A Quick Tutorial on Pollard's Rho Algorithm

在研究 pollard rho 算法的时候，无意间看到的这篇文章，觉得写得很好，就将其翻译为中文，可能有些地方翻译地不是很好。原文链接，请点这里

目录

历史

Pollard's Rho 算法是一个非常有趣又容易理解的整数因子分解算法。它并不是目前最快的算法，但它要比试除法快上多个量级。它基于非常简单的思想，而这种思想同样可以用于其他地方。

这个算法最早出自 John M. Pollard 在 1975 年所写的论文

Pollard, J. M. (1975), "A Monte Carlo method for factorization", BIT Numerical Mathematics 15 (3): 331–334

紧接着在 1980 年 Richard Brent 在他的论文中提出了一些改进

Brent, Richard P. (1980), "An Improved Monte Carlo Factorization Algorithm", BIT 20: 176–184, doi:10.1007/BF01933190

更多的内容，可以查看维基百科

Pollard's rho algorithm

问题的设置

我们假设 N 是一个能被分解为 $p * q$ 的数（$N = p * q$ 且 $p \neq q$）
我们的目标是找到 N 的其中一个因子 p 或 q （另外一个因子可以通过除 N 来得到）

我们先来看传统的试除法

```
int main () {
   int N,i;
   // Read in the number N
   scanf("%d", &N);
   printf ("You entered N = %d \n", N);
   if (N %2 == 0) {
      puts ("2 is a factor");
      exit(1);
   }

   for (i = 3; i < N; i+= 2){
       if (N % i == 0) {
          printf ("%d is a factor \n", i);
          exit(1);
       }
   }

   printf("No factor found. \n");
   return 1;
}
```

让我们用一个更暴力的版本：I am feeling lucky algorithm（注意，下面的代码并不是完美的）

```
int main (int argc, char * const argv []) {
   int N,i;
   // Read in the number N
   scanf("%d", &N);
   printf ("You entered N = %d \n", N);

   i = 2 + rand(); // Gets a number from 0 to RAND_MAX

   if (N % i == 0) {
       printf(" I found %d \n", i);
       exit(1);
   }
   printf ("go fishing!\n");
}
```

I am feeling lucky algorithm 将会生成一个 $[2, N]$ 的随机数，
然后检查是否找到了某个因子。那么找到因子的概率是多少呢？
答案非常简单：
在这 $N-1$ 个数中，我们恰好只有两个因子 p 和 q。
因此概率就是 $\frac{2}{N-1}$，如果 $N \sim 10^{10}$，那么成功的机会约为 $\frac{\mathbf{1}}{\mathbf{5000000000}}$，
这比购买一张彩票中奖的概率还要小，
换句话说，我们不得不使用不同的随机数重复将近 N 次来找到一个因子，
显然，这比试除法还要糟糕。

通过 Birthday Trick 提高概率

这是一个简单而又十分有用的提高概率的技巧，它被叫做Birthday Trick。

我们举例来说明这个trick。

从[1, 1000]中随机取一个数，取得 42 的概率为 $\frac{1}{1000}$，

事实上，取得任意一个数的概率均为 $\frac{1}{1000}$。

我们稍微修改一下这个问题：
从[1, 1000]中随机选取两个数 i 和 j，$i - j = 42$ $(i \neq j)$ 的概率是多少？
能够粗略地算出，此时的概率大约为 $\frac{1}{500}$。
如果我们不再坚持仅选取 1 个数并且这个数必须为 42，而是能够选取 2 个数并且他们的差刚好等于 42，那么，成功的概率被提高了。

如果我们在[1,1000]中选取 k 个数 $x_1, \dots \dots, x_k$，

取得的 k 个数中满足 $x_i - x_j = 42$ 的概率是多少呢？这个概率和 k 有什么关系呢？

答案的计算并不是很困难，但是我们还是编写一个简单的程序来实际验证一下

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, int * argv){
  int i,j,k,success;
  int nTrials = 100000, nSucc = 0,l;
  int * p;

  /* Read in the number k */
```

```
  printf ("Enter k:");
  scanf ("%d", &k);
  printf ("\n You entered k = %d \n", k);
  if ( k < 2){
    printf (" select a k >= 2 please. \n");
    return 1;
  }
  /* Allocate memory for our purposes */
  p = (int *) malloc(k * sizeof(int));

  //nTrials = number of times to repeat the experiment.
  for (j = 0; j < nTrials; ++j){

    success = 0;
    // Each experiment will generate k random variables
    // and check if the difference between
    // any two of the generated variables is exactly 42.
    // The loop below folds both in.
    for (i = 0; i < k; ++i){
      // Generate the random numbers between 1 and 1000
      p[i] = 1+ (int) ( 1000.0 *  (double) rand() / (double) RAND_MAX );

      // Check if a difference of 42 has been achieved already
      for (l = 0; l < i; ++l)
    if (p[l] - p[i] == 42 || p[i] - p[l] == 42 ){
      success = 1; // Success: we found a diff of 42
      break;
    }
    }

    if (success == 1){ // We track the number of successes so far.
      nSucc ++;
    }
  }
  // Probability is simply estimated by number of success/ number of
trials.
  printf ("Est. probability is %f \n", (double) nSucc/ (double)
nTrials);
  // Do not forget to cleanup the memory.
  free(p);
  return 1;

}
```

你可以使用不同的 $k(\,k \geq 2\,)$ 值来运行上面的代码

K	Prob. Est.
2	0.0018
3	0.0054
4	0.01123
5	0.018
6	0.0284
10	0.08196
15	0.18205
20	0.30648
25	0.4376
30	0.566
100	0.9999

这张表格展示了概率奇妙的分布情况。
大约在 $k = 30$ 的时候，成功的概率已经超过一半。
我们有一个大区间[1,1000]，但是仅仅生成约 30 个随机数就让我们成功的概率超过了一半，
大约在 $k = 100$ 的时候，我们几乎可以确保我们是成功的。这是一个非常重要的观察，
它被成为 **生日问题** $(birthday\ problem)$ 或者 **生日悖论** $(birthday\ paradox)$

我们随机选择一名学生，他的生日为 4 月 1 日的概率为 $\frac{1}{365}$

这相当于我们在[1,365]中随机选取一个数，该数为 90 的概率是多少？
那么我们又回到了上面那个问题。
我们随机选取 $k(k \geq 2)$个人，他们的生日相同的概率是多少？
我们只需要将代码中的差值 42 进行修改即可

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, int * argv){
  int i,j,k,success;
  int nTrials = 100000, nSucc = 0,l;
  int * p;
  printf ("Enter k:");
  scanf ("%d", &k);
  printf ("\n You entered k = %d \n", k);


  p = (int *) malloc(k * sizeof(int));
  // We will do 1000 reps
  for (j = 0; j < nTrials; ++j){
```

```
    success = 0;
    for (i = 0; i < k; ++i){
      // Generate the random numbers between 1 and 365
      p[i] = 1+ (int) ( 365 *  (double) rand() / (double) RAND_MAX );
      // Check if a difference of 42 has been achieved
      for (l = 0; l < i; ++l)
    if (p[l] - p[i] == 0  ){
      success = 1;
      break;
    }
    }
    if (success == 1){
      nSucc ++;
    }
  }

  printf ("Est. probability is %f \n", (double) nSucc/ (double) nTrials);
  return 1;
}
```

我们可以看到，k = 10 的时候大概有 11%的可能性存在两个人生日相同的情况，
而 k = 23 时，可能性提高至 50%，我们的班级总共有 58 个人，而$k = 58$时的可能性已达到 99%，几乎可以肯定地说，一个班级里必定有两个同学的出生日期是相同的，而这么多年的求学生涯过来了，这个概率“似乎”是不正确的，这便是悖论了。
（这里我们把 $k \geq 366$ 作为数学上精确的 100%）

如果一年中有 N 天(在我们的星球上$N = 365$)，
那么每 $k = \sqrt{N}$ 个人中有 50%的可能性产生生日冲突。

利用生日悖论来因数分解

让我们回到 I am feeling lucky algorithm。我们随即地从$[1, N]$中选择一个数，
这个数是 p 或者 q 的可能性是非常小的，所有我们不得不重复运行算法来提高概率。

那么，我们现在可以提出一个不同的问题：

不再只选取一个整数，我们能够选取 k 个数，并问是否存在$x_i - x_j$能够整除N

我们已经知道对于$k = \sqrt{N}$，我们可以将可能性提高到50%，

所以，可以粗略地说，我们可以将可能性从$\frac{1}{N}$ 提高到 $\sqrt{\frac{1}{N}}$。

因此，对于一个 10 位的整数，我们只需要选取$k = 10^5$个随机数而不是10^{10}个。
不幸的是，这没有节省我们的开销，对于$k(\ k = 10^5)$个人来说，
我们仍然要做$k^2 = 10^{10}$次比较和除法。但幸运的是，这里有一个更好的办法。

我们可以选取 k 个数 $x_1, \dots\dots, x_k$，不再问是否存在$x_i - x_j$能够整除N，

转而询问是否存在 $gcd(x_i - x_y, N) > 1$的情况。

换句话说，我们问 $x_i - x_y$ 和 N 是否存在一个给平凡的最大公约数

如果我们问有多少个数能整除N，答案只有两个，p 和 q
如果我们问有多少个数使得$gcd(x, N) > 1$，答案便很多了
$p, 2p, 3p, 4p, \dots\dots, (q-1)p, q, 2q, 3q, 4q, \dots\dots, (p-1)q$
准确的说，我们有$p + q - 2$ 个数

所以，一个简单的策略如下：

- **在区间$[2, N-1]$中随即选取 k 个数，$x_1, \dots\dots, x_k$**
- **判断是否存在$gcd(x_i - x_y, N) > 1$，若存在，$gcd(x_i - x_y, N)$ 是N的一个因子（p 或 q）**

但是很早就出现了一个问题，我们需要选取大约 $N^{1/4}$ 个数，
这个数量太大了，以至于我们并不能将其存放在内存中

Pollard's Rho 算法

为了解决数太多无法存储的问题，Pollard's rho algorithm 只将两个数存放在内存中。
Pollard's rho algorithm 解决了这个问题。
我们并不随机生成 k 个数并两两进行比较，而是一个一个地生成并检查连续的两个数。
反复执行这个步骤并希望能够得到我们想要的数。

我们使用一个函数来生成伪随机数。
换句话说，我们不断地使用函数 f 来生成（看上去或者说感觉上像的）随机数。
并不是所有的函都能够这样做，但是有一个神奇的函数可以。它就是

$$f(x) = (\ x^2 + a\ ) \bmod N$$

（我们可以自己指定a，也可以用 rand()随即函数来生成，这不是重点）
我们从$x_1 = 2$ 或者其他数开始，让$x_2 = f(x_1), x_3 = f(x_2), \dots\dots$ 生成规则为$x_{n+1} = f(x_n)$

我们来看下面的伪代码：

```
x := 2;

while (.. exit criterion .. )
```

```
   y := f(x);
   p := GCD( | y - x | , N);
   if ( p > 1)
       return "Found factor: p";
   x := y;
end

return "Failed. :-("
```

假设 $N = 55$，$f(x) = (\ x^2 + 2\ )\ mod\ 55$

x_n	x_{n+1}	$\gcd(\vert x_n - x_{n+1}\vert\ , N)$
2	6	1
6	38	1
38	16	1
16	36	5

你可以发现对于大部分的数据这个算法能够正常运行，但是对于某些数据，它将会进入无限循环。为什么呢？这是因为存在 f 环的原因。
当它发生的时候，我们会在一个有限数集中进行无限循环

例如，我们可以构造一个伪随机函数并生成如下伪随机数：

$$2, 10, 16, 23, 29, 13, 16, 23, 29, 13 \dots \dots$$

在这个例子中，我们最终将会在$16, 23, 29, 13$这个圈中无限循环，永远找不到因子

那么，如何探测环的出现呢？
一种方法是记录当前产生过的所有的数$x_1, x_2, \dots \dots\ x_n$,并检测是否存在$x_l = x_n (l < n)$。
在实际过程中，当 n 增长到一定大小时，可能会造成的内存不够用的情况。

另一种方法是由Floyd发明的一个算法，我们举例来说明这个聪明而又有趣的算法。
假设我们在一个很长很长的圆形轨道上行走，我们如何知道我们已经走完了一圈呢？
当然，我们可以像第一种方法那样做，但是更聪明的方法是让 A 和 B 按照 B 的速度是 A 的速度的两倍从同一起点开始往前走，当 B 第一次敢上 A 时(也就是我们常说的套圈)，我们便知道，B 已经走了至少一圈了。

```
a := 2;
b := 2;
while ( b != a )

   a = f(a); // a runs once
   b = f(f(b)); // b runs twice as fast.
   p = GCD( | b - a | , N);
```

```
   if ( p > 1)
       return "Found factor: p";
end

return "Failed. :-("
```

如果算法失败了，我们只需要找到一个新的函数 f 或者重新选择一个随机种子即可

现在我们已经得到了完整的基于 Floyd 的周期检测策略的 Pollard’s rho 算法