小为k的数组排序），并且使用了C++自带哈希表`unordered_map`时间复杂度的常数较大，因此实际表现并不是很优秀。

- 接下来，尝试n=1e8，k=1e5，本组数据主要区分时间复杂度和n、k相关算法的运行效率差异。

```
→  week4 ./1a
100000000 100000
[1a] Time elapsed: 10546.9ms
Token=116680176
→  week4 ./1b
100000000 100000
[1b] Time elapsed: 0ms
Token=48862785
→  week4 ./1c
100000000 100000
[1c] Time elapsed: 125ms
Token=48862785
```

可见，在n比k大几个数量级时，时间复杂度和k相关的算法会有较大优势。

- 最后，尝试n=2e9，k=1e5。显然前两个算法由于空间复杂度过高，已经无法在本机内存允许范围内运行，因此仅测试第三个。

```
→  week4 ./1c
2000000000 100000
[1c] Time elapsed: 1531.25ms
Token=389807508
```

可见，在n极大，无法将整个数组存储在内存时，第三种做法依然可以在较短的时间内成功运行。

如果想测试更大的n，则需要修改一些循环变量的数据类型至`long long`，否则会因为超出`int`的范围导致溢出。

## b. 实现计算$\pi$的Monte Carlo算法

取n=1e9，运行10次算法，发现极差在1.5e-4左右，比标准略高一点。

取n=5e8时，同样运行10次，得到以下结果：

```
→  week4 g++ 2a.cpp -o 2a -Ofast
→  week4 ./2a
3.1416261520
3.1416288240
3.1415740640
3.1415852240
3.1415096320
3.1416177600
3.1416501760
3.1416432000
3.1416780000
3.1414792320
Min = 3.1414792320
Max = 3.1416780000
Delta = 0.0001987680
```

极差小于`2e-5`，因此答案落在此区间内，相对误差不超过`1e-5`，因此可以认为在`n=5e8`左右时，可以大致满足要求。

**以上均直接粘贴自shell运行结果，对应截图为文件夹内`output-log*.png`。**

# 五、个人总结

实验1通过一个看起来很简单，但是实际上有很大优化空间的算法问题出发，引导我进行逐步优化，同时借助了常见数据结构，实现了时间空间上的平衡，可谓学到许多。最后一种算法来源于数据流算法的思路，可以处理一类更为特殊的数据。

实验2则消耗了我大量时间，每次运行程序都需要运行很久才可以跑出结果，然而结果有时又令人失望，而缩小测试规模又会导致结果不准确，对于这类问题，还应该多积累调试技巧，以后可以尝试二分n的大小进行测试，也许可以提高测试效率。