

Example Numerical Calculations in IronPython: Prime Numbers

Introduction	1
Identifying Prime numbers	1
Prime Factorization Algorithm	2
Description of Algorithm.....	2
Manual Prime Factorization Example	2
First Algorithm Implementation	3
Second Algorithm Implementation	4
Third Algorithm Implementation	4
Element 1: Set Up the Environment.	4
Element 2: Obtain the Integer Square Root of a Long Number	5
Element 3: Test Whether Number is Prime	5
Element 4: Factorize An Integer Number	6
Element 5: Condense the Results Into a Readable Format	6
Element 6: Compare Observed and Expected Results.....	6
Time complexity of Algorithm.....	6

Introduction

The syntax of the **Python programming language** is the set of rules that defines how a Python program will be written and interpreted (by both the runtime system and by human readers). Python was designed to be a highly readable language. It aims toward an uncluttered visual layout, uses English keywords frequently where other languages use punctuation, and has notably fewer syntactic constructions than many structured languages such as C, Perl, or Pascal. A program that does not conform to these rules will not run, or it will produce unexpected results.

Identifying Prime numbers

In mathematics, a **prime number** (or a **prime**) is a natural number that has exactly two (distinct) natural number divisors, which are 1 and the prime number itself. There exists an infinitude of prime numbers, as demonstrated by Euclid in about 300 B.C.. The first 30 prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109 and 113.

The Python code below tests all the numbers in the range of 2 to 114 to determine whether they are prime numbers and returns a list of the ones that are prime:

```
def main():
    import sys
    # range to evaluate
    min, max = 2, 114
    if sys.argv[1:]:
        min = int(eval(sys.argv[1]))
        if sys.argv[2:]:
            max = int(eval(sys.argv[2]))
    primes(min, max)
```

```
def primes(min, max):
    if 2 >= min: print 2
    primes = [2]
    i = 3
    while i <= max:
        for p in primes:
            if i%p == 0 or p*p > i: break
        if i%p <> 0:
            primes.append(i)
            if i >= min: print i
        i = i+2

print main()
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113]

Prime Factorization Algorithm

A **prime factorization algorithm** is any algorithm by which an integer (whole number) is "decomposed" into a product of factors that are prime numbers (see prime factor). The fundamental theorem of arithmetic guarantees that this decomposition is unique. The sections below provides

- a manual example of the prime factorization algorithm;
- a first implementation of the algorithm, which works well for numbers whose prime factors are small;
- a second implementation, which is easy to understand; and
- a third, faster implementation for numbers with larger prime factors are discussed in the article on integer factorization.

A 'fast' algorithm (which can factorize large numbers in a reasonably small time) is much sought after.

Description of Algorithm

We can describe a recursive algorithm to perform such factorizations: given a number n

- if n is prime, this is the factorization, so stop here.
- if n is composite, divide n by the first prime p_1 . If it divides cleanly, recurse with the value n/p_1 . Add p_1 to the list of factors obtained for n/p_1 to get a factorization for n . If it does not divide cleanly, divide n by the next prime p_2 , and so on.

Note that we need to test only primes p_i such that $p_i \leq \sqrt{n}$.

Manual Prime Factorization Example

Suppose we wish to factorize the number 9438.

- $9438/2 = 4719$ with a remainder of 0, so **2 is a factor** of 9438. *We repeat the algorithm with 4719.*
- $4719/2 = 2359$ with a remainder of 1, so 2 is NOT a factor of 4719. *We try the next prime, 3.*

- $4719/3 = 1573$ with a remainder of 0, so **3 is a factor** of 4719. *We repeat the algorithm with 1573.*
- $1573/3 = 524$ with a remainder of 1, so 3 is NOT a factor of 1573. *We try the next prime, 5.*
- $1573/5 = 314$ with a remainder of 3, so 5 is NOT a factor of 1573. *We try the next prime, 7.*
- $1573/7 = 224$ with a remainder of 5, so 7 is NOT a factor of 1573. *We try the next prime, 11.*
- $1573/11 = 143$ with a remainder of 0, so **11 is a factor** of 1573. *We repeat the algorithm with 143.*
- $143/11 = 13$ with a remainder of 0, so **11 is a factor** of 143. *We repeat the algorithm with 13.*
- $13/11 = 1$ with a remainder of 2, so 11 is NOT a factor of 13. *We try the next prime, 13.*
- $13/13 = 1$ with a remainder of 0, so **13 is a factor** of 13. *We stop when we reached 1.*
- Thus working from top to bottom, we have $9438 = 2 \times 3 \times 11 \times 11 \times 13$.

First Algorithm Implementation

Here is some code in Python for finding the factors of numbers less than 2147483647. We will use it to factorize the number 9438:

```
from math import sqrt

# number to factorize
n = 9438

def factorize(n):
    def isPrime(n):
        return not [x for x in xrange(2,int(sqrt(n))+1) if n%x == 0]
    primes = []
    candidates = xrange(2,n+1)
    candidate = 2
    while not primes and candidate in candidates:
        if n%candidate == 0 and isPrime(candidate):
            primes = primes + [candidate] + factorize(n/candidate)
            candidate += 1
    return primes
```

Observed Results:	<code>print factorize(n)</code> <code>[2, 3, 11, 11, 13]</code>
Expected Results:	<code>[2, 3, 11, 11, 13]</code>

Second Algorithm Implementation

This python algorithm is not efficient, but easy to understand. If there are large factors, it will take forever to find them, because we try all odd numbers between 3 and \sqrt{n} ... Let's use it to factorize, again, the number 9438:

```
from math import sqrt

# number to factorize
n = 9438

error = 'fact.error'          # exception

def fact(n):
    if n < 1: raise error      # fact() argument should be >= 1
    if n == 1: return []      # special case
    res = []
    # Treat even factors special, so we can use i = i+2 later
    while n%2 == 0:
        res.append(2)
        n = n/2
    # Try odd numbers up to sqrt(n)
    limit = sqrt(float(n+1))
    i = 3
    while i <= limit:
        if n%i == 0:
            res.append(i)
            n = n/i
            limit = sqrt(n+1)
        else:
            i = i+2
    if n != 1:
        res.append(n)
    return res
```

Observed Results:	<code>print fact(n)</code> <code>[2, 3, 11, 11, 13]</code>
Expected Results:	<code>[2, 3, 11, 11, 13]</code>

Third Algorithm Implementation

Here is more complex code in Python for finding the factors of any arbitrarily large number. We will walk through the elements of the code below:

Element 1: Set Up the Environment.

We define some date and set up some housekeeping functions:

```
# number to factorize
n = 173248246132375748867198458668657948626531982421875

ListOfPrimes=[2,3,5,7,11,13,17,19]
maxindex=len(ListOfPrimes)
maxprimeinlist=ListOfPrimes[-1]

# Put Primes in a dictionary
DictPrime={}
DictPrime.fromkeys(ListOfPrimes,True)
```

Element 2: Obtain the Integer Square Root of a Long Number

Define the function `intsqrt(n)`, which corresponds to the integer square root of a long number

```
def intsqrt(n):
    """ Return the integer square root of a long number """
    def intsqrt_core(digitpair,remainder,results):
        # function intsqrt_core returns (results,remainder)
        if digitpair<100:
            currvalue=remainder*100 + digitpair
            for d in range(9,-1,-1):
                x=(2*10*results + d)*d
                if x <= currvalue:
                    remainder= currvalue - x
                    results=results*10 + d
                    return(results,remainder)
            else:
                (results,remainder)=intsqrt_core(digitpair//100,remainder,results)
                (results,remainder)=intsqrt_core(digitpair%100,remainder,results)
                return(results,remainder)
        (results,remainder)=intsqrt_core(n,0,0)
    return results
```

Element 3: Test Whether Number is Prime

Define the function `isPrime(n)`, which returns True if n is prime:

```
def isPrime(n):
    """ Return True if n is a prime """
    if DictPrime.has_key(n):
        return True
    high=intsqrt(n)
    for x in ListOfPrimes:
        if x <= high and n%x == 0:
            return False
        if x >= high:
            return True
    x=maxprimeinlist + 2
    while x<=high:
        if n%x == 0:
            return False
        x += 2
    return True
```

Element 4: Factorize An Integer Number

Define the function `factorize(n)`, which factorizes an integer number:

```
def factorize(n):
    """ Factorize an integer number """
    primes = []
    index=0
    candidate = ListOfPrimes[index]
    while not primes and candidate <= n:
        if n%candidate == 0 and (index < maxindex or isPrime(candidate)):
            primes = primes + [candidate] + factorize(n//candidate)
            index += 1
        if index < maxindex:
            candidate = ListOfPrimes[index]
        else:
            candidate += 2
    return primes
```

Element 5: Condense the Results Into a Readable Format

Define the function `condense(L)`, which condenses the results in a list to prime^{nth} power format:

```
def condense(L):
    """ Condense result in list to prime^nth_power format """
    prime,count,list=0,0,[]
    for x in L:
        if x == prime:
            count += 1
        else:
            if prime != 0:
                list = list + [str(prime) + '^' + str(count)]
                prime,count=x,1
    list = list + [str(prime) + '^' + str(count)]
    return list
```

Element 6: Compare Observed and Expected Results

Observed results:	<code>print condense(factorize(long(n)))</code> <code>['3^24', '5^14', '7^33', '13^1']</code>
Expected results:	<code>['3^24', '5^14', '7^33', '13^1']</code>

Time complexity of Algorithm

The algorithm described above works fine for small n , but becomes impractical as n gets larger. For example, for an 18-digit (or 60 bit) number, all primes below about 1,000,000,000 may need to be tested, which is taxing even for a computer. Adding two decimal digits to the original number will multiply the computation time by 10. The difficulty (large time complexity) of factorization makes it a suitable basis for modern cryptography.