BRIEF ARTICLE

THE AUTHOR

1. Problems

1.1. Problem 50. Consecutive prime sum

Python, about 35 ms. I start with sums of primes from 2, and find the longest sequence of primes that sum to a prime, then do the same from 3, and stop once it is not possible to find a sequence longer than the longest already found, which starts from 7.

I use a 3.5x faster version of David Epstein's very nice primes generator, and a simple 6n1 test for primality that I have used on other problems.

```
import time
import numpy as np
import itertools as it

def cp(n):
    t = time.clock()
    nmoretries=1000
    ntries=0
    for startFrom in list(primesieve(100)):
        ntries+=1
        if ntries>nmoretries:
            break
        psums={}
        psums[startFrom],count,countmax=0,0,-1
        for p in erat2a():
            if p<startFrom:
                continue
            count+=1
            psums[startFrom]+=p
            if psums[startFrom]>n:
                break
            if isPrime(psums[startFrom]):
                if count>countmax:
                    pmax=p
                    nmax=psums[startFrom]
                    countmax=count
        print (startFrom,pmax,nmax,countmax)
```

```
        # is it worth continuing?
        if primesthatsumto(n)-countmax<nmoretries:
            nmoretries=primesthatsumto(n)-countmax
            ntries=0
    print(time.clock()-t)

def primesthatsumto(n):
    psum=0
    count=0
    for p in erat2a():
        psum+=p
        if psum>n:
            break
        count+=1
    return count
#
# prime generator
#http://code.activestate.com/recipes/117119/
def erat2a():
    D = {}
    yield 2
    for q in it.islice(it.count(3), 0, None, 2):
        p = D.pop(q, None)
        if p is None:
            D[q * q] = 2 * q # use here 2 * q
            yield q
        else:
            x = p + q
            while x in D:
                x += p
            D[x] = p

def primesieve(n):
    """return array of primes 2<=p<=n"""
    sieve=np.ones(n+1,dtype=bool)
    for i in range(2, int((n+1)**0.5+1)):
        if sieve[i]:
            sieve[2*i::i]=False
    return np.nonzero(sieve)[0][2:]

def isPrime(n):
    """Returns True if n is prime."""
```

```
    if n==2 or n==3:
        return True
    if not n%2 or not n%3:
        return False
    i = 5
    w = 2
    while i * i <= n:
        if n % i == 0:
            return False
        i += w
        w = 6 - w
    return True
```

1.2. **Problem 60. Prime pair sets**

Bah! I cannot get this to go any quicker than about 3.6 s. At least that is way faster than the 120 s+ I was getting when I first solved this.

I first generate a dictionary of primes p_i up to some limit, then to each of these I attach a set of the other primes in that dict, p_j, for which the concatenations p_ip_j and p_jp_i are prime. I then use set intersections to find the primes that connect to however many others is required.

Creating the dictionary takes 90% of the total time.

Creating primes is far faster than checking for primality - so if checking is what you want to do, then it can be faster to put lots of primes in a data structure with constant look-up time, such as a Python set or dictionary, then check whether your candidate prime is in that structure.

This problem is clearly amenable to graph-theoretical analysis, and what I have done resembles looking for cliques in a graph. I look forward to pursuing this more explicitly.

```
def primesfrom2to(n):
    """ Input n>=6, Returns a array of primes, 2 <= p < n """
    #Code by Robert William Hanks
    #http://stackoverflow.com/questions/2068372/fastest-way-to-list-all-primes-below-n/3035188
    sieve = numpy.ones(n//3 + (n%6==2), dtype=numpy.bool)
    for i in range(1,int(n**0.5/3)+1):
        if sieve[i]:
            k=3*i+1|1
            sieve[       k*k//3   ::2*k] = False
            sieve[k*(k-2*(i&1)+4)//3::2*k] = False
    return numpy.r_[2,3,((3*numpy.nonzero(sieve)[0][1:]+1)|1)]

def goodprimes(n,m):
    primes=list(primesfrom2to(n))
    pc=set(primesfrom2to(n**2))
    [x.remove(y) for x in [primes,pc] for y in [2,5]]
```

```
    pdic={prime:set([prime]) for prime in primes}
    for prime1 in primes:
        for prime2 in primes:
            try1=prime1*10**(int(math.log10(prime2))+1)+prime2
            if try1 not in pc:
                continue
            try2=prime2*10**(int(math.log10(prime1))+1)+prime1
            if try2 in pc:
                pdic[prime1].add(prime2)
    opdic={}
    for k,v in pdic.items():
        if len (v)>=m:
            opdic[k]=v
    return opdic,pc

def PE_0060(n,m):

    start1=timer()
    pdic,pc=goodprimes(n,m)
    print ('Elapsed time:',timer()-start1)

    start2=timer()
    smin=math.inf
    tts={}
    for k,v in pdic.items():
        for x in v:
            if x ==k: continue
            try:
                tt=v.intersection(pdic[x])
                tts.setdefault(len(tt),[]).append(tt)
            except:
                pass
    print ('Elapsed time:',timer()-start2)

    start3=timer()
    for tt in tts[m]:
       if sum([x*10**(int(math.log10(y))+1)+y in pc for x in tt for y in tt if x!=y ])==
            if sum(tt)<smin:
                smin=sum(tt)
                ttmin=tt
    print(ttmin,smin)
    print ('Elapsed time:',timer()-start3)
```

```
    print ('Total elapsed time:',timer()-start1)
```

1.3. **Problem 64. Odd period square roots**

About 240 ms in Python

```
import time
import math
def p64(n):
    """returns number of  integers <=n that have odd-period continued fractions"""
    t=time.clock()
    c=0
    for i in range (1,n+1):
        if  i%4==0 or int(math.sqrt(i))==math.sqrt(i):
            continue
        if len(sqcf(i)[1])%2==1:
            c+=1
    print (c,time.clock()-t)

def sqcf(S):
    """
    S is a natural number. Must not be a perfect square

    returns (a0,[r0,..,rn]) where a0 is the stem and [r0,...,rn] is the
    repeating part of the square root continued fraction of S
    """
    a=[int(math.sqrt(S))]
    d0,d=1,1
    m=0
    while 1:
        m=d*a[-1]-m
        d=int((S-m**2)/d)
        a.append(int((a[0]+m)/d))
        if d==d0:
            return (a[0],a[1:])
            break
```

1.4. **Problem 74. Digit factorial chains**

About 50 ms*, in Python.

I only check for numbers with integers that increase in order, then find the number of valid permutations of those digits - all permutations must have the same chain length. To speed things up when working through chains, I develop a dictionary of the chain lengths found for all numbers that have appeared in chains before, and stop when I get to one of these numbers, because the chain length of the present number can now be calculated directly. This saves a lot of time.

When I first solved the problem using brute force, my code required 70 s. Getting the dictionary/cache method to work correctly (which took me ages!) brought that down by a factor of 20 to 3.5 s, then the realisation that digit order did not matter brought the time down by a further factor of 70, to 50 ms.

```
def p74(n):
    start=timer()
    fs=[1,1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
    chainlengths={169:3,871:2,872:2,145:1,69:5,78:4,540:2}
    fd=set()
    for number in itertools.combinations_with_replacement('0123456789',len(str(n-1))):
        nx=int(''.join([x for x in number]))
        for number in [nx,10*nx]:
            if number>n:
                continue
            chain=[number]
            while 1:
                candidate=sum([fs[int(x)] for x in str(chain[-1])])
                if candidate in set(chain):
                    chainlengths[candidate]=len(chain)-chain.index(candidate)
                    break
                if candidate in chainlengths:
                    chainlengths[number]=len(chain)+chainlengths[candidate]
                    break
                chain.append(candidate)

            for j in range(len(chain)):
                if chain[j] in chainlengths:
                    continue
                if candidate in set(chain):
                    chainlengths[chain[j]]=chainlengths[candidate]+chain.index(candidate
                else:
                    chainlengths[chain[j]]=chainlengths[candidate]+len(chain)-j

            if chainlengths[number]==60:
                fd.add(number)

    #how many permutations are there of each of these numbers?
    ysum=[]
    for x in fd:
        y=[i for i in str(x)]
        ysum.append(math.factorial(len(y)))
        if '0' in y:
```

```
            ysum[-1]-=math.factorial(len(y)-1)
        y=''.join([i for i in y])
        xdic={}
        for digit in y:
            xdic[digit]=xdic.get(digit,0)+1
        for k,v in xdic.items()]
            ysum[-1]=ysum[-1]//math.factorial(v)

    print(sum(ysum))
    print('Elapsed time',timer()-start)
```

1.5. **Problem 75. Singular integer right triangles**

Feels slow - about 4 s in Python.

I use the three generating matrices to generate a ternary tree of primitive triads, then add in the non-primitives. Nothing more clever going on.

```
import numpy as np
import copy

def perimeters(n,m):

   print('Up to a maximum perimeter of',n,'there are:')
   L=primitives(n)
   allL(n,L,m)

def primitives(n):
    """
    returns dictionary L {k:v} where k are the perimeters of primitive
    Pythagorean triangles less than n, and v are the number primitives that share
    that perimeter
    """

    #generating matrices
    A = np.array( [[1,-2,2], [2,-1,2],[2,-2,3]] )
    B = np.array( [[1,2,2], [2,1,2],[2,2,3]] )
    C = np.array( [[-1,2,2], [-2,1,2],[-2,2,3]] )

    L={12:1}
    tripgen=[[3,4,5]]  # the root of the tree

    while True:
        nextgen=[]
        for triplet in tripgen:
```

```
            for matrix in [A,B,C]:
                c=np.dot(matrix,np.array(triplet))
                if c[2]<=n/2:
                    length=sum(c)
                    nextgen.append(list(c))
                    L[length]=L.get(length,0)+1
        if len(nextgen)==0:
            break
        tripgen=copy.deepcopy(nextgen)
    p=(sum([y for x,y in L.items() if x<=n and y>=1]))
    print(p,'primitive Pythagorean trangles')
    return L


def allL(n,L,m):
    """
    returns the number of perimeters less than n shared by m Pythagorean triangles
    L is a dict of primitive perimeters returned by primitives()
    """
    AllPT={}
    for primitive,v in L.items():
        length=0
        i=0
        while True:
            i+=1
            length=i*primitive
            if length>n:
                break
            AllPT[length]=AllPT.get(length,0)+1
    perims= (len({x:y for x,y in AllPT.items() if x<=n and y==m}))
    print(perims,'perimeters common to',m,'Pythagorean triangles')
```

1.6. **Problem 94. Almost equilateral triangles**

More Pythagorean triples, where this time we need only consider primitive triples. About 1.3 ms in Python.

```
import numpy as np

def aet(pmax):
    """
    returns L, a dictionary of all the right-angle triangles a<b<c that, when
    mirrored on b, give an almost-equilateral triangle (c,c,2a) where 2a=c+/-1,
    with perimeter less than or equal to pmax. These almost-equilateral triangles
```

```
necessarily comprise the set of those that have integer area.
"""
#generating matrices
A = np.array( [[1,-2,2], [2,-1,2],[2,-2,3]] )
B = np.array( [[1,2,2], [2,1,2],[2,2,3]] )
C = np.array( [[-1,2,2], [-2,1,2],[-2,2,3]] )

tripgen=[[3,4,5]]
L={5:[3,4,5]}

while True:
    nextgen=[]
    for triplet in tripgen:
        for matrix in [A,B,C]:
            c=sorted(list(np.dot(matrix,np.array(triplet))))
            if 2*(c[0]+c[2])<=pmax:
                if c[0]==(c[2]+1)/2 or c[0]==(c[2]-1)/2:
                    nextgen.append(c)
                    L[c[2]]=c

    if len(nextgen)==0:
        break
    tripgen=nextgen[:]

print(sum([2*(v[0]+v[2]) for k,v in L.items()]))
return L
```

1.7. **Problem 114.** $50-100\mu$s in Python, using recursion and a memo. For block lengths up to 3 units I worked out the number of solutions that had a red left-most block, or a black left-most block, and put these pairs of values in a dictionary with the block length as key. Then, working upwards in block lengths one unit at a time, I imagined placing $k=1$ to n black blocks at the left-most end. For each value k, this gave a rightward space of $n-k$ blocks in length that had to be filled with any allowed combination of sub-blocks that had a red left-most edge. I could look this value up in the dictionary. The sum of these values for each k then gave the number of solutions for blocks of length n with a black left edge. To find the solutions for n units that have a red edge, I do the same, starting this time with a red block of 3 units at the left edge.

In fact, on inspecting the numbers for the first few block lengths, one sees that the recursion relation is:

$$n_{\text{black left edge}}(L) = n_{\text{black left edge}}(L-1) + n_{\text{red left edge}}(L-1)$$
$$n_{\text{red left edge}}(L) = n_{\text{black left edge}}(L-3) + n_{\text{red left edge}}(L-1)$$

Here are the values up to $L = 20$:

L	R	B	$R+B$
1	0	1	1
2	0	1	1
3	1	1	2
4	2	2	4
5	3	4	7
6	4	7	11
7	6	11	17
8	10	17	27
9	17	27	44
10	28	44	72
11	45	72	117
12	72	117	189
13	116	189	305
14	188	305	493
15	305	493	798
16	494	798	1292
17	799	1292	2091
18	1292	2091	3383
19	2090	3383	5473
20	3382	5473	8855

```
import time
def F(m,n,memo={}):
    t=time.clock()
    blocks={k:[0,1] for k in range (1,m)}
    blocks[0]=[0,0]
    blocks[m]=[1,1]
    try:
        print(m,n,memo[(m,n)],time.clock()-t)
        return
    except KeyError:
        for L in range (m+1,n+1):
            blocks.setdefault(L,[]).append(blocks[L-1][0]+blocks[L-m][1]) #red left-edge
            blocks.setdefault(L,[]).append(blocks[L-1][0]+blocks[L-1][1]) #black left-ed
        result =sum(blocks[L])
        memo[(m,n)]=result
    print(m,n,sum(blocks[L]),time.clock()-t)
```

1.8. **Problem 213: Flea Circus**

About 550ms in Python, without taking any symmetries into account..

I use a Markov chain. First I construct the transition matrix $\mathbf{P}$, then for each flea I calculate the probability that it would not occupy each of the cells after n bells. This is given by $1-\mathbf{P}^n\mathbf{t}$ where $\mathbf{t}$ is a column vector of zeros, except for $t_i = 1$ when we consider the ith flea. Starting with a value of 1 for the probability of any cell being empty, I multiply that at each stage by the probability for each flea not ending up there. After all fleas are accounted for, we have the probability that each cell is unoccupied after n steps. The sum of these probabilities is the expected number of empty cells.

```
import time
import numpy as np

def p213(N,steps):

    t0=time.clock()

    nsq=N**2
    P=tpm(N)
    Psteps=np.linalg.matrix_power(P,steps)
    tcurrent=np.zeros([nsq,1])
    tsum=np.ones([nsq,1])

    for i in range(nsq):
        t=np.zeros([nsq,1])
        t[i][0]=1
        tcurrent=np.dot(Psteps,t)
        tsum*=(1-tcurrent)

    print(round(sum(tsum)[0],6))
    print(time.clock()-t0)

#construct transition matrix
def tpm(N):

    P=np.zeros([N**2,N**2])

    #corners (0,N-1,(N-1)*N+1,N**2):
    P[0][1]=0.5
    P[0][N]=0.5
    P[N-1][N-2]=0.5
    P[N-1][2*N-1]=0.5
    P[(N-1)*N][(N-2)*N]=0.5
    P[(N-1)*N][(N-1)*N+1]=0.5
    P[N**2-1][N**2-2]=0.5
```

```
    P[N**2-1][N**2-N-1]=0.5

    #top row
    for i in range(1,N-1):
        P[i][i-1]=1/3
        P[i][i+1]=1/3
        P[i][i+N]=1/3

    #bottom row
    for i in range(N**2-N+1,N**2-1):
        P[i][i-1]=1/3
        P[i][i+1]=1/3
        P[i][i-N]=1/3

    #left edge
    for i in range(N,(N-1)*N,N):
        P[i][i-N]=1/3
        P[i][i+N]=1/3
        P[i][i+1]=1/3

    #right edge
    for i in range(2*N-1,(N-1)*N,N):
        P[i][i-N]=1/3
        P[i][i+N]=1/3
        P[i][i-1]=1/3

    #interior
    for i in range(N+1,(N-1)*N,N):
        for j in range(N-2):
            P[i+j][i+j-1]=0.25
            P[i+j][i+j+1]=0.25
            P[i+j][i+j-N]=0.25
            P[i+j][i+j+N]=0.25

    return P.transpose()
```

1.9. **Problem 227.** About 2.7 ms in Python.

```
import time
import numpy as np
from numpy import linalg as LA

def p227(players):
```

```
t0=time.clock()

#if there are 2N players, distances between the dice-holding
# players will vary from 0 to N.
# We need a N+1 x N+1 transition matrix


N=(players+2)//2

#construct the transition matrix P
P=np.zeros([N,N])
P[N-1][N-1]=36

P[0][0]=18
P[0][1]=16
P[0][2]=2

P[1][0]=8
P[1][1]=19
P[1][2]=8
P[1][3]=1

P[N-2][N-4]=1
P[N-2][N-3]=8
P[N-2][N-2]=19
P[N-2][N-1]=8

for i in range(2,N-2):
    P[i][i-2]=1
    P[i][i-1]=8
    P[i][i]=18
    P[i][i+1]=8
    P[i][i+2]=1

#normalise P
P/=36

#Q is the matrix of tranisitions between non-absorbing states
qslice=[True]*(N-1)
Q=np.compress(qslice,P,0)
Q=np.compress(qslice,Q,1)

#I-Q
```

```
IMQ=np.identity(N-1)-Q
#N=inv(I-Q) is the fundamental matrix for P
#entry n_ij of N is expected number of times that chain will be in state s_j
#if it starts in state s_i
Nm=LA.inv(IMQ)

c=np.ones([N-1,1])
#Let ti be the expected number of steps before the chain is absorbed, given
#that the chain starts in state s_i, and let t be the column vector whose
#ith entry is t_i. Then
t=np.dot(Nm,c)

#We start in state s_0 (maximum distance apart)

print(round(t[0][0],6))
print(time.clock()-t0)
```

This was clearly solvable as an absorbing Markov chain problem, so I spent a happy day or two reading more on them. [url=https://math.dartmouth.edu/ prob/prob/prob.pdf]Grinstead and Snell[/url] Chapter 11 was particularly useful.

First, for $2N$ players I used pen and paper to construct the $N+1$ square transition matrix $\mathbf{P}$, with the element p_{ij} being the probability for transition from a state where the dice are a distance $N-i$ apart to one where they are a distance $N-j$ apart. That is, a transition between states a distant N apart is in the top left and the element for the transition that begins and ends on the absorbing state of zero distance apart is in the bottom right. The top-left N by N sub-matrix of this is $\mathbf{Q}$. It's elements give the probabilities of transitions between transitory states.

Then we construct $\mathbf{N} = (\mathbf{I} - \mathbf{Q})^{-1}$. Grinstead and Snell call $\mathbf{N}$ the [i]fundamental[/i] matrix for $\mathbf{P}$. An entry n_{ij} of $\mathbf{N}$ is the expected number of times that the chain will be in state s_j if it starts in state s_i. Finally, we let t_i be the expected number of steps before the chain is absorbed, given that the chain starts in state s_i, and let $\mathbf{t}$ be the column vector whose [i]i[/i]th entry is t_i. Then

$$\mathbf{t} = \mathbf{Nc}$$

where $\mathbf{c}$ is a column vector all of whose entries are 1. The answer to our problem is given by t_0.

1.10. **Problem 243.** About 0.3 ms in Python. I use the same ideas as many here, noting that $R(d) = \frac{\phi(n)}{n-1} \approx \frac{\phi(n)}{n}$ for large n, and so is a function only of the prime factors of n, independent of the exponents of those factors. The denominator will be minimised if n is a primorial. Thus we look for the smallest primorial number for which $\frac{\phi(n)}{n} < \frac{15499}{94744}$and then multiply that successively by 2 until $\frac{\phi(n)}{n-1} < \frac{15499}{94744}$, knowing that by doing so we will not change the value of $\frac{\phi(n)}{n}$.

```
import time
```

```
import numpy as np

def p243():
    t=time.clock()
    primes=primesieve(100)
    Rtrial=1
    i=0
    while et(Rtrial)/Rtrial>15499/94744:
        Rtrial*=primes[i]
        i+=1
    while et(Rtrial)/(Rtrial-1)>15499/94744:
        Rtrial*=2
    print(Rtrial)
    print(time.clock()-t)

def primesieve(n):
    """return array of primes 2<=p<=n"""
    sieve=np.ones(n+1,dtype=bool)
    for i in range(2, int((n+1)**0.5+1)):
        if sieve[i]:
            sieve[2*i::i]=False
    return np.nonzero(sieve)[0][2:]

#Euler totient is number of integers m 1 <= m <=n that are coprime with n
def et(n):
    """returns Euler totient (phi) of n """
    phi=n
    pfs=set(prime_factors(n))
    for pf in pfs:
        phi*=(1-1/pf)
    return int(phi)

def prime_factors(n):
    """returns the prime factors of n"""
    i = 2
    factors = []
    while i * i <= n:
        if n % i:
            i += 1
        else:
            n //= i
            factors.append(i)
    if n > 1:
```

```
        factors.append(n)
    return factors
```

1.11. **Problem 407.** Just meeting the 1000s rule in Python. :(

But, it is good to be here, at last :)

As of now I cannot see how to speed things up, using this method: I find the prime factors of each n, and then, via the Chinese Remainder Theorem, I find one idempotent per prime factor, and then all others as a power set of combinations of these. I couldn't work out how to avoid calculating all a for each n.

I note the humbling contributions above, but it has been useful to finally understand how the CRT works and to devise code to implement it.

```
import time
import itertools as it

def p407(limit):
    t=time.clock()
    misum=0
    for n in range(2,limit+1):
        misum+=max_idempotent(n)
    print(misum)
    print(time.clock()-t)

def max_idempotent(n):
    """returns maximum idempotent a < n: a^2=a mod n"""
    pfs=pflist(n)
    pfnum=len(pfs)
    if pfnum==1:
        return 1 #idempotent=1 for primes or powers of primes

    #Use the CRT to find m 'base' idempotent solutions from m prime factors p_i^a_i
    idems=[]
    for i in range(pfnum):
        allButOnePfs=pfs[:i]+pfs[i+1:]
        xsum=0
        for i in range(pfnum-1):
            Ni=n//allButOnePfs[i]
            xsum+=inverse(Ni,allButOnePfs[i])*Ni
        idems.append(xsum % n)

    #generate all other idempotents from these, and return the maximum
    maxval=max(idems)
    for i in range(2,len(idems)):
        for a in it.combinations(idems, i):
```

```
            aprod=1
            for x in a:
                aprod*=x
                aprod=aprod%n
            if aprod>maxval:
                maxval=aprod
    return maxval

def pflist(n):
    """returns the distinct prime factors of n as [2^a,3^b.....]"""
    i = 2
    factors = []
    while i * i <= n:
        if n % i:
            i += 1
        else:
            factors.append(1)
            while not n %i:
                n //= i
                factors[-1]*=i
    if n > 1:
        factors.append(n)
    return factors

#with thanks to Wikipedia and numerous other sources
def inverse(a, n):
    """returns multiplicative inverse of a mod n. a and n must be-co-prime"""
    t1,t2=0,1
    r1,r2=n,a
    while r2!=0:
        q = r1 // r2
        t1, t2 = t2, t1 - q * t2
        r1, r2 = r2, r1 - q * r2
    if t1 < 0:
        t1 +=n
    return t1
```

1.12. **Problem 512.** I managed to work out that $g(n) = \sum_{i=1,i\nmid 2}^{n} \varphi(i)$ and then did the sum by brute-force sieving in 6 minutes.

$$\sum_{i=1}^{n} \phi\left(n^i\right) = \frac{\left(n^n-1\right)\phi(n)}{n-1}$$

$$(\left(\sum_{i=1}^{n} \phi\left(n^i\right)\right) \bmod (n+1)) = (\left((\frac{n^n-1}{n-1} \bmod (n+1))(\phi(n) \bmod (n+1))\right) \bmod (n+1))$$

$$(\frac{n^n-1}{n-1} \bmod (n+1)) = 1$$

Hence

$$g(n) = \sum_{i=1, i \nmid 2}^{n} \varphi(i)$$

```
import numpy as np
import time

def p512(n):
    t=time.clock()
    primes=primesieve(n)
    g=sum(etsieve(n,primes)[1::2])
    print(g,time.clock()-t)

def etsieve(n,primes):
    """return array of euler totient(x) for x from 2 to n"""
    sieve=np.array(range(n+1),dtype=float)
    for i in primes:
        if sieve[i]==i:
            sieve[i::i]*=(1-1/i)
    return sieve.astype(int)

def primesieve(n):
    """return array of primes 2<=p<=n"""
    sieve=np.ones(n+1,dtype=bool)
    for i in range(2, int((n+1)**0.5+1)):
        if sieve[i]:
            sieve[2*i::i]=False
    return np.nonzero(sieve)[0][2:]
```

To get it within one minute, I had to understand how to sum totients without actually calculating them. This is covered in daniel fischer's excellent overview to problem 73 and in [url=http://math.stackexchange.com/questions/316376/how-to-calculate-these-totient-summation-sums-efficiently.]this stack exchange page[/url] (see the post by 'Andy'), which

is also used by the very helpful post earlier in this forum from chsl95. This code (which is essentially the same as that of chsl95) goes in about 10s.

```
def p512v2(n):
    t=time.clock()
    print(oddTotientSum(n))
    print(time.clock()-t)

#implements stack exchange 'Andy'
#http://math.stackexchange.com/questions/316376/how-to-calculate-these-totient-summation-sums-
def R2(N,X2={}):
    if N==1:
        return 0
    try:
        return X2[N]
    except KeyError:
        fsum = F2(N)
        m=2
        while 1:
            x = N//m
            nxt = N//x
            if(nxt >= N):
                result=fsum - (N-m+1)*R2(N//m,X2)
                X2[N]=result
                return result
            fsum -= (nxt-m+1) * R2(N//m,X2)
            m = nxt+1

#returns sum of totients of x<=n
#wrapper for R2
#sum of totient(x) for x<=n
def totientSum(n):
    return R2(n)+1

#sum of totient(x) for x<=n and x is even
def evenTotientSum(N):
    if N < 2:
        return 0
    return totientSum(N//2)+evenTotientSum(N//2)

#sum of totient(x) for x<=n and x is odd (answer to PE p512)
def oddTotientSum(N):
    return totientSum(N)-evenTotientSum(N)
```

The key insight I got from chsl95 is how to calculate the sum of totients of even numbers.

1.13. **Problem 540:**

We use parametrization of Pythagorean triples: $a = p^2 - q^2, b = 2pq, c = p^2 + q^2$, where $q < p$, p and q have different parity and are relatively prime. If we require that $c \leq n$ then the number of valid p, q pairs, without restriction except that $p, q > 0, q < p$, is:

$$\begin{aligned} C(n) &= \sum_{q<p, p^2+q^2 \leq n} 1 \\ &= \sum_{2q^2<n} \left(\left\lfloor \sqrt{n - q^2} \right\rfloor - q \right) \end{aligned}$$

We need to subtract from this the number of p, q pairs that are either not co-prime or have the same parity.

1.14. **Problem 613: Pythagorean Ant**

I imagined four quadrants, with axes parallel to the triangle sides of length 3 and 4. If the ant walked off in the quadrant facing the hypotenuse it was certain to exit the triangle on that side, and if it walked off towards the right angle, it was certain not to. That left the other two quadrants. For each of these I derived an expression for the probability of exit across the hypotenuse, to find:

$$P(\text{exit}) = \frac{1}{4} \left(1 + \frac{2}{\pi} \left(\frac{1}{4} \int_0^4 \tan^{-1} \frac{x}{3} dx + \frac{1}{3} \int_0^3 \tan^{-1} \frac{x}{4} dx \right) \right)$$

which, with Mathematica's help, I find integrates to:

$$\frac{1}{2} + \frac{-25 \log 5 + 9 \log 3 + 32 \log 2}{24\pi}$$

where we use the fact that the two smaller angles of the triangle sum to $\pi/2$.