Algo_Homework_12

唐浩然

2201111746

素数测试算法：Miller-Rabin算法

前言：在判断一个给定数是否为素数时，直观的方法可以枚举$1-\sqrt{N}$之间所有数字判断是否整除即可，这样带来的时间复杂度为$O(\sqrt{N})$；而在某些实际应用中，我们需要更快速的判断出给定数字是否为素数，在这种前提下，我们可以使用Miller-Rabin算法来**大概率**的判断出给定数字是否为素数，该方法时间复杂度为$O(\log n)$，而代价是不能保证准确判断出是否为素数，而当给定数字小于一定范围时，通过选取合适的底数，可以保证在范围内算法的准确性；

Miller-Rabin算法的实现主要基于下述两个定理：

定理一：费马小定理

若 p 是一个素数，则对于任意的 $0<a<p$，有$a^{p-1}\equiv 1(mod\ p)$；

定理二：二次探测定理

若 p 是一个素数，且$x^2\equiv 1(mod\ p)$，那么$x\equiv 1(mod\ p)$ 和 $x\equiv p-1(mod\ p)$中有一个成立；

因此，算法实现步骤如下：

1. 先用定理1进行判断，即判断 $a^{p-1}\equiv 1(mod\ p)$ 是否成立。如上式不成立，则 p 不是素数，判断完毕。
2. 如步骤一成立，且 p-1 为偶数，则运用定理2，$a^{\frac{p-1}{2}}\equiv 1(mod\ p)$ 和 $a^{\frac{p-1}{2}}\equiv p-1(mod\ p)$中是否有一个成立；如都不成立，则p不是素数，判断完毕；
3. 如步骤二成立，且取模后结果为1，同时 $\frac{p-1}{2}$ 是偶数，则此时依旧满足定理二，继续进行步骤二，直到：

   （1）$\frac{p-1}{2}$ 不是偶数，或者步骤二取模后结果为 p-1，此次判定无法判断 p 是否不为素数，因此尝试下一个底数 a;

   （2）步骤二得到取模后的结果不为 1 或 p-1，则 p 不是素数，判断完毕；

关于底数 a 的选取：（摘自维基百科）

当 N < 4,759,123,141，选取 a = [2,7,61] 即可确保算法得出正确结果;

当 N < 3,825,123,056,546,413,051≈3∗10^18，选取 a = [2,3,5,7,11,13,17,19,23] 即可确保算法得出正确结果;

当 N < 18,446,744,073,709,551,616=2^64 ，选取 a = [2,3,5,7,11,13,17,19,23,29,31,37] 即可确保算法得出正确结果;

代码实现（Python）：

Miller-Rabin:

```
class Solution_MR:
```

```
    def quickpow(self,x, y, p):
        ans = 1
        while y:
            if y & 1:
                ans = ans * x % p
            x = x * x % p
            y >>= 1
        return ans
    def isPrime(self, n: int) -> bool:
        self.prior = [2,3,5,7,11,13,17,19,23,29,31,37]
        if n < 3:
            return n == 2
        for a in self.prior:
            if a >= n:
                break
            if self.quickpow(a,n-1,n) % n != 1:
                return False
            check = n - 1
            while check  % 2 == 0:
                check = check / 2
                mod_res = self.quickpow(a,int(check),n) % n
                if mod_res == n-1:
                    break
                elif mod_res == 1:
                    continue
                else:
                    return False
        return True
```

Brute-Force：（时间复杂度为$O(\sqrt{N})$的枚举方法）

```
class Solution_BF:
    def isPrime(self, n:int) -> bool:
        if n < 3:
            return n == 2
        for i in range(2,int(math.sqrt(n))+1):
            if n % i == 0:
                return False
        return True
```

比较代码：

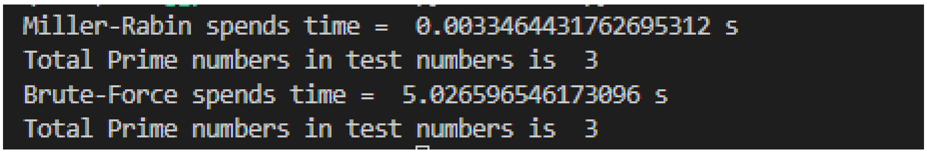
```
miller_rabin = Solution_MR()
brute_force = Solution_BF()
test_num = [randint(3,1e15) for i in range(100)]
ans_mr = []

start_time = time.time()
for test_i in test_num:
     ans_mr.append(miller_rabin.isPrime(test_i))
end_time = time.time()
print('Miller-Rabin spends time = ', end_time - start_time, 's')
print('Total Prime numbers in test numbers is ', sum(ans_mr))

ans_bf = []
start_time = time.time()
```

```python
for test_i in test_num:
     ans_bf.append(brute_force.isPrime(test_i))
end_time = time.time()
print('Brute-Force spends time = ', end_time - start_time, 's')
print('Total Prime numbers in test numbers is ', sum(ans_bf))
```

代码运行结果：



```
Miller-Rabin spends time =  0.0033464431762695312 s
Total Prime numbers in test numbers is  3
Brute-Force spends time =  5.026596546173096 s
Total Prime numbers in test numbers is  3
```

可以看到Miller-Rabin方法所需时间明显由于后者的枚举算法，且二者得到的结果相同；

算法复杂度证明：

在上述算法中可以看到，主要的时间开销在于步骤二中对于二次探测定理的迭代使用（其他时间开销均为常数项），由于每次判断时将当前结果除2带入下一轮，因此易知时间复杂度为$O(\log n)$