Проверка числа на простоту в Python

Информатика. Язык программирования Python в курсе информатики. Все классы.

Проверку числа на простоту оформим в виде функции, которая будет возвращать True для простых чисел и False для составных.

Наивный алгоритм заключается в том, что будем перебирать числа, начиная с 2, пока не не найдем делитель числа n. Если этот делитель будет равен n, то число будет простым, иначе у n есть нетривиальный делитель и число n будет составным.

Запишем алгоритм в виде функции **IsPrime** (по-английски простое число — *prime number*, составное число — *composite number*).

def IsPrime(n):
   d = 2
   while n % d != 0:
       d += 1
   return d == n

В данной записи алгоритма реализована идея линейного поиска с барьерным элементом. Мы хотим найти наименьший делитель числа n. Для этого берем число d и пока n не делится на d переходим к следующему возможному делителю. Алгоритм остановится на числе, которое будет делителем числа n. Если алгоритм остановился на числе n, то число n простое, иначе — составное.

*Сложность* этого алгоритма — O(n).

Однако данный алгоритм можно оптимизировать, если заметить, что у любого составного числа есть собственный (то есть не равный 1) делитель, не превосходящий квадратного корня из числа. Это позволит сократить сложность алгоритма до O(n):

def IsPrime(n):
   d = 2
   while d \* d <= n and n % d != 0:
       d += 1
   return d \* d > n

Соответственно, такой алгоритм заканчивает работу либо при нахождении делителя, либо если проверяемый делитель станет больше корня из n. Чиcло n является простым, если алгоритм закончился по причине того, что проверяемый делитель стал больше, чем корень из n.

ПЕРЕБОР ТОЛЬКО НЕЧЕТНЫХ ДЕЛИТЕЛЕЙ

Сделаем ещё одну оптимизацию — будем перебирать только нечетные делители, если число не делится на два.

def isPrime(n):
    if n % 2 == 0:
        return n == 2
   d = 3
   while d \* d <= n and n % d != 0:
       d += 2
   return d \* d > n

Факторизация перебором делителей

Информатика. Язык программирования Python в курсе информатики. Все классы.

Разложение числа на простые множители называется **факторизацией**.

Факторизация целых чисел обеспечивается **основной теоремой арифметики**:

Каждое натуральное число n>1 можно представить в виде n=p1∗p2∗p3∗…∗pk, где p1,p2,p3,…pk — простые числа, причём такое представление единственно с точностью до порядка следования сомножителей.

Факторизация числа методом перебора делителей производится почти так же, как и тест простоты числа методом перебора делителей. Производится перебор всех целых (как вариант: простых) чисел от 2 до квадратного корня из факторизуемого числа ***n*** и в вычислении остатка от деления ***n*** на каждое из этих чисел. Если остаток от деления на некоторое число ***m*** равен нулю, то ***m***является делителем ***n***. В этом случае ***n*** сокращается на ***m*** и процедура повторяется. По достижении квадратного корня из оставшегося числа и невозможности сократить его ни на одно из меньших чисел, оно объявляется простым и также приписывается к простым сомножителям исходного числа ***n***.

Факторизация целых чисел для больших чисел является сложной задачей. Её сложность лежит в основе некоторых алгоритмов безопасности с открытым ключом шифрования, например, алгоритма RSA.

ФАКТОРИЗАЦИЯ ПЕРЕБОРОМ ДЕЛИТЕЛЕЙ НА PYTHON

def factorize(n):
    divisor = 2
    while divisor \*\* 2 <= n:
        if n % divisor == 0:
            n //= divisor
            print(divisor)
        else:
            divisor += 1
    if n != 1:
        print(n)

n = int(input())
factorize(n)

ФАКТОРИЗАЦИЯ ПЕРЕБОРОМ ДЕЛИТЕЛЕЙ НА PASCAL

procedure factorize(n: longint);
var
 divisor: longint;
begin
 divisor := 2;
 while divisor \* divisor <= n do
    if n mod divisor = 0 then begin
     n := n div divisor;
     writeln(divisor);
   end
   else
     inc(divisor);
  if n <> 1 then
   writeln(n);
end;
var
 n: longint;
begin
 readln(n);
 factorize(n);
end.

1. Разложение числа на множители в Python
2. Информатика. Язык программирования Python в курсе информатики. Все классы.
3. Основная теорема арифметики гласит, что разложение числа на простые множители существует и единственно. Решение этой задачи оформим в виде функции, которая по заданному числу n возвращает список его простых делителей с учетом кратности.

Алгоритм разложения числа на множители похож на алгоритм проверки числа на простоту. Будем проверять делимость данного числа n на натуральные числа подряд, начиная с числа 2. Если мы находим делитель числа n, то будем делить число n на данный делитель, пока число делится на него и добавлять делитель в список простых делителей.

Перебор также необходимо продолжать до n. Если после окончания этого алгоритма число n не станет равно 1, то оставшееся значение также является простым, так как не делится ни на одно число, не превосходящее корня из оставшегося значения, поэтому его нужно добавить к списку простых делителей:

def Factor(n):
   Ans = []
   d = 2
   while d \* d <= n:
       if n % d == 0:
           Ans.append(d)
           n //= d
       else:
           d += 1
   if n > 1:
       Ans.append(n)
   return Ans

Сложность такого алгоритма, как и сложность алгоритма проверки числа на простоту, O(n)



Генерация псевдослучайных чисел

Информатика. Язык программирования Python в курсе информатики. Все классы.

*«Генерация случайных чисел слишком важна, чтобы оставлять её на волю случая»*
*Роберт Р. Кавью*

ОПРЕДЕЛЕНИЕ

**Генерация псевдослучайных чисел** — порождение последовательности чисел, элементы которой подчиняются заданному распределению (обычно равномерному).

Псевдослучайные числа используются в методе Монте-Карло, в криптографии, для моделирования физической и игровой реальности, для придания действиям игрового искусственного интеллекта элемента спонтанности.

Соответствующие алгоритмы называют генераторами псевдослучайных чисел (ГПСЧ).

ДЕТЕРМИНИРОВАННЫЕ ГЕНЕРАТОРЫ

Детерминированный алгоритм не может генерировать полностью случайные числа, он может только аппроксимировать некоторые их свойства.

Любой ГПСЧ с ограниченными ресурсами рано или поздно зацикливается — начинает повторять одну и ту же последовательность чисел. Если порождаемая ГПСЧ последовательность сходится к слишком коротким циклам, то такой ГПСЧ становится предсказуемым и непригодным для практических приложений.

Большинство простых арифметических генераторов имеют недостатки:

* Слишком короткий период.
* Последовательные значения не являются независимыми.
* Некоторые биты «менее случайны», чем другие.
* Неравномерное одномерное распределение.
* Обратимость.

Линейный конгруэнтный метод

Члены последовательности вычисляются рекуррентно по формуле:

mod Xk+1=(aXk+c)mod m

где a и c — некоторые целочисленные коэффициенты, а mod mod m — взятие остатка по модулю некоторого натурального числа m.

1. НОДНОД(c,m)=1 (то есть, c и m взаимно просты);
2. a−1 кратно p для всех простых делителей p числа m;
3. a−1 кратно 4, если m кратно 4.

длина периода меньше либо равна m.

ЧАСТО ИСПОЛЬЗУЕМЫЕ ПАРАМЕТРЫ

1. m=232,a=69069,c=5
2. m=232,a=22695477,c=1
3. m=232,a=1664525,c=1013904223
4. m=264,a=6364136223846793005,c=1442695040888963407

Линейный конгруэнтный метод дает статистически приемлемое распределение псевдослучайных чисел, но не является криптостойким.

Реализация линейного конгруэнтного метода на Си

unsigned long g\_random\_seed = 1; //глобальная переменная с последним числом

unsigned long my\_rand()
{
    //размер unsigned long равен 4 байта!
    //m = 2^32
    return g\_random\_seed = 69069LU\*g\_random\_seed + 5LU;
}

Реализация линейного конгруэнтного метода на Python

g\_random\_seed = 1
def rr():
    global g\_random\_seed
    g\_random\_seed = (69069\*g\_random\_seed+5)%(2\*\*32)

ВОПРОС

Почему в алгоритме на Си отсутствует деление по модулю m, но код работает, а в Python наличие %(2\*\*32)обязательно?