2 Algorithms in Number Theory

Number theory is a sprawling topic with a very long history, but benefits from the availability of many excellent textbooks, such as Ireland [**IR90**], Hardy [**HWHBS08**], and Niven [**NZM91**]. Silverman's *A Friendly Introduction to Number Theory* [**Sil12**] is particularly good for beginners, and several chapters are online at `https://www.math.brown.edu/johsilve/frint.html`. The *LibreTexts Mathematics* 'Combinatorics and Discrete Mathematics' section (`https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics`) includes several textbooks on number theory. Youtube has many videos on different topics, and an entire introductory course from Berkeley at `https://www.youtube.com/playlist?list=PL8yHsr3EFj53L8sMbzIhhXSA0puZ1Fov8`.

2.1 Greatest Common Divisor

2.1.1 Euclid's Algorithm. Let a and b be integers. We denote their greatest common divisor by $gcd(a,b)$. If we define $gcd(0,0) = 0$, then for all integers a, b

$$\gcd(a,1) = 1, \quad \gcd(a,a) = |a|, \quad \gcd(a,0) = |a|, \quad \gcd(a,b) = \gcd(b,a)$$

If $gcd(a,b) = 1$, we say that a and b are relatively prime, co-prime, or that one is prime to the other.

Let's assume $a \geq b > 0$, then a can be uniquely represented by b in the form

$$a = bq + r, \quad 0 \leq r < b. \tag{2.1}$$

The integers q and r are the quotient and remainder on division of a by b. Thus $q = \lfloor a/b \rfloor, r = a - \lfloor a/b \rfloor$, as depicted on the number line in Fig. 2.1.

In Python these operations are implemented as

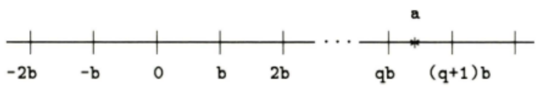


Figure 2.1. $a = bq + r$ on the number line

```
q = a // b;  r = a % b
```

Any divisor of a and b is also a divisor of $ax + by$ for any integers x, y. Thus in Equ. (2.1)

$$d|a,\ d|b \to d|r \quad \text{and} \quad d|b,\ d|r \to d|a.$$

where $a|b$ means that a *divides* b; i.e. b is a *multiple* of a for integers a, b. Hence

$$\gcd(a,b) = \gcd(b,r) = \gcd(b, a\%b). \tag{2.2}$$

By means of Equ. (2.2) we can replace the pair (a,b) by the smaller pair $(b, a\%b)$. Repeating this step, we obtain ever smaller pairs, until finally a pair $(g,0)$ is reached. Then

$$\gcd(a,b) = \gcd(g,0) = g.$$

This can be translated into the recursive function in Listing 2.1 (gcdrec.py), and its iterative version in Listing 2.2 (gcditer.py) is obtained by replacing the tail-recursion by a loop.

```
def gcd(a,b):
  if b == 0:
    return a
  else:
    return gcd(b, a % b)
```

Listing 2.1. Recursive GCD

```
def gcd(a,b):
  while b > 0:
    a, b = b, a%b
  return a
```

Listing 2.2. Iterative GCD

2.1.2 Euler's Totient Function.

$\phi(n)$ is the number of elements in $1..n$ which are co-prime with n.

Listing 2.3 (totient.py) computes $\phi(n)$ by using GCD to check for primality. $\phi(3000000) = 800000$ takes 46 seconds with the recursive version of GCD in Listing 2.1 but only 26 seconds with the iterative version from Listing 2.2.

```
import time
# from gcdrec import gcd
from gcditer import gcd

n = int(input("n=? "))
start_time = time.time()
```

```
phi = 0
for i in range(1,n+1):
  if gcd(n,i) == 1:
    phi += 1
print("phi(", n, ")==", phi)
print(f"Elapsed time: {(time.time() - start_time):.2f} secs")
```

Listing 2.3. Timing Euler's totient function

We could compute ϕ(n) faster by employing the formula

$$\phi(n) = n\frac{p-1}{p} \cdot \frac{q-1}{q}...$$

where $p, q, ...$ are the *distinct* prime factors of n. For example $\phi(150) = 40$ according to the above code, and also note that $150 = 2 \cdot 3 \cdot 5^2$. This means we can also calculate the result using:

$$\begin{aligned}\phi(150) &= 150 \cdot \tfrac{2-1}{2} \cdot \tfrac{3-1}{3} \cdot \tfrac{5-1}{5} \\ &= 5 \cdot 1 \cdot 2 \cdot 4 = 40\end{aligned}$$

This formula is derived from the Chinese Remainder Theorem (sec. 2.1.4), and is rather involved. A fairly simple explanation can be found at `https://math.stackexchange.com/questions/4346635/chinese-remainder-theorem-in-euler-phi-function`.

2.1.3 Extended Euclidean Algorithm.

The GCD of a and b can be written as a linear combination of a and b with integer coefficients x, y. That is, there exist integers x, y such that

$$\gcd(a, b) = ax + by.$$

x and y can be obtained by keeping track of what happens as we carry out Euclid's algorithm. We set up two columns to record the contributions of a and b to the remainders we compute. We start with the two lines

$$\begin{array}{llll} a & 1 & 0 & (\text{i.e. } a = 1 * a + 0 * b), \\ b & 0 & 1 & (\text{i.e. } b = 0 * a + 1 * b), \end{array}$$

Let $q = a \,//\, b$. Subtracting q-times row (2) from row (1) we get

$$r \quad 1 \quad -q \quad (\text{i.e. } r = 1 * a + (-q) * b).$$

Now we cross out row (1) and repeat the step with the remaining rows. The last line before r becomes zero will be

$$\gcd(a, b) = ax + by.$$

Listing 2.4 (egcd.py) returns $gcd(a, b)$, x, and y by using a recursive subfunction, egRec() to produce the rewrites.

```
def egcdRec(a, b):
  return egRec(a, 1, 0, b, 0, 1)

def egRec(g, x, y, g1, x1, y1):
  if g1 == 0:
    return g, x, y
  else:
    q = g // g1
    g2 = g - q*g1
    return egRec(g1, x1, y1, g2, x
         -q*x1, y-q*y1)
```

Listing 2.4. Recursive extended GCD

```
def egcd(a, b):
  x, x1 = 1, 0
  y, y1 = 0, 1
  g, g1 = a, b
  while g1 > 0:
    q = g // g1
    x, x1 = x1, x - q*x1
    y, y1 = y1, y - q*y1
    g, g1 = g1, g - q*g1
  return g, x, y
```

Listing 2.5. Iterative extended GCD

Probably the iterative version of the algorithm (Listing 2.5; egcd.py) is easier to understand since the changes to $g, x, y, g1, x1$, and $y1$ are more explicit. Examples:

```
> python egcd.py
a b? 13 7
gcd == a*x + b*y
gcd == 1 ; x == -1 ; y == 2

> python egcd.py
a b? 7 5
gcd == a*x + b*y
gcd == 1 ; x == -2 ; y == 3
```

Of course, both Listings 2.4 and 2.5 give the same answers.

2.1.4 The Chinese Remainder Theorem. The Chinese Remainder Theorem (CRT) involves $m_1, m_2, ..., m_k$ natural numbers, each greater than 1, and every pair co-prime.

Let $M = m_1 \cdot m_2 \cdot ... \cdot m_k$, and $b_1, b_2, ..., b_k$ be integers. There are a system of congruences involving x:

$$\begin{array}{ccll} x & \equiv & b_1 & \pmod{m_1} \\ x & \equiv & b_2 & \pmod{m_2} \\ & \vdots & & \\ x & \equiv & b_k & \pmod{m_k} \end{array}$$

The CRT states that there is a unique solution for x somewhere in the set {0,1, 2,..., M-1}. In other words, we are given k numbers which are pairwise co-prime, and the remainders when an unknown number x is divided by them. We need to find the minimum possible value of x that produces those remainders. The CRT tells us that such an x exists $0 \le x < M$.

An example:

$$\begin{aligned} x &\equiv 2 \pmod 3 \\ x &\equiv 3 \pmod 4 \\ x &\equiv 1 \pmod 5 \end{aligned}$$

The naive approach for finding x (see Listing 2.6; crt.py) is to start with 0 and check if dividing it by the given modulo values produces the corresponding remainders. If not, then keep incrementing x until an answer is found or M is reached. One optimization would be to first use gcd() to check that the pairs of modulo values were indeed all co-prime.

```python
def crt(mods, rems):
  M = math.prod(mods)
  n = len(mods)
  x = 1
  for x in range(M): # Check x against all mods and rems
    j = 0
    while j < n:
      if x % mods[j] != rems[j]:
        break
      j += 1
    # if all remainders  matched, we found x
    if j == n:
      return x

rems = [2, 3, 1]
mods = [3, 4, 5]
print("Rems=", rems, " Mods=", mods)
print("x =", crt(mods, rems));
```

Listing 2.6. Naive CRT implementation

The program outputs:

```
> python crt.py
Rems= [2, 3, 1]  Mods= [3, 4, 5]
x = 11
```

A more efficient approach requires the calculation of the *modulo multiplicative inverse*. The inverse of integer a modulo m is x such that:

$$ax \equiv 1 \pmod m$$

A value for x can be obtained by using the egcd() function. Recall that $egcd(a, b) = g, x, y$ where $ax + by = g$. So:

$$\begin{aligned} & \text{egcd}(a, m) &= &\ 1, x, y \\ \equiv\ & ax + my &= &\ 1 \\ & ax &\equiv &\ 1 \pmod m \end{aligned}$$

This can be coded as:

```
def modinv(a, m):
  g, x, y = egcd(a, m)
  if g != 1:
    raise Exception('modular inverse does not exist')
  else:
    return x % m
```

The modulo inverse gives a value for x that leaves a remainder of 1, which is a good starting point for finding values of x that give larger remainders.

A nice video introduction to CRT can be found at `https://www.youtube.com/watch?v=ru7mWZJlRQg`.

2.1.5 Visible Points in the Plane Lattice.

Let L_2 denote the plane lattice, i.e. the set of points (x, y) with integer coordinates. A point (x, y) in L_2 will be *visible* from the origin if $gcd(x, y) = 1$. Otherwise, the point is *invisible*. Fig. 2.2 shows that (9, 4) is visible which is confirmed by $gcd(9, 4) = 1$. In other words, since 4 and 9 are co-prime, then the diagonal from (0, 0) to (9, 4) does not intersect any other lattice points

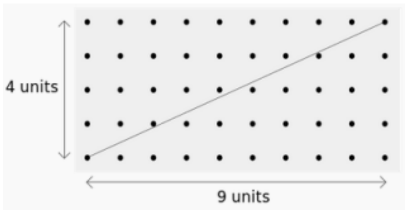


Figure 2.2. (9,4) is visible from (0,0)

Let $s(n)$ be the number of visible lattice points in the square $1 \leq x, y \leq n$. Then the proportion of visible lattice points in the square is $p(n) = s(n)/n^2$. What happens to $p(n)$ for $n \to \infty$? Each visible point (x, y) obscures infinitely many points $(kx, ky), k = 2, 3, 4, ...$ Hence we might guess that $p(n) \to 0$. Listing 2.7 (visible.py) computes $p(n)$ for input n to see if this hypothesis might be true.

```
from gcdrec import gcd

n = int(input("n=? "))
s = 0
for i in range(1, n+1):
  for j in range(1, n+1):
    if gcd(i,j) == 1:
      s += 1
p = s/n/n
print(f"s == {s} p == {p:.8f}")
```

Listing 2.7. How many visible lattice points?

n	s(n)	p(n)
100	6087	0.60870000
500	152231	0.60892400
1000	608383	0.60838300
5000	15200915	0.60803660

Table 2.1. Output from visible.py.

The stability of $p(n)$ for various values of n in Table 2.1 suggests that on the contrary $\lim_{n\to\infty} p(n) \approx 0.6080$. But $n = 5000$ isn't particularly large, so let's consider $n = 30$ million. Now we have to test $n^2 = 900$ million points, a large task. So we settle for a random sample of $m = 1$ million points and find the proportion of visible points in the sample. Listing 2.8 (randCoprime.py) estimates p(n) to be 0.60842700.

```
import random
from egcd import gcd

M = 1000000
N = 3000000
count = 0
for i in range(M):
  a = random.randint(1,N)
  b = random.randint(1,N)
  if gcd(a, b) == 1:
    count += 1

print(f"{count/M:.8f}")
```

Listing 2.8. Random testing for co-primality

We now sketch two ways of obtaining the value of the probability $p(n)$ for large values of n. To make the proof rigorous we must deal with the fact that for any finite n the equations we use are only approximately true. We should also prove that p() approaches the same limit for shapes other than squares, e.g. circles.

a) Let A_d represent $gcd(x, y) = d$. A_d is equivalent to the occurrence of three independent events: x is a multiple of d, y is a multiple of d, and $gcd(\lfloor x/d \rfloor, \lfloor y/d \rfloor) = 1$. Thus the probability of the event A_d is

$$P(A_d) = \frac{1}{d} \cdot \frac{1}{d} \cdot p \ = \ \frac{p}{d^2}.$$

Note: if your grasp of probability is a bit rusty, please read Appendix A.

Since any one of the disjoint events $A_1, A_2, A_3, \ldots$ must occur, we have

$$p/1 + p/4 + p/9 + \ldots = 1,$$

$$p(\sum_{i=1}^{\infty} 1/i^2) = 1$$

and using Euler's result $\sum_{i=1}^{\infty} 1/i^2 = \pi^2/6$, we find that $p = 6/\pi^2 \approx 0.6079271019$.

b) Let q and r be different primes. In the sequence of natural numbers we consider the subsequences $a_n = qn, b_n = rn, c_n = qrn$ of all multiples of q, r, qr. A randomly chosen natural number falls into the subsequences a_n, b_n, c_n with probabilities $1/q$, $1/r$, $1/qr$, respectively.

Since $1/(qr) = (1/q)(1/r)$, we can say that divisibility by q and r are independent events. If we choose the natural numbers x, y at random, then both will fall into a_n with probability $1/q^2$, and q is not a common factor of x and y with probability $1 - 1/q^2$. The events that neither x nor y is divisible by q are independent for different primes q. So the probability for $gcd(x, y) = 1$ is

$$\begin{aligned} p &= \prod(1 - \frac{1}{q^2}) \quad \text{(over all primes } q\text{)}, \\ \frac{1}{p} &= \frac{1}{1-1/2^2} * \frac{1}{1-1/3^2} * \frac{1}{1-1/5^2} * \ldots, \\ &= \sum_{n\ge0} \frac{1}{2^{2n}} * \sum_{n\ge0} \frac{1}{3^{2n}} * \sum_{n\ge0} \frac{1}{5^{2n}} * \ldots \\ &= \sum_{n\ge1} \frac{1}{n^2} = \frac{\pi^2}{6} \end{aligned}$$

By multiplying out the brackets we got the reciprocal of each square exactly once. This follows from the unique factorization theorem. Hence

$$p = \frac{6}{\pi^2} \approx 0.6079271019$$

How can we find the sum $s = \sum_{n\ge1} \frac{1}{n^2}$ to the maximum accuracy of real arithmetic in Python, 15 digits, without using Euler's result? Let $s = 1 + 1/4 + \ldots + 1/n^2 + E(n)$, and we want to estimate the error $E(n) = 1/(n+1)^2 + 1/(n+2)^2 + \ldots$

For large values of k, k^2 is relatively close to $(k - \frac{1}{2})(k + \frac{1}{2})$ and hence

$$\frac{1}{k^2} \approx \frac{1}{(k-\frac{1}{2})(k+\frac{1}{2})} = \frac{1}{k-\frac{1}{2}} - \frac{1}{k+\frac{1}{2}}.$$

Consequently

$$\begin{aligned} E(n) &= \frac{1}{(n+1)^2} + \frac{1}{(n+2)^2} + \ldots \\ &\approx \left(\frac{1}{n+\frac{1}{2}} - \frac{1}{n+\frac{3}{2}}\right) + \left(\frac{1}{n+\frac{3}{2}} - \frac{1}{n+\frac{5}{2}}\right) + \ldots \\ &= \frac{1}{n+\frac{1}{2}}. \end{aligned}$$

The right side is $> E(n)$, but by a factor $\leq (n+1)^2/((n+1)^2 - 1/4)$. Having found a good estimate of E(n), we compute $s(n) = 1 + ... + 1/n^2$ starting at the end (to minimize rounding errors) and add the correcting term $1/(n+0.5)$ (Listing 2.9; sumInvSquares.py). We test $n =$ 50000, 100000, ... until there is no further change, producing Table 2.2.

```
n = int(input("n=? "))
sum = 0.0
for i in range(n, 0 ,-1):
  sum += 1.0/(i*i)
sum += 1.0/(n+0.5)
print(sum)
```

Listing 2.9. Sum inverse squares

n	sum (n)
50000	1.6449340668482269
100000	1.6449340668482266
150000	1.6449340668482264
200000	1.6449340668482264

Table 2.2. sum Inverse Squares Results

```
def gcdBinary(u,v):
  if u == v:
    return u
  if (u%2==0) and (v%2==0): #even
    return 2*gcdBinary(u//2, v//2)
  if (u+v)%2 == 1:
    if v%2 == 1:
      return gcdBinary(u//2, v)
    else:
      return gcdBinary(u, v//2)
  if (u%2 == 1) and (v%2 == 1):
    if u > v:
      return gcdBinary(u-v, v)
    elif v > u:
      return gcdBinary(u, v-u)
```

Listing 2.10. Recursive binary GCD

```
def gcdBinaryIt(u,v):
  x = u; y = v
  k = 1
  while (x%2==0) and (y%2==0):
    x = x//2;  y = y//2
    k *= 2
  while x != y:
    while x%2 == 0:
      x = x//2
    while y%2 == 0:
      y = y//2
    if x > y:
      x -= y
    if y > x:
      y -= x
  return x*k
```

Listing 2.11. Iterative binary GCD

2.1.6 The Binary GCD Algorithm.

The binary GCD algorithm is based on the relations

$$\begin{aligned} u \text{ even}, v \text{ even} &\quad\to\quad \gcd(u,v) = 2 * \gcd(u \mathbin{/\!/} 2,\ v \mathbin{/\!/} 2) \\ u \text{ even}, v \text{ odd} &\quad\to\quad \gcd(u,v) = \gcd(u \mathbin{/\!/} 2,\ v) \\ u > v &\quad\to\quad \gcd(u,v) = \gcd(u-v, v) \\ u \text{ odd}, v \text{ odd} &\quad\to\quad u - v \text{ even}, |u-v| < \max(u,v). \end{aligned}$$

This algorithm can be made a little faster by employing Python's bitwise right shift operator (»), although the performance difference from using integer division (//) will be small. Indeed, many compilers automatically convert power-of-two multiples and divides into left and right bit shifts when the shift amount is known at compile time (as here).

Listing 2.10 (gcdBinary.py) is recursive, and closely matches the definition above, while Listing 2.11 (gcdBinaryIt.py) is an iterative version.

2.1.7 GCD and LCM.

The following algorithm due to Stanley Gill [**Dij76**] computes $gcd(a, b)$ and $lcm(a, b)$ at the same time, employing the fact that $lcm(a, b) = ab/gcd(a, b)$.

We start with $x = a, y = b, u = a, v = b$ and progress as follows:

```
if x < y:
  y -= x; v += u
if y < x:
  x -= y; u += v
```

The algorithm ends with $x = y = gcd(a, b)$ and $(u + v)/2 = lcm(a, b)$. The invariants of these transformations are

$$\begin{aligned} P &: \quad \gcd(x,y) = \gcd(x-y,y) = \gcd(x,y-x), \\ Q &: \quad xv + yu = 2ab, \\ R &: \quad x > 0, y > 0. \end{aligned}$$

P and R are obviously invariant, but we'll show the invariance of Q. At the beginning we have $ab + ab = 2ab$, which is clearly correct. At the next step, the left side of Q becomes either

$$x(v+u) + (y-x)u = xv + yu \quad \text{or} \quad (x-y)v + y(u+v) = xv + yu.$$

Thus, the left side of Q does not change. At the end we have

$$x = y = \gcd(a,b) \quad \text{and} \quad \frac{u+v}{2} = \frac{ab}{\gcd(a,b)} = \text{lcm}(a,b).$$

Listing 2.12 (gcdlcm.py) implements this algorithm.

```
def gcdlcm(u, v):
  x = u; y = v
  while x != y:
    if x < y:
      y -= x
      v = u+v
    else:
      x -= y
      u = u+v
  return x, (u+v)//2    # gcd, lcm
```

Listing 2.12. Calculating GCD and LCM

Exercises

(1) Let $a = 8991, b = 3293$.

a) Find $gcd(a, b)$.

b) Find a solution (x, y) of $gcd(a, b) = ax + by$ using Listing 2.4.

c) Find $gcd(a, b)$ using Listing 2.10.

(2) Find $gcd(a, b)$ and $lcm(a, b)$ for $a = 2431$ and $b = 1309$ using Listing 2.12.

(3) Let a, b be integers made up of at most 11 digits. Find $gcd(a, b)$ for

a) $a = 987654321, b = 123456789$

b) $a = 9753197531, b = 2468008642$

c) $a = 12345678901, b = 10234567891$.

(4) Find the last three (or four) digits of a) 7^{9999} b) 7^{99999} c) 7^{999999}

(5) Make Listing 2.7 twice as efficient by using the fact that if $t(n)$ is the number of visible points in the triangle $0 < y < x \leq n$, then $s(n) = 2t(n) + 1$.

(6) Write an iterative and a recursive function to find $1 + 2 + ... + n$.

(7) Write a recursive and an iterative function to find the binary representation of n.

(8) Write a function which computes the digital sum $d(n)$ of n. For instance, $d(1234) = 10$.

(9) Let $d(n)$ be the digital sum of n. Write a program which finds $d(d(n)), d(d(d(n))), ...$ until a single digit number $f(n)$ is reached. Show that $f(mn) = f(f(m)f(n))$.

(10) Write a function which reverses the digits of an integer. For instance, $2024 \mapsto 4202$.

(11) Write a slow recursive function for $C(n, s)$, the number of s-subsets of an n-set, based on the recursion $C(n, s) = C(n - 1, s - 1) + C(n - 1, s), C(n, 0) = C(n, n) = 1$.

(12) Write a fast recursive function for $C(n, s)$ based on the recursion $C(n, s) = C(n - 1, s - 1) * n/s, C(n, 0) = 1$.

(13) Write a program which prints fib(n+1)/fib(n) for $n = 1$ to 30. What do you see, and can you prove it?

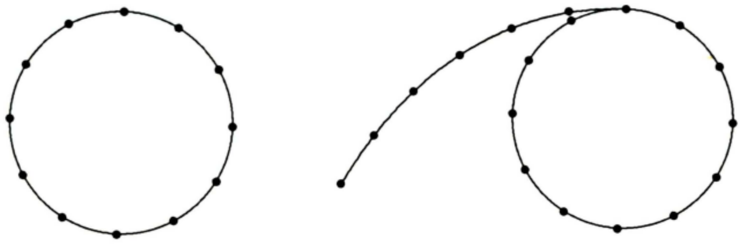
(14) Print those terms of the Fibonacci sequence which are divisible by some fixed positive integer d. Experiment with different values of d, make observations and conjectures. Prove your conjectures.

(15) The Fibonacci sequence $\pmod{m}$ will eventually become periodic.

a) Why?

b) A periodic sequence can be either purely periodic or periodic with a tail, as in Figs. 2.3a and 2.3b. Show that the Fibonacci sequence $\pmod{m}$ is always purely periodic.

c) Write a program which finds the period $L(m)$ for the modulus m.



(a) An immediate cycle. (b) An eventual cycle.

Figure 2.3. Periodic sequences

(16) Experiment with the program in Ex. 15c) and make conjectures about

a) $L(p^n)$ for any prime p.

b) $L(p_1^{a_1}, p_1^{a_2}, ..., p_1^{a_r},)$ for primes p_1 to p_r.

By means of b), conjecture a formula for $L(10^n)$.

(17) Formulate theorems about the length L of the period of the Fibonacci sequence $\pmod{m}$ if the prime p has the form

a) $p = 5k \pm 1$ b) $p = 5k \pm 2$ c) $p = 5$

(18) The Tribonacci sequence is defined by $t(1) = t(2) = t(3) = 1, t(n) = t(n-l) + t(n-2) + t(n-3), n > 3$.

a) Why is this sequence purely periodic $\pmod{m}$?

b) Find the period $L(m)$ for the modulus m and discover some theorems about $L(m)$.

(19) Find Zeta3 $= 1 + 1/2^3 + 1/3^3 + 1/4^3 + ...$, in a way similar to Listing 2.9.

(20) Define a lattice point (x, y, z) to be visible if $gcd(x, y, z) = 1$. Rewrite Listings 2.7 and 2.8 for 3D space. Guess a formula for the probability that $gcd(x, y, z) = 1$. Can you give a heuristic argument for this formula?

(21) Guess a formula for the probability that k random integers have a $GCD = d$. Check this formula empirically. Give a heuristic argument for this formula.

(22) *The Morse-Thue Sequence.* This infinite sequence of binary digits can be constructed recursively: start with 0; to each initial segment append its complement: $0, 01, 0110, 0110\,1001, 01101001\,10010110, \ldots.$ Thus, having constructed the first $1 + 1 + 2 + 4 + 8 + 16 + \ldots + 2^n = 1 + 2^{n+1}$ digits, the next block of $1 + 2^{n+1}$ is determined.

a) Show that the sequence is aperiodic.

b) The sequence has a much more interesting property: it is self-similar. Striking out every second digit reproduces the sequence. Prove this.

c) Let the digits of the sequence be $x(0), x(1), x(2), \ldots.$ Show that $x(2n) = x(n), x(2n+1) = 1 - x(2n)$. Write a recursive algorithm to find $x(n)$ based on these recurrences.

d) Show that $x(n) = 1 - x(n-2^k)$, where 2^k is the largest power of 2 contained in n, i.e. the largest power of 2 which is $\leq n$. Write an algorithm to find $x(n)$ based on this property.

e) Write the non-negative integers in binary notation: 0, 1, 10, 11, 100, 101, 110, 111, ... Now replace each number by the sum of its digits $\pmod 2$. You get the sequence 01101001..., which is the Morse-Thue sequence once again. Prove this and write an algorithm for finding the n-th digit $x(n)$ based on this idea.

This exercise and the next were inspired by Jacobs [**Jac69**].

(23) The binary *Keane Sequence* $x(0)\,x(1)\,x(2)\ldots = 001001110001001110110110001\ldots$ is formed as follows (with commas separating the successively longer segments):

$$0, 001, 001001110, \ldots, A, AAC(A), \ldots$$

where $C(A)$ is the digit-wise complement of the binary block A.

a) Construct an algorithm which finds the n-th digit $x(n)$. *Hint*: translate 0,1,2,3, ... into the ternary system. Do you see a pattern?

b) Show that $x(3n) = x(n), x(3n+1) = x(n), x(3n+2) = 1 - x(n)$. Write a program based on these recurrences.

(24) Rewrite Listing 2.11 to use bit shifting as explained in the text. Test its speed versus the GCD function currently used in Listing 2.8).

(25) a) Print one row of Pascal's triangle, i.e. the sequence $C(n, 0) \ldots C(n, n)$. Use a list generated by addition, as in Table 2.3.

b) Modify a), so that rows 0 to n of Pascal's triangle are printed.

c) Write a function which prints Pascal's triangle $\pmod 2$ and study the recursive pattern.

1	5	10	10	5	1	
	1	5	10	10	5	1
1	6	15	20	15	6	1

Table 2.3. Pascal Triangle row

(26) a) A woman is walking to the market with n eggs in her basket, and a man on a bicycle knocks her down, causing all the eggs to break. The cyclist offers to pay for all n eggs, but she can't remember n exactly. She only knows that by grouping them 2,3,4,5,6 at a time, one egg was always left over, but when they were grouped into 7's, no eggs were surplus. Find all numbers ≤ 3000 satisfying these properties.

b) Which is the most likely number if one egg weighs about 56 grams?

c) Generalize.

(27) Consider Perrin's sequence $v_0 = 3, v_1 = 0, v_2 = 2, v_n = v_{n-2} + v_{n-3}, n > 2$. Write a program which prints a table of those numbers satisfying $3 \leq n \leq 4001$ and $n|v_n$ Compare this table with the table printed in Listing 2.19. Do you think there is enough evidence for the theorem $n|v_n \Leftrightarrow n$ is prime? We will return to this problem later.

(28) *Analysis of an Algorithm*. Start with two piles of a and b chips. You may double the number of chips in the smaller pile by taking them from the larger one, but must stop as soon as the first pile is reduced to 0. This can be summarized as:

```
while a > 0:
  if a < b:
    (a, b) = (2a, b-a)
  else:
    (a, b) = (a-b, 2b)
```

a) For what initial positions does the algorithm stop, and after how many steps?

b) When do you get a pure cycle (e.g. like Fig. 2.3a), and how long is it?

c) When do you get a cycle with a tail (e.g. like Fig. 2.3b), and how long is the tail?

d) Suppose a and b are positive rationals. Answer a) to c) in that case.

e) Let a and b be positive reals. Now answer a) to c) in that case. Hint: you can simplify the algorithm substantially by noting that $a + b$ is an invariant of the transformation.

(29) Find a recursive equation for the number $f(n)$ of ones in the binary representation of n. Hint: express $f(n)$ for $n > 0$ in terms of values of $f()$ for smaller arguments.

(30) By a systematic search find a number n with the property $\phi(n) = \phi(n+1) = \phi(n+2)$. See Listing 2.1.2 for more details on $\phi()$.

(31) $f()$ is defined by $f(1) = 1, f(2n) = f(n)+g(n-1), f(2n+1) = f(n+1)+g(n/2)$ for all positive integers n. Here, $g(x)$ is the smallest power of $2 > x$, where x is a real. Write a recursive program which tests that $f(f(n)) = n$ for any input n. A function with this property is called an *involution*.

2.2 Representing n in the Form $x^2 + y^2$

We want to generate all non-negative integer solutions of $x^2 + y^2 = n$ with $\sqrt{n} \geq x \geq y \geq 0$. It pays to subdivide the task into three parts:

```
1. Initialization.
2. Go from A to B.
3. Go from B to C.
```

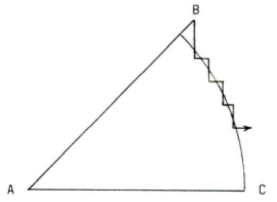


Figure 2.4. B to C

Stage 3 is illustrated in Fig. 2.4. The resulting code is in Listing 2.13 (quadsum1.py).

```
n = int(input("n=? "))
x = 0; y = 0
while x*x + y*y < n:
  # go from A to B
  x += 1
  y += 1
while x*x <= n:
  # go from B to C
  while x*x + y*y > n:
    y -= 1
  if x*x + y*y == n:
    print("x ==", x, "; y ==", y)
  x += 1
```

Listing 2.13. Printing integer squares

Stage 2 (going from A to B) can be optimized to a single step, resulting in Listing 2.14 (quadsum2.py).

```
n = int(input("n=? "))
x = math.trunc(math.sqrt(n/2))
y = x    # jump from A to B
while x*x <= n:
  # go from B to C
  while x*x + y*y > n:
    y -= 1
  if x*x + y*y == n:
    print("x ==", x, "; y ==", y)
  x += 1
  y -= 1
```

Listing 2.14. Printing integer squares (optimized)

We've utilized the fact that the part of the circle we are concerned with (i.e. B to C) descends more steeply than 45°. Consequently, if $(x, y+1)$ is outside the circle then so is $(x+1, y)$. That allows us to bypass the latter point and go directly to $(x+1, y-1)$.

2.3 Pythagorean Triples

We now extend Listing 2.14 so it prints a table of primitive Pythagorean triples, i.e. all the positive integer solutions (x, y, z) such that

$$x^2 + y^2 = z^2,\ \gcd(x, y) = 1,\ z \leq \ \text{max}.$$

z must be odd because if it was even then x and y would have the same parity. Also, if x and y are both even then the triple can't be primitive. If they're both odd then $x^2 + y^2 = z * 2$ cannot be true modulo 4.

These conditions mean that we only need to run through the odd values of z, and print an (x, y, z) triple only if $gcd(x, y) = 1$. The result is Listing 2.15 (pyttrip1.py).

```
max = int(input("max=? "))
z = 1
while z <= max:
  z += 2
  n = z*z
  x = math.trunc(math.sqrt(n/2))
  y = x
  while x*x <= n:
    while x*x + y*y > n:
      y -= 1
    if x*x + y*y < n:
      x += 1
    else:
      if gcd(x,y) == 1:
        print(f"{y:4} {x:4} {z:5}")
      x += 1
      y -= 1
```

Listing 2.15. Prints all triples with z ≤ max

The values in Table 2.4 are output when max is set to 100.

Primitive Pythagorean triples (x, y, z) can also be represented as

$$\begin{gathered} x = a^2 - b^2, \qquad y = 2ab, \qquad z = a^2 + b^2, \\ a > b, \qquad \gcd(a, b) = 1, \qquad a + b \text{ odd}; \end{gathered}$$

y	3	5	8	7	20	12	9	28	11	33	16	48	36	13	39	65	20
x	4	12	15	24	21	35	40	45	60	56	63	55	77	84	80	72	99
z	5	13	17	25	29	37	41	53	61	65	65	73	85	85	89	97	101

Table 2.4. Pythagorean triples

Listing 2.16 (pyttrip2.py) is based on these formulas. Note that max is now linked to a rather than z, and so a slightly larger number of triplets is printed since z can increase above 101.
One useful change to both of these programs would be to store the triplets in a list, which could then be sorted.
For an excellent introduction to this way of solving the Pythagorean equation (and the rest of number theory), see Ore **[Ore17]**.

```
max = int(input("max=? "))
a = 2; b = 1
while a <= max:
  while b > 0:
    if gcd(a,b) == 1:
      print(f"{a*a - b*b:4} {2*
          a*b:4} {a*a + b*b:5}"
          )
    b -= 2
  b = a
  a += 1
```

Listing 2.16. Pythagorean triplets

2.4 Counting the Lattice Points in a Ball

Let $lat(s, n)$ be the number of lattice points in the closed n-ball

$$x_1^2 + x_2^2 + ... + x_n^2 \leq s.$$

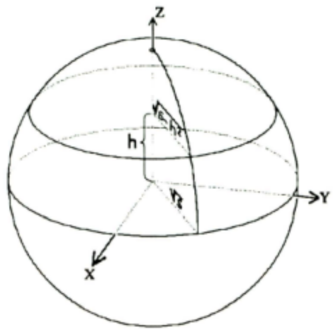
In Fig. 2.5a we use $n = 3$ for illustration. The equatorial section is an $(n-1)$-ball with radius $\sqrt{s}$.

By definition there are $lat(s, n-1)$ lattice points on the ball. To those we must add the latitudinal sections at heights $h = 1, 2, ..., \lfloor\sqrt{s}\rfloor$ above the equatorial section.

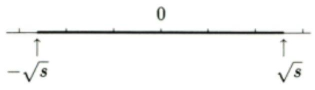
In Fig. 2.5a, the latitudinal section with height h is an $(n-1)$-ball with radius $\sqrt{s-h^2}$ which contributes $lat(s-h^2, n-1)$ lattice points.

For symmetry reasons we ignore the sections in the "southern hemisphere" and instead count the sections above the equator twice. Thus, we get the program fragment:

```
count = lat(s, n-1)
for h in range(int(math.sqrt(s)):
  count += 2 * lat(s-h*h, n-1)
```



(a) A 3D ball



(b) A 1D ball for $x^2 \le s$

Figure 2.5. Lattice balls

The cleanest termination condition would be:

```
if n == 0:
  lat = 1
```

That is, the 0-ball contains just the origin. Or we could set $lat(s, 1) = 1+2\lfloor\sqrt{2}\rfloor$ (Fig. 2.5b). Then the termination condition would become:

```
if n == 1:
  lat = 1 + 2*int(math.sqrt(s))
```

Listing 2.17 (lattice.py) is based on this approach.

```
def lat(s, n):
  if n == 1:
    return 1 + 2*int(math.sqrt(
        s))
  else:
    count = lat(s,n-1)
    h = 1
    while h*h <= s:
      count += 2*lat(s-h*h, n
          -1)
      h += 1
    return count
```

Listing 2.17. Number of lattice points

Table 2.5 was computed using this function, with s varying as in the table and n set to 2 (i.e. we generated lattice points inside a 2D ball, better known as a circle).

The code is arguably the simplest way to compute π (3.141592653589793) – by counting. The main drawback is that errors will start to accumulate due to all the calls to math.sqrt().

In 1800, Gauss developed a simple estimate for the error bound for the circle's lattice points (the Gauss circle problem; `https://mathworld.wolfram.com/GausssCircleProblem.html`). To each point in or on the circle he assigned the unit square "north-east" of the point. The sum of the areas of these squares is $f(s)$:

$$f(s) = \text{lat}(s, 2) = \pi s + e(s),$$

where $e(s)$ is the error.

s	lat(s, 2)
10	37
100	317
1000	3149
10000	31417
100000	314197
1000000	3141549
10000000	31416025
100000000	314159053
1000000000	3141592409
10000000000	31415925457

Table 2.5. Lattice points in a circle

This is not quite equal to the actual area of the circle since some squares protrude beyond it and there are some unfilled areas, as can be seen in Fig. 2.6.

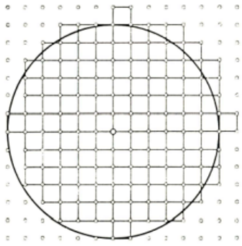


Figure 2.6. A 2D lattice ball

However, the circle about the origin with radius $\sqrt{s}+\sqrt{2}$ covers these squares completely and the circle about the origin with radius $\sqrt{s}-\sqrt{2}$ is completely covered by these squares. Thus, we have the bounds:

$$\pi(\sqrt{s}-\sqrt{2})^2 < f(s) < \pi(\sqrt{s}+\sqrt{2})^2$$

which implies

$$|f(s)-\pi s| < 2\pi(\sqrt{2s}+1)).$$

We can write this less precisely, for large s, as

$$|f(s)-\pi s| < C\sqrt{s}, \text{ where C is some constant}$$

It took over 100 years to find a better estimate, when Sierpinski proved the surprising result in 1906:

$$|f(s)-\pi s| < Cs^{1/3}, \text{ where C is some constant}$$

Hardy and Landau followed this by showing that there is **no** constant C such that

$$|f(s)-\pi s| < Cs^{1/4}.$$

It's currently believed that, for each t > 1/4, there is a number C_t such that

$$|f(s)-\pi s| < C_t s^t,$$

but as of 2003 this has only been proved for $t > 131/416$.

Exercises for Sections 2.2 to 2.4

(1) Find the number of representations as a sum of two squares of the following integers:
$5, 13, 17, 29, 5*13 = 65, 5*13*17 = 1105, 5*13*17*29 = 32045, 5^2*13 = 325$, $5^3 * 13 = 1625, 5^2 * 13^2 = 4225, 5^3 * 13^3$. Devise theorems about the number of representations.

(2) Write a program which finds all lattice points on the circle $x^2 + y^2 = n$ by starting at C and going to B (as depicted in Fig. 2.4).

(3) If $n \bmod 4 = 3$ then $x^2 + y^2 = n$ has no solutions. Show this. Use this fact to write a brute force algorithm for finding all Pythagorean triples. The variable a runs from 2 to $c - 1$, and print a, b, c if $b = \sqrt{c^2 - a^2}$ is an integer. How do you test if b is an integer? Only one of a, b, c and b, a, c should be printed.

(4) Make Listing 2.14 more efficient by using the fact that

$$(x + 1)^2 = x^2 + (2x + 1), \quad (y - 1)^2 = y^2 - (2y - 1).$$

(5) There is another formula for the number $f(s)$ of lattice points in the circle $x^2 + y^2 \le s$:

$$f(s) = 1 + 4(\lfloor s \rfloor - \lfloor s/3 \rfloor + \lfloor s/5 \rfloor - \lfloor s/7 \rfloor + \lfloor s/9 \rfloor - \ldots). \tag{2.3}$$

Find by means of this formula a) $f(10000)$ b) $f(1000000)$. Hint for b): with $s = 500000$ we have $\lfloor 2s/(s+1) \rfloor - \lfloor 2s/(s+3) \rfloor + \ldots + \lfloor 2s/(2s-3) \rfloor - \lfloor 2s/(2s-1) \rfloor = +1 - 1 + 1 - 1 + \ldots + 1 - 1 = 0$. Using similar relations reduce the number of terms from 500000 to about 50000.

Equ. (2.3) is proved in Hilbert and Cohn-Vossen [**HCV99**] and Shanks [**Sha78**].

(6) Evaluate lat(10000, 3) and give an estimate of π based on this number. Compare it with the estimate obtained with lat(10000, 2).

(7) The volume of the 4-ball is $v_4(r) = c_4 r^4$, where c_4 is the volume of the unit ball in 4-space. Estimate the constant c_4 by counting lattice points using lat(100, 4) and lat(1000, 4), which give estimates for c_4.

a) Can you guess the exact value of c_4?

b) Divide the estimate by π. Can you now guess c_4?

c) Multiply the estimate by 2 and divide by π. Can you now guess c_4?

(8) The circle occupies $\pi/4$ (roughly 3/4) of the circumscribed square. The 3-ball occupies $\pi/6$ (roughly 1/2) of the circumscribed cube. Roughly what proportion of the circumscribed cube do the 4-, 5-, and 6-balls occupy?

(9) The following formula was known to Gauss:

$$\mathrm{lat}(s,2) = 1 + 4\left\lfloor\sqrt{s}\right\rfloor + 4\left\lfloor\sqrt{s/2}\right\rfloor^2 + 8\sum_{i=a}^{b}\left\lfloor\sqrt{s-i^2}\right\rfloor$$
$$\text{with } a = \left\lfloor\sqrt{s/2}\right\rfloor + 1 \text{ and } b = \left\lfloor\sqrt{s}\right\rfloor$$

Prove this formula. Write a program based on it, and compare its speed with that of Listing 2.17.

(10) From Equ. 2.3 in Ex. 5 we can derive Leibniz's formula:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

However, this formula was known at least a century earlier in India with three successively better error estimates:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \ldots + \frac{1}{(n-3)} - \frac{1}{(n-1)} + f(n);$$
$$f(n) \approx \frac{1}{2n}, \quad f(n) \approx \frac{n}{2n^2+2}, \quad f(n) \approx \frac{n^2+4}{2n^3+10n}.$$

Find π using the above sum and third remainder formula with $n = 20$ and $n = 50$.

Sources for this formula can be found in C.T. Rajagopal and M.S. Rangachari, "On an Untapped Source of Medieval Keralese Mathematics", *Archive for History of Exact Sciences* 18 (1978), 81-101 and "On Medieval Kerala Mathematics", 35 (1986), 91-99.

2.5 Sieves

Sieving is when we strike out all elements from a set which don't possess a certain property. Sieves play a fundamental role in computer science and mathematics.

2.5.1 Square-free Integers. An integer is called *square-free* if it's not divisible by the square of an integer > 1. Let $A(n)$ be the number of square-free integers in the set 1..n, and $q(n) = A(n)/n$ is the fraction of square-free integers in that range. If

$$q = \lim_{n\to\infty} q(n)$$

exists, we call it the *density* of square-free integers. We'll compute $q(500)$ to $q(15000)$ in steps of 500 in Listing 2.18 (`squareFree.py`) to get some idea of q.

```
n = int(input("n=? "))
sqs = [False]*(n+1); i = 2
while (i * i <= n):
  for k in range(0, n+1, i*i):
    sqs[k] = True
  i += 1
s = 0
print("  n       q(n)")
for i in range(1, n+1):
  if not sqs[i]:
    s += 1
  if i % 500 == 0:
    print(f"{i:5} {s/i:10.8f}")
```

Listing 2.18. Counting square-free integers

We set each element of the list sqs to False to indicate that all of its values are initially not divisible by a square. Then we sieve with square factors (e.g. test i^2, $2i^2$, $3i^2$, ...) for $k = 0 \cdots n$, and if k is divisible by one of them then it is 'marked' by setting sqs[k] = True.

```
> python squareFree.py
n=? 15000
  n       q(n)
  500 0.61200000
 1000 0.60800000
 1500 0.61000000
  :  # more lines, not shown
14000 0.60778571
14500 0.60772414
15000 0.60800000
```

The output suggests that $q \approx 0.608$, and we conjecture that $q \approx 6/\pi^2$ (0.607927). The exercises explore this in more detail.
The printed sequence gets closest to $6/\pi^2$ at $n = 11000$, and then moves away. Is this due to accumulated errors or a property of the sieving?

2.5.2 The Sieve of Eratosthenes. The following algorithm is attributed to Eratosthenes of Cyrene, a 3rd Century BCE Greek mathematician:

(1) Set $p = 2$.

(2) Strike out all multiples of $p \geq p * p$.

(3) Set $p =$ the first integer beyond p not yet struck out, and jump to 2.

```
n = int(input("n=? "))
primes = [True]*(n+1); p = 2
while (p * p <= n):
  if primes[p]:
    for i in range(p*p, n+1, p):
      primes[i] = False
  p += 1
count = 1
for p in range(2, n+1):
  if primes[p]:
    print(f"{p:6}", end='')
    count += 1
  if count % 10 == 0:
    print(); count = 1
```

Listing 2.19. The sieve of Eratosthenes

The primes $\leq$ 200 produced by eratsiev.py:

```
> python eratsiev.py
n=? 200
     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
   127   131   137   139   149   151
   157   163   167   173   179   181
   191   193   197   199
```

2.5.3 A Closed Formula for an Irregular Sequence.

The sequence a_n:

```
1, 3, 4, 6, 8, 9, 11, 12, 14, 16, 17, 19, 21, 22, 24, 25, 27...
```

was produced by the following sieve:

- Take $a_1 = 1$ and delete $a_1 + 1 = 2$.
- Keep the next integer $a_2 = 3$, but delete $a_2 + 2 = 5$.
- Keep the next integer $a_3 = 4$, but delete $a_3 + 3 = 7$, etc.

The program makes use of two lists, x and a, as depicted in Fig. 2.7.

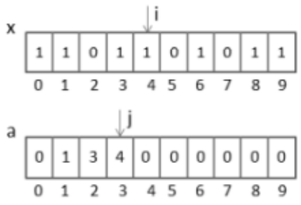


Figure 2.7. Irregular sequence data structures

The x list is used as a source of available numbers, such that if $x[i] = 1$ then the number i is available. If $x[i] = 0$ then it has been 'deleted' by the sieve. The a list holds the growing irregular sequence.

Listing 2.20 (irrSieve.py) finishes by calculating $a[j]/j$ and plots the values in the a list.

```
n = int(input("n=? "))
x = [1]*(2*n+1)  # available numbers
a = [0]*(2*n+1)  # irregular sequence
i = 1; j = 1     # indicies for the two lists
a[j] = 1
while i < n:
  x[i+j] = 0    # delete number i+j
  while True:   # look for an available number
    i += 1
    if x[i] == 1:   #  number i is available
      break
  j += 1
  a[j] = i   # store i in irregular sequence
print(f"{a[j]/j:.8f}")
plt.scatter(range(j+1), a[:j+1])
plt.xlabel('j')
plt.ylabel('a[j]')
plt.title('Irregular Sieve (n='+str(n)+')')
plt.show()
```

Listing 2.20. Irregular sequence sieve

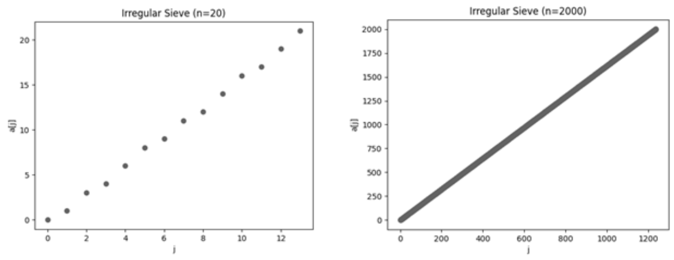


Figure 2.8. Two plots of the irregular sequence for $n = 20$ and $n = 2000$

Our aim is to find a closed formula for a_n. A look at Fig. 2.8 shows that a_n grows almost linearly. It seems likely that $a_n \approx t_n$, where t is some irrational number which accounts for the small but erratic fluctuation about a straight line. To find a good approximation for t we must evaluate a_n/n for large n.

The values for $a[j]/j$ printed by the program are 1.61538462 for $n = 20$ and 1.61762328 when $n = 2000$. This is close to the golden ratio $\frac{1}{2}(1 + \sqrt{5}) =$ 1.61803....

We can convince ourselves that a_n/n approaches the golden ratio by the following argument. The number of integers in the complementary sequence up to $a_n + n$ is n. (Two sequences of natural numbers are *complementary* if together they contain each natural number exactly once.)

The number of numbers in the a sequence up to that point is a_n. So $a_{a_n} \approx a_n + n$. Assuming that $a_n = t_n$, we get $t^2 n \approx tn + n$ and hence $t^2 = t + 1$. This equation is satisfied by the golden ratio.

Also, since a_n/n is always smaller than t, then we can conjecture that $a_n = \lfloor t_n \rfloor$. This can be confirmed by adding the following to the program:

```
isSmallT = True
t = (1+math.sqrt(5))/2
for i in range(j):
  if a[i] != math.trunc(t*i):
    isSmallT = False
print("Smaller than t:", isSmallT)
```

2.5.4 An Olympiad Problem. Do there exist 1983 distinct positive integers $\leq$ 100000 with no three in arithmetic progression? (Problem 5 of the International Mathematical Olympiad XXIV, Paris 1983; available at `https://www.imo-official.org/problems.aspx`.)

To gather some numerical evidence we construct a sequence a_j with no three elements in arithmetic progression by using the following "greedy" algorithm: $a_0 = 0, a_1 = 1, a_j =$ the smallest positive integer, which does not form an arithmetic progression with any two previous terms.

We start with two lists, x and a, and set $a[0] = 0, a[1] = 1, x[i] = 1$ for $i = 2$ to 8000. The terms $a < b < c$ are in arithmetic progression if $2b = a + c$, or $c = 2b - a$. Of the numbers $i = 2$ to 8000 we sieve out those which are in arithmetic progression with any two of the preceding elements $a[0], ..., a[j-1]$. The first i that remains is $a[j]$.

Listing 2.21 (olympiad.py) is the resulting program, and generates the following sequence.

```
N = 8000; M = 132
x = [1]*N
a = [0]*N
a[1] = 1
i = 1; j = 1
while (j < M):
  for k in range(j):
    x[2*i - a[k]] = 0
  i += 1
  while x[i] != 1:
    i += 1
  j += 1
  a[j] = i
for i in range(M):
  print(f'{a[i]:6}',end='')
  if (i+1) % 12 == 0:
    print()
```

Listing 2.21. Olympiad sieving

```
     0     1     3     4     9    10    12    13    27    28    30    31
    36    37    39    40    81    82    84    85    90    91    93    94
   108   109   111   112   117   118   120   121   243   244   246   247
   252   253   255   256   270   271   273   274   279   280   282   283
   324   325   327   328   333   334   336   337   351   352   354   355
   360   361   363   364   729   730   732   733   738   739   741   742
```

```
 756   757   759   760   765   766   768   769   810   811   813   814
 819   820   822   823   837   838   840   841   846   847   849   850
 972   973   975   976   981   982   984   985   999  1000  1002  1003
1008  1009  1011  1012  1053  1054  1056  1057  1062  1063  1065  1066
1080  1081  1083  1084  1089  1090  1092  1093  2187  2188  2190  2191
```

We observe ever longer gaps – between 1 and 3, 4 and 9, 13 and 27, 40 and 81, etc. – and the upper endpoints of these gaps are successive powers of 3: 3, 9, 27, 81, 243, 729, 2187.

Also, the upper endpoint, U, of a gap is one greater than twice its lower endpoint, L, i.e. $U = 2L + 1$. This leads to an alternative algorithm for generating sequences. Beginning with any finite sequence having the required property:

$$a_1, a_2, ..., a_k \tag{2.4}$$

we can add not just the next term, but a block of k terms obtained by translating each term in Equ. (2.4) by $2a_k + 1$ units. The result is a sequence of $2k$ terms

$$a_1, a_2, ..., a_k, a_{k+1} = 2a_k + 1 + a_1, ..., a_{2k} = 2a_k + 1 + a_k \tag{2.5}$$

Exercises for Section 2.5.4

(1) a) Prove that the sequence of $2k$ terms in Equ. 2.5 inherits the desired property from the sequence of k terms in Equ. 2.4.

b) Beginning with the sequence 0, 1, write a program for this algorithm, and check whether it generates the same sequence as our greedy algorithm.

c) Prove that $a_{2^j+1} = 3^j$.

(2) a) Rewrite Listing 2.21 so that it prints the output in base 3, and note that all the numbers in the sequence only use the ternary digits 0 or 1, and that they all appear in increasing order, i.e. 0, 1, 10, 11, 100, 101, 110, ... up to $3^j = 10...0$ (j zeros).

b) Prove that the finite sequence generated by the greedy algorithm consists exactly of the numbers whose ternary digits are 0's and 1 's. (Hint: use induction.)

c) Prove that the sequence generated by the alternative algorithm starting with 0, 1 consists exactly of the numbers whose ternary digits are 0's and 1 's. (Note that b) and c) together prove that your observation in Ex. 1 b) persists for arbitrarily long sequences.)

To answer the question posed in the Olympiad problem, we need only verify that the first $2^{11} = 2048 > 1983$ terms of our sequence are all $\leq 10^5$. The largest of them is $a_{2^{11}}$, the lower endpoint of the gap with upper endpoint 3^{11}. Its value, because of the relation $U = 2L + 1$, or $3^{11} = 2a_{2^{11}} + 1$, is $a_{2^{11}} = (3^{11} - 1)/2 = 88573 < 10^5$. We conclude that our sequence of 2048 integers from 0 to 88573 contains no three in arithmetic progression.

2.5.5 Ulam's Sequence. At the 1963 number theory conference in Boulder, Colorado, Stanislaw Ulam proposed the following sequence $U : u_1 = 1, u_2 = 2, u_n =$ the smallest integer that can be uniquely represented as a sum of two different preceding terms.

This is now known as the standard Ulam sequence, or the (1, 2)-Ulam sequence, since the idea can be generalized to use different starting values (u, v).
There's an obvious, but slow, program for this sequence (see Ex. 2 below). There's also a much faster version which uses a "double sieve" to compute the elements in $U \leq n$. It uses two boolean lists, `lst` and `mst`. For $i > 2$:

- lst[i] = True iff i is representable as a sum in at least one way.
- mst[i] = True iff i is representable as a sum in at most one way.

Listing 2.22 (ulam107.py) implements the algorithm. Several exercises are based on it.

```
n = int(input("n=? "))
lst = [False]*(n+1)
mst = [True]*(n+1)
lst[1] = True; lst[2] = True
k = 1
while k < n:
  k += 1
  while not(lst[k] and mst[k])
      and (k < n):
    k += 1
  m = min(k-1,n-k)
  for i in range(1, m+1):
    mst[k+i] = mst[k+i] and \
      not(lst[k+i] and lst[i]
          and mst[i])
    lst[k+i] = lst[k+i] or (lst
        [i] and mst[i])
for i in range(1,n+1):
  if mst[i]:
    print(f"{i:4}", end='')
```

Listing 2.22. Ulam's sequence

The first few terms are:

```
> python ulam107.py
n=? 100
   1   2   3   4   6   8  11  13  16  18  23  25
  26  28  33  35  36  38  43  45  47  48  53  57
  62  67  69  72  77  82  87  92  94  96  97  99
```

which can be checked at `https://en.wikipedia.org/wiki/Ulam_number`. More information is available in Guy [**Guy04**], section C4. This algorithm is due to M.C. Wunderlich, "The Use of Bit and Byte Manipulation in Computing Summation Sequences", *BIT Numerical Mathematics*, 11 (1971), 217-224, `https://link.springer.com/article/10.1007/BF01934371`, and is discussed at length in `https://math.stackexchange.com/questions/2165222/generating-ulam-sequences-using-bit-manipulation`.

Exercises for Section 2.5

(1) In the central prison of Sikinia there are n cells numbered 1 to n, each occupied by a single prisoner. The state of each cell can be changed from closed

to open and vice versa by a half-turn of the key. To celebrate the Centennial Anniversary of the Republic it was decided to grant a partial amnesty. The president sent an officer to the prison with the instruction

```
for i in range(1,n):
  turn the keys of cells i, 2i, 3i, ....
```

A prisoner was freed if at the end his door was open. Which prisoners are set free? Remark: don't think, just sieve!

(2) a) Write the "obvious" program for the Ulam sequence and compare its speed with Listing 2.22.

The following should be answered using Listing 2.22.

b) In the interval 1..32000 find all solutions of $u_i + u_{i+1} = u_k$.

c) In 1..32000 find all pairs of consecutive numbers which are U-numbers.

d) Among the distances $g_i = u_{i+i} - u_i$, between successive U-numbers in the interval 1..32000, find the largest.

e) What proportion of U-numbers are twins with distance 2?

f) Find the frequency distribution of the distances g_i.

g) Does the asymptotic distribution of U-numbers parallel that of the distribution of primes?

h) What sequence do you get if the two preceding terms need not be different?

(3) Construct a sequence $0 < a_1 < a_2 < a_3 < \ldots$ of integers with the property that each non-negative integer n can be uniquely represented in the form $n = a_i + 2a_j$, where i and j need not be distinct. Collect numerical data, conjecture, prove. This sequence is due to L. Moser, "An Application of Generating Series", *Mathematics Magazine* 35 (1962), 37-38, `https://www.jstor.org/stable/i326617`.

(4) Consider the sequence of the first digit in

a) fib(n) for $n = 1$ to 10000,

b) 2^n for $n = 1$ to 10000,

c) 3^n for $n = 1$ to 10000.

Find the frequencies $x[1], \ldots, x[9]$ of the digits 1 to 9 in each of the sequences in a), b), and c). The result is surprising, but it can be shown that the frequency of the digit n is $\log_{10}(1 + 1/n)$ for $n = 1, \ldots, 9$.

(5) Prove that the density of square-free integers is $p = 6/\pi^2$. Use the code from Listing 2.1.5.

(6) **The Frobenius Problem**. Let $a_1, ..., a_k$ be natural numbers with $gcd(a_1, ..., a_k) = 1$. The natural number N can be represented by $a_1, ..., a_k$ if there are non-negative integers $x_1, ..., x_k$, such that $N = a_1x_1 + ... + a_kx_k$.

It's not hard to show that all sufficiently large numbers can be so represented. The Frobenius Problem is to find the largest number $g(a_1, ..., a_k)$ without such a representation – a problem that has a surprising large number of interpretations. For example, given coins with denominations $a_1, ..., a_k$, find the largest amount which cannot be paid with these coins. The wikipedia article on this topic at `https://en.wikipedia.org/wiki/Coin_problem` is a good source of references.

a) For $k = 2$ there is a simple solution.

b) For $k > 2$ only a few partial results are known. Write a program for $k = 3$, which collects experimental data.

c) Write an efficient program which for any input $a_1, ..., a_k$ finds $g(a_1, ..., a_k)$ (This is a difficult problem, requiring dynamic programming techniques for an efficient solution.)

(7) **The 3n + 1 Problem**. The following algorithm for generating a sequence is quite old, but we still don't know if it always stops:

(a) Start with a natural number n.
(b) If $n = 1$ then stop.
(c) If n is odd then set $n = 3n + 1$.
(d) If n is even then set $n = n \mathbin{//} 2$ and jump to (b)

a) Write a program, which for input n prints the sequence $n, f(n), f(f(n)), ..., 1$, where

$$f(n) = \begin{cases} 3n + 1 & \text{if } n \text{ is odd} \\ n \mathbin{//} 2 & \text{if } n \text{ is even} \end{cases}$$

b) Write a program which for input n counts the number $t(n)$ of terms of the sequence.

c) Print a table $(i, t(i))$ for $i = a, a + 1, ..., b$.

d) For input n print the triple $(n, t(n), max(n))$, where $max(n)$ is the largest value in the sequence.

e) The algorithm can stop only if it encounters a power of 2. Write a function, which for input n prints the exponent of this power of 2.

f) Write a function which for i between a and b prints the largest $t(i)$ and the values of i for which it occurs.

This algorithm, better known as the Collatz conjecture, is one of the most famous unsolved problems in mathematics (`https://en.wikipedia.org/w` `iki/Collatz_conjecture`), and is named after Lothar Collatz who posed it in 1937. Paul Erdös later remarked: "Mathematics may not be ready for such problems."

(8) Suppose you want to test, for all numbers from 2 to b, whether or not the $3n+1$ algorithm stops.

a) Show that you only need to test odd numbers.

b) Show that if you start with n you may stop as soon as you encounter a number smaller than n.

c) Show that you need to test only numbers of the form $4n - 1$.

d) Show that you need not test numbers of the form $16k + 3$ or $128k + 7$.

(9) Let's again consider the $3n + 1$ algorithm, this time focusing on the record-breaking numbers, i.e. those numbers that require more steps to reach 1 than any preceding number. Write a program which prints these numbers, the number of steps to reach 1 and the maximum value reached. A few are listed in Table 2.6.

record breaking number	steps	largest term
2	1	2
3	7	16
6	8	16
7	16	52
9	19	52
18	20	52
25	23	88
27	111	9232
54	112	9232
73	115	9232

Table 2.6. 3n+1 records

(10) **Fibonacci squares**. Are there squares among the Fibonacci numbers? Obviously $fib(1) = fib(2) = 1$ and $fib(12) = 144$ are squares, and also $fib(0) = 0$, but not everybody starts with 0. Are there more squares in the sequence? Our aim is to sieve out the non-squares among the Fibonacci numbers from 1 to 10000. We first set$x[i] = 1$ for i in range(1,10001). These are our potential squares.

The idea is to consider the sequence modulo a prime p. Now any square is also a square $\pmod{p}$. Equivalently, a non-square $\pmod{p}$ is not a square. So we sieve out the non-squares mod 3, 5, 7, 11, 13, ...

The Fibonacci sequence mod 3 is 1, 1, 2, 0, 2, 2, 1, 0, 1, 1, which has a period of 8. The only non-square mod 3 is 2. So we can sieve out all terms with indices $8n + 3$, $8n + 5$, $8n + 6$ by setting the corresponding $x[i] = 0$.

Now consider the sequence mod 5: 1, 1, 2, 3, 0, 3, 3, 1, 4, 0, 4, 4, 3, 2, 0, 2, 2, 4, 1, 0, 1, 1, ..., which has a period of 20. The non-squares mod 5 are 2 and 3. So we set $x[20n + r] = 0$ for $r = 3, 4, 6, 7, 13, 14, 16, 17$.

Continue in this way until all the elements except $x[1], x[2], x[12]$ are zero. A proof of this result can be found online at `https://www.ma.imperial.ac.uk/~buzzard/2017nt2.pdf`, based on J.H.E. Cohn, "Square Fibonacci Numbers", *Fibonacci Quart.* 2 (1964), 109-113.

(11) Explore the sequence

$$f(n) = \begin{cases} 3n - 1 & \text{if } n \text{ is odd} \\ n \,/\!/\, 2 & \text{if } n \text{ is even.} \end{cases}$$

The sequence always seems to enter one of four cycles. Find them and prove this for $n < 10000$. Also check matters with some large values for n.

(12) **Conway's permutation sequences**. A sequence is defined by

$$f(n) = \begin{cases} \lfloor (3n + 1)/4 \rfloor & \text{if } n \text{ is odd} \\ 3n \,/\!/\, 2 & \text{if } n \text{ is even} \end{cases}$$

or, more lucidly, $2m \mapsto 3m$, $4m - 1 \mapsto 3m - 1$, and $4m + 1 \mapsto 3m + 1$, from which it is clear that the operation is invertible. So, if we connect n with $f(n), n = 1, 2, ...$, the resulting graph consists only of disjoint cycles and doubly infinite chains.

Explore some sequences. It is conjectured that there are only four cycles, but there's no conjecture about the number of infinite chains. What is the status of the sequence containing the number 8?

This exercise appears in Guy [**Guy04**], section E17.

(13) Proceeding as in Ex. 10, find by successive sieving the *triangular* Fibonacci numbers below 1000, i.e. Fibonacci numbers of the form $m(m + 1)/2$.

2.6 Rotation of a List

How do you rotate a list v of n elements left by m positions? For instance, with $n = 10$ and $m = 3$ the list 0123456789 should be transformed into 3456789012. Can you do this with little extra storage, which makes sense if m and n are both large?

Of course, you could copy the first m elements into a second list $y[1..m]$, moving the remaining elements left m places and then copy the elements from the temporary list back to v. But what if m is so large that there's not enough space for this approach?

We first consider an easy to understand and quite efficient solution. Rotation means switching two blocks $AB \mapsto BA$. Suppose we have a function reverse(), which reverses a block: $A \mapsto A^R$. Then we form $(A^R B^R)^R$ which becomes BA. For our example with $n = 10$, $m = 3$ we get

$$\begin{aligned} 0123456789 &\mapsto 2109876543 \\ &\mapsto 3456789012. \end{aligned}$$

So we call reverse(a, b) (reverse the part of the list from the a-th to the $(b-1)$-th elements) three times:

```
reverse(0, m): 210 3456789
reverse(m, n): 210 9876543
reverse(0, n): 3456789 012.
```

Listing 2.23 (rotate.py) implements this idea.

```
def reverse(l,r):
  i = l; j = r-1
  while j-i > 0:
    v[i], v[j] = v[j], v[i]
    i += 1
    j -= 1

m,n = map(int, input("m n=? ").
    split())
v = [i for i in range(n)]
print("Original:", v)
reverse(0, m)
reverse(m, n)
reverse(0, n)
print(" Rotated:", v)
```

Listing 2.23. Rotation by reversals

Switching adjacent blocks by means of reversals is part of computing folklore; for example, see Gries [**Gri81**] and Bentley [**Ben00**].

There's a different algorithm which may be somewhat more efficient if $gcd(m, n) = 1$ or some small number. The **dolphin** algorithm consists of shifting list elements into their new places one by one, with the next move being to fill the space that was just vacated.

First we look at the case $m = 3$, $n = 10$, with $gcd(m, n) = 1$. We save a copy of $v[0]$ in x. Now we shift $v[3]$ to its final position $v[0]$ leaving $v[3]$ free to be occupied by $v[6]$, etc.:

$$\begin{aligned} x \leftarrow v[0] \leftarrow v[3] \leftarrow v[6] \leftarrow v[9] \leftarrow v[2] \leftarrow v[5] \\ \leftarrow v[8] \leftarrow v[1] \leftarrow v[4] \leftarrow v[7] \leftarrow x. \end{aligned}$$

Each move uses $v[i] = v[(i + 3)\%10]$ or $v[i] = v[(i + m)\%n]$ and since $gcd(m, n) = 1$ all the shifts can be achieved in a single pass which ends with the original x being copied into $v[7]$.

Consider another example. This time $m = 6$, $n = 15$ with $gcd(m, n) = 3$.

Each move uses $v[i] = v[(i+6)\%15]$ or $v[i] = v[(i+m)\%n]$ and since $gcd(m,n) = 3$ this will require three passes over the list:

$$x \leftarrow v[0] \leftarrow v[6] \leftarrow v[12] \leftarrow v[3] \leftarrow v[9] \leftarrow x$$
$$x \leftarrow v[1] \leftarrow v[7] \leftarrow v[13] \leftarrow v[4] \leftarrow v[10] \leftarrow x$$
$$x \leftarrow v[2] \leftarrow v[8] \leftarrow v[14] \leftarrow v[5] \leftarrow v[11] \leftarrow x.$$

The corresponding Python program is shown in Listing 2.24 (dolphin.py).

```
m,n = map(int, input("m n
     =? ").split())
v = [i for i in range(n)]
print("Original:", v)
d = gcd(m, n)
print("No. of passes:", d)
for i in range(d):
  j = i; x = v[i]
  while True:
    k = j; j = (j+m)%n
    v[k] = v[j]
    if j == i: # at start
      break
  v[k] = x
print(" Rotated:", v)
```

Listing 2.24. Rotation by shifting

2.7 Partitions

Let $p(n)$ be the number of different partitions of n into parts. For instance,

$$5 = 4+1 = 3+2 = 3+1+1 = 2+2+1 = 2+1+1+1 = 1+1+1+1+1.$$

Thus $p(5) = 7$. In addition let $f(m,n)$ be the number of partitions of m with each part at most n. Then p(n) = $f(n,n)$. $f(m,n)$ can be defined recursively as:

$$f(1,n) = f(m,1) = 1 \text{ for all } m,n; \tag{2.6}$$
$$f(m,n) = f(m,m) \text{ for } m \le n; \tag{2.7}$$
$$f(m,m) = 1 + f(m,m-1); \tag{2.8}$$
$$f(m,n) = f(m,n-1) + f(m-n,n) \text{ for } n < m. \tag{2.9}$$

We can combine Equs. (2.7) and (2.8) into

$$f(m,n) = 1 + f(m,m-1) \text{ for } m \le n.$$

Thus, we get the recursive $f()$ in Listing 2.25 (parts.py).

```
def f(m,n):
  if (m == 1) or (n == 1):
    return 1
  elif m <= n:
    return 1 + f(m,m-1)
  else:
    return f(m,n-1) + f(m-n,n)

n = int(input("n=? "))
print(f"p({n}) = {f(n,n)}")
```

Listing 2.25. Recursive partition counting

Note that the code assumes that $n = m$, and so the user only needs to supply a single value.

Typical output:

```
> python parts.py
n=? 20
p(20) = 627

> python parts.py
n=? 39
p(39) = 31185
```

The drawback of this approach is that the recursive computation time grows exponentially with n, and so even medium size inputs (e.g. $n = 100$) take too long to return.
After some deliberation we succeeded in developing a fast iterative algorithm.

The first step is to set $f(0, n) = f(0, 0) = 1$. Then Equ. (2.9), $f(m, n) = f(m, n-1) + f(m-n, n)$, becomes valid for $n = m$ since the second term on the right-hand side becomes $f(m-m, n) = f(0, n) = 1$. This makes Equ. (2.9) become $f(m, n) = f(m, m-1)+1$, which is equivalent to the combined equation for (2.7) and (2.8) above.

$f(m, n)$ can now be written as just two equations:

$$f(0, n) = 1 \text{ for all } n;$$
$$f(m, n) = f(m, n-1) + f(m-n, n) \text{ for } n \leq m.$$

A standard way to implement a two-argument function like $f(m, n)$ is to use a 2D list (e.g. $p[m][n]$). This has the advantage that once a particular $f(m, n)$ value has been calculated then it no longer needs to be reevaluated (i.e. memoization). Also, a list is usually easier to iterate over rather than employing recursion.

If we study how the 2D list is filled in this problem, it becomes clear that the $f(m, n-1)$ value in $f(m, n) = f(m, n-1) + f(m-n, n)$ is never needed again after this calculation, and so the $f(m, n)$ value could be stored in $p[m][n-1]$.

More drastically, there's actually no need to store the $f()$ values in a 2D list at all; a 1D list is enough. This means that $f(m, n) = f(m, n-1) + f(m-n, n)$ can be implemented as $p[m] = p[m] + p[m-n]$. Note, that this still requires us to have two loops, one to iterate over n, and the other for m.

The resulting code is in Listing 2.26 (partiter.py).

```
n = int(input("n=? "))
p = [1]*(n+1)    # so p[n] can store a value
for i in range(2,n+1):
  for j in range(i,n+1):
    p[j] += p[j-i]
print(f"p({n}) = {p[n]}")
```

Listing 2.26. Iterative partition counting

We again assume that $n = m$, and so the user only needs to supply a single value. It also means that both loops use $n + 1$.

Some results:

```
p(127) =   3913864295                p(159) =  97662728555
p(138) =  12292341831                p(160) = 107438159466
p(149) =  37027355200
```

Srinivasa Ramanujan proved the congruences

$$\begin{aligned} p(5m+4) &\equiv 0 \pmod{5}, \\ p(7m+5) &\equiv 0 \pmod{7}, \\ p(11m+6) &\equiv 0 \pmod{11}. \end{aligned}$$

Of the above values of n, 159 is of the form $7m + 5$ and the others have the form $11m + 6$. So $p(159)$ should be divisible by 7 and the other values should be divisible by 11. This is indeed the case.

A good starting point for more details on the partition function and Ramanujan's involvement is `https://en.wikipedia.org/wiki/Partition_function_(number_theory)`.

2.8 The Money Changing Problem

In how many different ways can you sum 100 cents using 1 cent coins, nickels, dimes, quarters and half-dollars (worth 1, 5, 10, 25, and 50 cents, respectively)? Let's generalize slightly and ask: in how many ways can you make n cents with US coins? Other countries have other coins, so generalize again:

> A country has coins with denominations $D = \{1, d_1, d_2, ..., d_{k-1}\}$ units. In how many ways can you make a sum of n units ?

We make sure to set $d_0 = 1$ so we can sum any amount. Let $a(n, k)$ be the number of ways to sum n units using coins from D. Then we have:

$$\begin{aligned} a(n,0) &= 1, \\ a(n,k) &= 0 \text{ for } n < 0, \\ a(n,k) &= a(n,k-1) + a(n-d_k,k). \end{aligned}$$

In words, the last equation states that a sum n can either use the coin d_k or not. There are $a(n, k-1)$ ways to make n without using d_k and $a(n-d_k, k)$ ways when using d_k.

$a(n, k)$ is implemented recursively in Listing 2.27 (change.py).

```
d = [1, 5, 10, 25, 50]    # d_0 = 1, ...

def a(n, k):
  if n < 0:
    return 0
  elif k == 0:  # d_0 = 1
    return 1
  else:
    return a(n,k-1) + a(n-d[k],k)
```

Listing 2.27. Recursive change sum

For US coins we get the results shown in Table 2.7.

n	100	200	300	400	500
a(n,k)	292	2435	9590	26517	59576

Table 2.7. Ways of summing n using US coins

We've implemented a tree recursive process, which will become exponential slow as n increases (or crashes at Python's recursion limit). Let's construct an iterative version to deal with larger n. The direct translation of Listing 2.27 would use a 2D list $a[n][k]$, but a study of how the data is built shows a similar optimization is possible as in the previous section's partitioning problem. Instead of a 2D list, we can keep overwriting a 1D list. We get Listing 2.28 (change2.py).

```
d = [1, 5, 10, 25, 50] #d_0=1
n = int(input("n? "))
print("Coins:", d)
k = len(d)
a = [1]*(n+1)
for j in range(1, k):
  for i in range(d[j], n+1):
    a[i] += a[i-d[j]]
print("Ways to pay:", a[n])
```

Listing 2.28. Iterative change sum

Listing 2.28 computes the number of ways (432699251)to sum 50000 (i.e. $500) in a fraction of a second whereas Listing 2.27 raises an error when Python's recursion limit is reached.

2.9 An Abstractly Defined Set

A set S is defined by means of three axioms:

A1. $0 \in S$
A2. $x \in S \rightarrow 2x + 1 \in S$ and $3x + 1 \in S$
A3. S is the minimal set satisfying axioms A1 and A2.

We'll use these as the basis of a recursive boolean function which recognizes if a number n belongs to S. Which of the numbers 511, 994, 995, 996, 997, 998, 999 do belong? The trees in Fig. 2.9 show that only 511, 994, and 999 are in S, since these numbers construct a path leading to 0 (and T for True).

Note that this search is the 'opposite' of the mathematical definition. The axioms builds a set starting from $x = 0$ by adding $2x + 1$ and $3x + 1$ recursively, while the search starts with the final number and attempts to reduce it to 0.

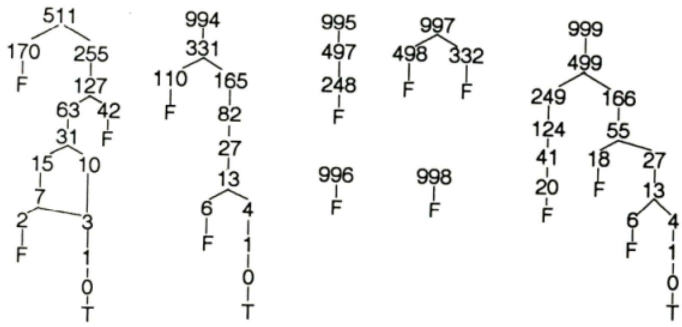


Figure 2.9. Decide if a number belongs in S

inS() in Listing 2.29 (inset.py) uses recursion to implement the tree search shown in Fig. 2.9

```
def inS(x):
  if x == 0:
    return True
  x -= 1
  if (x % 2 != 0) and (x % 3 != 0):
    return False      # nothing divisible
  elif x % 3 != 0:    # one is dvisible
    return inS(x//2)
  elif x % 2 != 0:
    return inS(x//3)
  else:               # both are divisible
    return inS(x//2) or inS(x//3)
```

Listing 2.29. Recursively check if a number is in S

In this case, there's little chance of Python's recursion limit being hit since the number of calls is proportional to a tree's longest branch, which is $O(\log_2 n)$ when it's checking $2n + 1$ reductions.

The iterative version in Listing 2.30 (insetIt.py) is much closer to the axiomatic definition.

```
n = int(input("n=? "))
s = {0}
for i in range((n+1)//2):
  if i in s:
    s.add(2*i+1)
    s.add(3*i+1)
print("in S:", n in s)
```

Listing 2.30. Iteratively check if a number is in S

It builds a Python set just big enough so that it can check if the user-supplied number is a member. The trick is to stop adding to the set once $i = (n+1) \,/\!/\, 2$ which ensures that n will be in the set if it's valid.
This approach illustrates the trade-off between recursion and data structures. This example replaces recursive calls by set building.

2.10 Recursive Pile Splitting

A single pile of n chips, $n > 1$, is randomly split into two piles of k and $n-k$ chips, and the product $k(n-k)$ is computed. Then each pile, provided it has more than one chip, is randomly split again, and the product of the number of chips in its parts is added to the preceding product. This splitting is repeated on all the piles until every one consists of just a single chip (i.e. there are n piles).
A recursive function to find the final sum of the $n-1$ products is given in Listing 2.31 (sumprod.py).

```
def f(n):
  if n < 3:
    return n-1
  else:
    k = random.randint(1,n-1)
    return k*(n-k) + f(k) + f(n
        -k)
```

Listing 2.31. Recursive piles splitting

Table 2.8 shows the final sum when the pile starts with different sizes. Surprisingly, the sum doesn't depend on how the piles are split.

n	2	3	4	5	6	7	8	9	10
f(n)	1	3	6	10	15	21	28	36	45

Table 2.8. Sum of $n-1$ splits

Based on an examination of Table 2.8, we guess that $f(n) = \binom{n}{2}$. For a proof, imagine tying strings from each chip to every other chip. Initially there will be $n(n-1)/2$ strings. After splitting a pile into piles X, Y with x and y chips, we cut all the strings connecting those two piles, which reduces the total by xy strings. This will continue until there are n piles of chips with all the strings cut. This will always total $n(n-1)/2$, which is also the total sum of the xy products, no matter how we split the piles.

2.11 The Stern-Brocot Sequence

The online reference for this sequence is `https://oeis.org/A002487`. (It was formerly labeled N0056 in the 1973 edition of Sloane's *Handbook of Integer Sequences*.)

We define a function $f()$ on the non-negative integers by means of the recursion

$$f(0) = 0, \quad f(1) = 1, \quad f(2n) = f(n), \text{ and}$$
$$f(2n+1) = f(n) + f(n+1) \text{ for all } n > 0.$$

The Stern-Brocot sequence (or Stern's diatomic series) are the values $f(0), f(1), f(2), ...$ Incidentally, this function is also known by the name fusc(), given to it by Edgser Dijkstra in the 1970s. Let

$$n = 2^s + e_{s-1}2^{s-1} + ... + e_0$$

be the binary representation of n. If we reverse its digits (e.g. 10100 $\mapsto$ 00101), we get

$$u = e_0 2^s + e_1 2^{s-1} + ... + e_{s_1} 2 + 1.$$

It's fairly difficult to prove that $f(n) = f(u)$ for all n, but quite easy to write a program that calculates $f(n)$ and $f(u)$ and checks for equality.
$f()$ is tree recursive but the depth of the tree is only about $O(\log_2 n)$ because the recursive calls are dividing n by 2. Nevertheless, an iterative version will be faster for very large numbers (reverse1.py).

```
def f(n):
  if n < 2:
    return n
  elif (n % 2 == 0):   # even
    return f(n//2)
  else:
    return f((n-1)//2) + f((n
        +1)//2)

n = int(input("n=? "))
fn = f(n)
print(f"n: {n}; f({n}): {fn}")
u = 0
while n > 0:
  u = u*2 + n%2
  n = n//2
fu = f(u)
print(f"u: {u}; f({u}): {fu}")
print("Equal?", fn == fu)
```

Listing 2.32. Reversing using f()

To better understand the transformation, let's hand-compute one call to $f()$ in detail:

$$\begin{aligned} f(85) &= f(42) + f(43) &&= f(21) + f(21) + f(22) \\ &= 2 * f(21) + f(22) &&= 2 * f(10) + 3 * f(11) \\ &= 5 * f(5) + 3 * f(6) &&= 5 * f(2) + 8 * f(3) \\ &= 13 * f(1) + 8 * f(3) &&= 13 * f(0) + 21 * f(1) = 21. \end{aligned}$$

The function g() will be useful:

$$g(n, i, j) = i * f(n) + j * f(n+1). \tag{2.10}$$

Then we have

$$\begin{aligned} g(2n, i, j) &= i * f(2n) + j * f(2n+1) \\ &= (i+j) * f(n) + j * f(n+1) = g(n, i+j, j), \\ g(2n+1, i, j) &= i * f(2n+1) + j * f(2n+2) \\ &= i * f(n) + (i+j) * f(n+1) = g(n, i, i+j). \end{aligned}$$

This means that g() can be defined using two recursions:

$$\begin{aligned} g(2n, i, j) &= g(n, i+j, j) \\ g(2n+1, i, j) &= g(n, i, i+j). \end{aligned}$$

Both of these reduce the n value, so eventually we'll reach g(0, i, j). The definition for g() in Equ. (2.10) means that:

$$\begin{aligned} g(n, i, j) &= i * f(n) + j * f(n+1) \\ g(0, i, j) &= i * f(0) + j * f(1) = j. \end{aligned}$$

Equ. 2.10 can also be used to express g() in terms of a single $f()$ call, by setting $i = 1$, and $j = 0$:

$$\begin{aligned} g(n, i, j) &= i * f(n) + j * f(n+1) \\ g(n, 1, 0) &= 1 * f(n) + 0 * f(n-1) \\ &= f(n). \end{aligned}$$

This lets us replace $f()$ by g() which is still recursive, but tail-recursive. The resulting code is in Listing 2.33 (reverse2.py). Another step is to translate the tail-recursion into a loop, as in Listing 2.34 (reverse3.py).

```
def g(n,i,j):
  if n == 0: return j
  elif n/2 != int(n/2):
    return g((n-1)/2, i, i+j)
  else:
    return g(n/2, i+j, j)

n = int(input("n=? "))
gn = g(n,1,0); u = 0
print(f"n: {n}; g({n},1,0): {gn}")
while n > 0:
  u = u*2 + n%2; n = n//2
gu = g(u,1,0)
print(f"u: {u}; g({u},1,0): {gu}")
```

Listing 2.33. Reversing using g()

```
def g(n,i,j):
  while n > 0:
    if n/2 != int(n/2):
      n, i, j = (n-1)/2, i, i+j
    else:
      n, i, j = n/2, i+j, j
  return j
```

Listing 2.34. Iterative version of g()

2.11.0.1 The Calkin–Wilf Tree. The $f(0), f(1), f(2), \ldots$ values of the Stern-Brocot sequence have the unexpected feature that $f(n)/f(n+1)$ runs through all the reduced non-negative rationals exactly once.

The start of the sequence is printed out by sbSeq.py which repeatedly calls $f()$, and uses Python's fraction class to print $f(n)/f(n+1)$

```
> python sbSeq.py
n=? 40
  0  1  1  2  1  3  2  3  1  4  3  5  2  5  3  4  1  5  4  7
  3  8  5  7  2  7  5  8  3  7  4  5  1  6  5  9  4 11  7 10
0
1
1/2 2
1/3 3/2 2/3 3
1/4 4/3 3/5 5/2 2/5 5/3 3/4 4
1/5 5/4 4/7 7/3 3/8 8/5 5/7 7/2 2/7 7/5 5/8 8/3 3/7 7/4 4/5 5
1/6 6/5 5/9 9/4 4/11 11/7 7/10
```

These rationals can be arranged into an infinite complete binary tree called a Calkin–Wilf tree tree, as in Fig. 2.10.

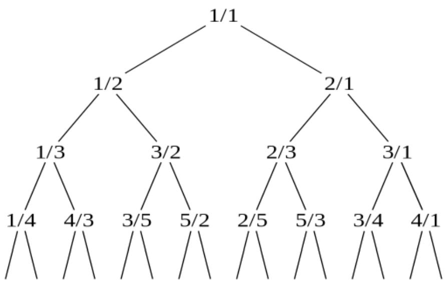


Figure 2.10. The first few levels of the Calkin-Wilf tree

The tree is rooted at the number 1, and any rational number expressed in simplest terms as the fraction a/b has as its two children the numbers $a/(a+b)$ and $(a+b)/b$. It is named after Neil Calkin and Herbert Wilf, who considered it in 'Recounting the Rationals', in the *The American Mathematical Monthly*, Vol. 107, No. 4, April 2000, online at `https://www.maths.ed.ac.uk/~v1ranick/papers/calkinwilf.pdf`.

The sequence of rationals obtained by a breadth-first traversal of the tree is known as the Calkin–Wilf sequence. Listing 2.35 (calkinwilf.py) prints out the requested n-th number in the sequence.

```
def nCalkinWilf(n):
  queue =[(1,1)]
  n -= 1
  for _ in range(n):
    numer, denom = queue.pop(0)
    leftNumer = numer; leftDenom = numer + denom
    rightNumer = numer + denom; rightDenom = denom
    queue.append((leftNumer, leftDenom))
    queue.append((rightNumer, rightDenom))
  finalNumer, finalDenom = queue.pop(0)
  if finalDenom == 1:
    return finalNumer
  else:
    return str(finalNumer) + '/' + str(finalDenom)
```

Listing 2.35. Generating the n-th value in the Calkin-Wilf sequence

For example:

```
> python calkinwilf.py
n? 5
3/2
> python calkinwilf.py
n? 12
2/5
```

The code doesn't build a tree, instead using a queue to implement a breadth-first traversal over the sequence.

2.11.0.2 Relationship to the Stern–Brocot tree. The Calkin–Wilf tree resembles the Stern–Brocot tree (see Fig. 2.11), especially at the top levels, and the same numbers appear at the same levels in both trees.

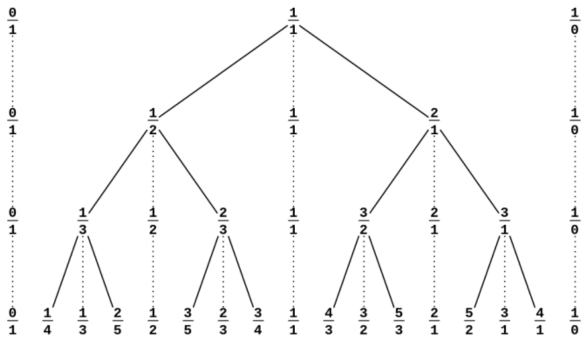


Figure 2.11. The first few levels of the Stern-Brocot tree

A crucial (and useful) difference is that Stern–Brocot is a binary search tree: its left-to-right traversal is the same as the numerical order of the numbers in it. This property is not true of Calkin–Wilf.

The tree's construction begins with the two fractions $\frac{0}{1}$, $\frac{1}{0}$ although the second isn't strictly a fraction, but think of it as representing ∞. At every subsequent level, the existing fractions are copied down (that's the meaning of the dotted lines in Fig. 2.11), and the *mediant* of adjacent fractions are inserted. For two fractions $\frac{a}{b}$ and $\frac{c}{d}$ their mediant is $\frac{a+c}{b+d}$.

Every fraction in the tree has two children. Each child is the mediant of the closest ancestor on the left and closest ancestor on the right.

One approach for building a Stern-Brocot tree is to convert an existing Calkin-Wilf tree by performing a bit-reversal permutation on the numbers at each level of the trees. However, an easier approach is to maintain a list for the current level of the Stern-Brocot tree, and keep expanding it with mediants until the desired level is reached. That is implemented in Listing 2.36 (sbTree.py). Typical output:

```
> python sbTree.py
lvls=? 3
(0, 1) (1, 3) (1, 2) (2, 3)
(1, 1) (3, 2) (2, 1) (3, 1)
(1, 0)
```

```
def expand(fracs):
  nFracs = []; i = 0
  while i < len(fracs)-1:
    nFracs.append( fracs[i])
    nFracs.append( mediant(
        fracs[i], fracs[i+1]))
    i += 1
  nFracs.append( fracs[-1])
  return nFracs

def mediant(f1, f2):
  return (f1[0] + f2[0],
          f1[1] + f2[1])

lvls = int(input("lvls=? "))
fracs = [ (0, 1), (1, 0) ]
for i in range(lvls):
  fracs = expand(fracs)
print(*fracs)
```

Listing 2.36. A Stern–Brocot tree level

An important aspect of the tree is that the mediant of two fractions is always a value between those fractions:

$$\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$$

given that

$$\frac{a}{b} \leq \frac{c}{d}.$$

The two inequalities can be shown by rewriting the fractions with common denominators.

This means that the tree can be used to find increasingly accurate rational approximations to decimals (see Listing 2.37; RationalApprox.py).

```
def mediant(f1, f2):
  return (f1[0] + f2[0],
          f1[1] + f2[1])

x = float(input("x=? "))
left  = (0, 1); right = (1, 0)
best = left
bestError = abs(x)
while bestError > EPS:
  med = mediant(left, right)
  if x < toFloat(med):
    right = med    # go left
  else:
    left = med     # go right
  error = abs(toFloat(med) - x)
  if error < bestError:
    best = med; bestError = error
    print(best, end=' ')
print()
```

Listing 2.37. Finding rational approximations

Approximations for π and e:

```
> python RationalApprox.py
x=? 3.14159265358979323846264338328
(1, 1) (2, 1) (3, 1) (13, 4)
(16, 5) (19, 6) (22, 7) (179, 57)
(201, 64) (223, 71) (245, 78)
(267, 85) (289, 92) (311, 99)
(333, 106) (355, 113)

> python RationalApprox.py
x=? 2.71828182845904523536028747135
(1, 1) (2, 1) (3, 1) (5, 2)
(8, 3) (11, 4) (19, 7) (49, 18)
(68, 25) (87, 32) (106, 39) (193, 71)
(685, 252) (878, 323) (1071, 394)
```

For more on the Stern–Brocot tree see Section 4.5 of Graham [**GKP94**] and `https://en.wikipedia.org/wiki/Stern%E2%80%93Brocot_tree`.

2.12 The Best Rational Approximation

Given a real number $r > 0$, we want to find a sequence of ever better rational approximations p/q with $q \leq qmax$. Listing 2.38 (ratap.py) is probably the simplest implementation.
We measure the distance between r and p/q with $d = |r - p/q|$, with $d = r$ initially. Each time that we get a smaller distance, the associated fraction p/q is printed.

```
r,qmax = map(float, input("r
    qmax=? ").split())
p = 0; q = 1
min = r
while q < qmax:
  if p/q < r:
    p += 1
  else:
    q += 1
  d = abs(r - p/q)
  if d < min:
    min = d
    print(p, "/", q)
  if (d == 0):
    break
```

Listing 2.38. Best rational approxs

If Listing 2.38 is run with r set to one of $\tau = (1+\sqrt{5})/2 \approx 1.6180339887498948432$, $\sqrt{2} \approx 1.4142135623730955049$, or $\pi \approx 3.14159265358979323 8$, we get the approximations listed in Table 2.9.

τ	$\sqrt{2}$	π
1/1	1/1	1/1
2/1	3/2	2/1
3/2	4/3	3/1
5/3	7/5	13/4
8/5	17/12	16/5
13/8	24/17	19/6
21/13	41/29	22/7
34/21	99/70	179/57
55/34	140/99	201/64
89/55	239/169	223/71
144/89	577/408	245/78
233/144	816/577	267/85
377/233	1393/985	289/92
610/377	3363/2375	311/99
987/610	4756/3363	333/106
1597/987	8119/5741	355/113
2584/1597	19601/13860	52163/16604
4181/2584	27720/19601	52518/16717

Table 2.9. Rational approximations for $\tau, \sqrt{2}, \pi$.

There are some interesting patterns hidden inside these rationals. For τ we get pairs of successive Fibonacci numbers. For $\sqrt{2}$ the p's and q's are the solutions to $p^2 - 2q^2 = \pm 1$ and $p^2 - 2q^2 = 2$. The π approximations include the well-known 22/7 and 355/113. More importantly, the sequence of approximations is exactly the same as that produced by Listing 2.37. Is this also the case case for the sequences generated by Listing 2.37 when given values for τ and $\sqrt{2}$? The answer appears to be 'yes', but why?

We'll return to this problem when we look at continued fractions in Sec. 11.

Exercises for Sections 2.6 to 2.12

(1) Here's yet another method for rotating an array. Initially we have $V = AB$, which must be transformed into $V = BA$. Suppose that the block A is shorter than B, then we can write $B = B_0B_1$ with block B_1 being the same length as A. Blocks of equal length are easy to swap and we get $V = B_1B_0A$. Now A is in its final position and we must transform B_1B_0 into B_0B_1, but this is the same task as before but with shorter blocks B_1, B_0.

The opposite case where block B is shorter than A is treated in a similar fashion.

Write a program based on this idea.

(2) Let $b(0) = 1$ and for $n \geq 1$ let $b(n)$ denote the number of ways of writing n as a sum of powers of 2 (order does not matter). Thus

$$7 = 4+2+1 = 4+1+1+1 = 2+2+2+1 = 2+2+1+1+1$$
$$= 2+1+1+1+1+1 = 1+1+1+1+1+1+1$$

and so $b(7) = 6$. Show that

$b(2n+1) = b(2n), \qquad b(2n) = b(2n-2) + b(n).$

a) Write a recursive function to compute $b(n)$.

b) Write an iterative function to compute $b(n)$. For example:

$$b(132) = 31196.$$

(3) Let $r(n, m)$ be the number of partitions of n into parts, the smallest being at least m. We define $r(0, m) = 1$. Obviously $p(n) = r(n, 1)$, with $p(n)$ defined as in Sec. 2.7.

a) Show that $r(n, m) = r(n, m+1) + r(n-m, m)$

b) Write a recursive function which computes $p(n)$.

c) Write an iterative function which computes $p(n)$.

(4) Let the number of partitions of n into parts $\leq m$ be denoted by $p(n, m)$. Prove that $p(n, m) = p(n, m-1) + p(n-m, m)$ (Euler). Find boundary conditions and write a recursive function which computes p. Compare its speed with the one in Sec. 2.7.

(5) Let $p(u, n, m)$ be the number of partitions of u into n parts each $\leq m$. Show that $p(u+1, n+1, m) = p(u, n+1, m) + p(u, n, m) - p(u-m, n, m)$ (Euler). Find boundary conditions and write a function to compute p.

(6) Prior to 2002, German coins consisted of D = {1, 2, 5, 10, 50, 100, 200, 500} Pfennigs. In how many ways could you change

a) 1 DM into smaller coins? (1 DM = 1 German Mark = 100 Pfennigs.)

b) 5 DM using all the coins at least once?

(7) Russian coins consist of D = {1, 2, 3, 5, 10, 15, 20} kopeks. In how many ways can you change 1 Ruble (= 100 kopeks) into coins?

(8) Swiss coins less than 1 SF (Swiss Franc) consists of D = {1, 2, 5, 10, 20, 50} Rappen. In how many ways can you change 1 SF into smaller coins? (1 SF = 100 Rappen.)

(9) In how many ways can you change 1 English Pound into coins? A few years ago English coins were D = {0.5, 1, 2, 5, 10, 20, 50, 100} pence. How do you handle 0.5 pence? (The half-penny is no longer in circulation, and the Pound is 100 pence.)

(10) You have a set $a_1 < a_2 < ... < a_s$, of integer weights. In how many different ways can the weight n be composed from these weights? That is, write a program which finds the number C_n of solutions of

$$a_1e_1 + a_2e_2 + \cdots + a_se_s = n, \quad e_i \in \{0, 1\}.$$

(11) Solve the preceding problem if the weights can be placed on *both* pans of the scales. That is, write a program which finds the number D_n of solutions of

$$a_1e_1 + a_2e_2 + \cdots + a_se_s = n, \quad e_i \in \{-1, 0, 1\}.$$

In Exs. 10 and 11, experiment with different sets of weights, especially the set $\{1, 2, 2^2, ..., 2^{s-1}\}$ in 10, and the set $\{1, 3, 3^2, ..., 3^{s-1}\}$ in 11.

(12) A set S is defined by means of three axioms:

A1. $1 \in S$
A2. $x \in S \rightarrow 2x \in S, \quad 3x \in S, \quad 5x \in S.$
A3. S is the minimal set satisfying axioms A1 and A2.

a) Write a boolean function, which recognizes if $n \in S$.

b) Show that S contains all elements of the form $2^a 3^b 5^c$ with natural numbers a, b, c.

c) Write a program which prints the elements of S in order.

(13) There is a beautiful geometrical solution for the result in Sec. 2.10. Find out about it. As a hint, here's the accompanying 'Proof by picture' which illustrates a proof by induction.

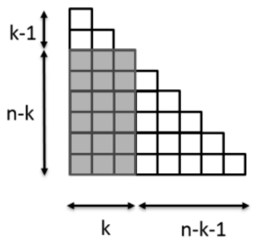


Figure 2.12. Dividing a pile of n chips

(14) The set S is the minimal set satisfying the two axioms:

A1. $0 \in S$
A2. $x \in S \rightarrow 2x + 1 \in S, \quad 3x + 1 \in S, \quad 5x + 1 \in S.$

Write recursive and iterative programs which print and count all elements of $S \leq n$.

(15) Does the set $\{1, 2, ..., 3000\}$ contain a subset A of 2000 elements such that $x \in A$ implies $2x \notin A$? (Posed at the 10th Austrian-Polish Mathematics Competition (APMC) 1987; available at `https://imomath.com/index.cgi?page=collectionApo` and Kuczma [**Kuc94**].)

This problem was generalized by E.T.H. Wang in "On double-free sets of integers", *Ars Combinatoria*, 28, 1989, pp. 97-100 in the following way: a set S of integers is called double-free (D.F.) if $x \in S \rightarrow 2x \notin S$. Let $N_n = \{1,2, ...,n\}$ and $f(n) = \max\{|A| : A \subset N_n \text{ is D.F.}\}$.

Wang proved that $f(n) = \lceil n/2 \rceil + f(\lfloor n/4 \rfloor)$. Write a recursive program for $f()$ and solve the original problem.

Problems 16 to 19 refer to Sec. 2.11.

(16) Write a program which finds the three smallest n such that $f(n) = x$ for $x = 8, 10, 20, 30$.

(17) Write an iterative program for $f(n)$. Hint: suppose we have values a, b such that $f(n) = af(m) + bf(m + 1)$. Then we can find a similar formula $f(n) = a'f(m') + b'f(m' + 1)$ with $m' = \lfloor m/2 \rfloor$ as follows. If m is even then $a' = a + b$, $b' = b$; if m is odd then $a' = a$, $b' = a+b$. We start with $m = n, a = 1, b = 0$ and when we reach $m = 0$ we have $f(n) = b$.

(18) A function f defined on the positive integers is given by

$$f(1) = 1, \quad f(3) = 3, \quad f(2n) = f(n),$$
$$f(4n + 1) = 2f(2n + 1) - f(n), \quad f(4n + 3) = 3f(2n + 1) - 2f(n)$$

Determine the number of positive integers ≤ 1988 for which $f(n) = n$. This was a difficult problem in the International Mathematical Olympiad XXIX (`https://www.imo-official.org/problems.aspx`), but is a fairly routine coding task.

a) Write a recursive function which computes $f(n)$.

b) Write a program which solves the problem by brute force.

c) Compute a table of $f(n)$ and guess what the function does.

d) Write a program which for input n finds the binary representations of n and $f(n)$. What does the function do?

e) Find the number of positive integers $\leq 2^k$ for which $f(n) = n$.

f) Prove that f reverses the binary representation of an odd integer.

g) Iteratively compute the array $f[k]$ for every $k \in 1..n$.

(19) Show that in Ex. 17 $f(n) = f(u)$. Hint: analyze Ex. 17 or 18f).

(20) a) Using Listing 2.29 find all set elements ≤ 10000.

b) What proportion of consecutive numbers have differences of 1?

c) What proportion of consecutive numbers have differences of 2?.

d) Find the number of elements in $1..n$ for $n = 1000, 2000, ..., 10000$. Observe the strange fluctuations in these ten "millenia". Does this behavior persist?

(21) In Sec. 2.6 we learned how to swap the blocks AB into BA. How would you transform ABC into CBA? That is, two nonadjacent blocks A, C must be swapped.

An efficient algorithm can be found in J .L Mohammed and C.S. Subi, "An Improved Block-Interchange Algorithm", *Journal of Algorithms* 8 (1987), 113-121.

(22) **The Frobenius Problem revisited**. Given coins with denominations $a[1], ..., a[s]$. Let $b[k]$ be the number of solutions of the equation $a[1]x_1 + ... + a[s]x_s = k$, where x_i are non-negative integers. Write a program which stores $b[1], ..., b[n]$ in a list b.

(23) Repeat Ex. 16 from "Exercises for Sections 1.1 to 1.7" in a similar way to the preceding problem.

(24) Run Listing 2.38 with r set to:

a) $\sqrt{3} \approx l.7320508076$
b) $\sqrt{7} \approx 2.2360679775$
c) $\sqrt{13} \approx 2.6457513110$
d) $\sqrt{5} \approx 3.1622776602$
e) $\sqrt{10} \approx 3.6055512754$
f) $e \approx 2.7182818285$

For each of these, also generate the sequence of approximations using Listing 2.37 to check if the two sequences are the same.

2.13 Primes

In this section we will deal with primes and their distribution.

2.13.1 How to Recognize a Prime. A natural number is a prime if it has exactly two divisors. If $n > 1$ is not a prime it can be decomposed into non-trivial factors

$$n = d_1 d_2, \quad 1 < d_1 < n, \quad 1 < d_2 < n.$$

The divisors d_1,d_2 cannot both be $> \sqrt{n}$, otherwise $d_1 d_2 >$ n. That is, a composite number n has non-trivial divisors $d \leq \sqrt{n}$. In other words:

If n has no divisors d in $1 < d \leq \sqrt{n}$, then n is a prime.

In Listing 2.39 (prime.py), isPrime() returns True or False depending on whether the supplied value is prime.

```
def isPrime(n):
  if n < 2:
    return False
  if n == 2 or n == 3:
    return True
  if n%2 == 0:
    return False
  for i in range(3, int(n**0.5)+1,
       2):
    if n%i == 0:
      return False
  return True
```

Listing 2.39. Test for primality

```
>>> from prime import *
>>> isPrime(17)
True
>>> isPrime(15)
False
```

prime.py contains two other useful prime functions: sievePrimes() and primes() (see Listing 2.40; prime.py).

```
def sievePrimes(n):
  if n < 2:
    return False
  prims = [False]*(n+1)
  for i in range(2,n+1):
    prims[i] = True
  for p in range(2,n+1):
    if (p*p <= n) and (prims[p]):
      for i in range(p*2, n+1, p):
        prims[i] = False
        p += 1
  return prims

def primes(n):
  prims = sievePrimes(n)
  return [i for i in range(len(
      prims)) if prims[i] ]
```

Listing 2.40. More tests for primality

sievePrimes() returns a list of booleans which indicate if the associated list index position is a prime. primes() converts this list into one containing only prime numbers. These functions can be called like so:

```
>>> sievePrimes(100)
[False, False, True, True,
... # many more
... True, False, False, False]
>>> primes(100)
[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]
```

2.13.2 The Prime Number Theorem.

The *Prime Number Theorem* (PNT) formalizes the notion that primes become less common as the integers increase. The theorem was proved independently by Jacques Hadamard and Charles Jean de la Vallé Poussin in 1896 using ideas introduced by Bernhard Riemann. The basic distribution is:

$$\pi(x) \sim x/\ln x$$

where $\pi()$ is a prime-counting function (the number of primes less than or equal to x). It states that for large enough x, the *probability* that a random integer not greater than x is prime is very close to $1/\ln x$.

countPlot.py collects counts of primes in groups of 10000 numbers between 2 and MAX. The count/$(x/\ln x)$ values are plotted, where x is the last number in each group. As you might expect, the curve in Fig. 2.13 slowly approaches 1 as x gets larger.

More details on the PNT, including a sketch of its proof can be found at `https://en.wikipedia.org/wiki/Prime_number_theorem`.

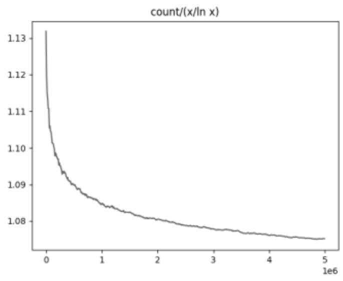


Figure 2.13. count/$(x/\ln x)$

2.13.3 The Goldbach Conjecture.

The Goldbach Conjecture is a deep number theoretic problem that is still unsolved after more than 250 years. It states that every even number, starting with 6, can be represented as the sum of two odd primes.

Listing 2.41 (goldbach.py) is supplied with an even number range (e.g. 200 to 228) and reports the number of ways each of those even numbers can be represented by the sum of two odd primes.

```
def goldbackFull(g):
  x = 3; count = 0
  while x <= g-x:
    if isPrime(x):
      if isPrime(g-x):
        print(f"{x}+{g-x}",end='')
        count += 1
    x += 2
  print()
  return count
```

Listing 2.41. Goldbach counts

For example:

```
> python goldbach.py
a b? 20 24
----- 20 -----
  3+17  7+13
 Count: 2; Expect: 1.1
----- 22 -----
  3+19  5+17  11+11
 Count: 3; Expect: 1.2
----- 24 -----
  5+19  7+17  11+13
 Count: 3; Expect: 1.2
```

- Which numbers have particularly low counts?
- Which numbers seem to have particularly high counts?
- Do you see any patterns?

From around 36 on, numbers divisible by 3 seem to have a high count. Powers of 2 and numbers $2p$ with p a large prime seem to have low counts. Divisibility by small odd primes seems to raise the Goldbach count on the average. Foe example, with $n = 3 * 5 * 7 = 105$, $2n = 210$ should have a high count. Indeed:

x	200	202	204	206	208	210	212	214	216	218	220	222	224	226	228
count	8	9	14	7	7	19	6	8	13	7	9	11	7	7	12

Table 2.10. Goldbach counts for x values

The reason why multiples of 3, say, have a higher than average Goldbach count is as follows. About half of all the primes up to any given limit are $\equiv 1$ mod 3 and the other half are $\equiv$ -1 mod 3. This is called the *Prime Number Theorem for Arithmetic Progressions*. Consequently, about half the sums $p + q$ of two primes are multiples of 3 while the other two residue classes mod 3 each have only about a quarter of these numbers.

A great deal of research has gone into developing probabilistic arguments in favor of the conjecture. One simplistic approach, based on the PNT, argues that an integer m selected at random has roughly a $\frac{1}{\ln m}$ chance of being prime. Thus if n is a large even integer and m is a number between 3 and $n/2$, then one might expect the probability of m and $n - m$ simultaneously being prime to be $\frac{1}{\ln(m)\ln(n-m)}$. Therefore, you might conclude that the total number of ways to write a large even integer n as the sum of two odd primes is roughly

$$\sum_{m=3}^{n/2} \frac{1}{\ln m} \frac{1}{\ln(n-m)} \approx \frac{n}{2(\ln n)^2}$$

Since $\ln n \ll \sqrt{n}$ goes to infinity as n increases, this suggests that every large even integer can be represented as the sum of two primes in many ways.

To test this heuristic, Listing 2.41 calculates $\frac{n}{2(\ln n)^2}$, and reports it as the 'Expect:' value. After some experimentation, it's clear that this approximation is somewhat inaccurate. The main problem is that it assumes that the probabilities of m and $n - m$ being prime are statistically independent, which isn't the case since they must both be odd numbers.

G.H. Hardy and J.E. Littlewood, developed much better probabilistic arguments in a paper published in *Acta Mathematica* in 1923:

$$P(2n) \sim C\left(\prod_{\substack{\text{odd prime}\\ \text{factors of n}}} \frac{p-1}{p-2}\right)\frac{2n}{(\ln 2n)^2}$$

where

$$C = 2 \prod_{\text{odd primes}} \left(1 - \frac{1}{(p-1)^2}\right) \approx 1.320324.$$

Actual Goldbach counts tend to differ from these approximations by several percent even for five-digit numbers.

goldbachComet.py plots Goldbach's comet, which reveals some interesting aspects of the counts. The graph is shown (in grayscale) in Fig. 2.14. A color version is online at colorComet.png.

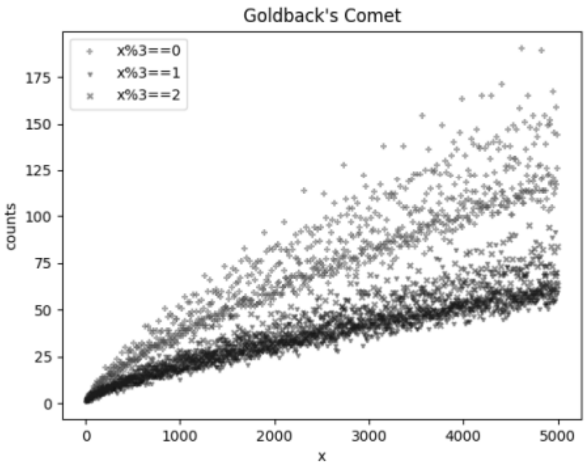


Figure 2.14. Goldbach's comet

The coloring and shape of the points is based on the value of $x/2\%3$. Red $+$ points corresponding to 0 mod 3, black $\vee$ points corresponding to 1 mod 3, and blue $\times$ points for 2 mod 3. In other words, the red points are multiples of 6, the green points are multiples of 6 plus 2, and the blue points are multiples of 6 plus 4.

The graph suggests that there are tight upper and lower bounds on the number of ways of splitting an even number into two primes, which depend strongly on the number modulo 3.

Two good places to start for more information on the conjecture are `https://en.wikipedia.org/wiki/Goldbach%27s_conjecture` and `https://mathworld.wolfram.com/GoldbachConjecture.html`. We also recommend *Uncle Petros and Goldbach's Conjecture*, by Apostolos Doxiadis [**Dox10**] which, although fictional, is still an enjoyable read. Back in 2001, as part of their effort to promote this book, the publishers announced a $1 million prize for the first person to prove Goldbach's Conjecture within the next two years. The reward went unclaimed.

2.13.4 Prime Twins.

Prime couples of the form $(p, p + 2)$ are called *twin primes*. For example: (3, 5), (5, 7), (11, 13), (17, 19), (29, 31), ...

One of the first discoveries when looking at twin primes was that, aside from (3,5), all of them have the form $(6n - 1, 6n + 1)$. This comes from noticing that any prime greater than 3 must be of the form $6n \pm 1$. To show this, note that any integer can be written as $6x + y$, where x is any integer, and y is 0,1,2,3,4, or 5. Now consider each y value individually.

When $y = 0$, $6x + y = 6x$ and is divisible by 6. When $y = 1$ there are no immediately recognizable factors, so this is a candidate for primacy. When $y = 2$, $6x+2 = 2\cdot(3x+1)$, so is not prime. For the case when $y = 3$, $6x+3 = 3\cdot(2x+1)$ and is not prime. When $y = 4$, $6x + 4 = 2 \cdot (3x + 2)$ and is not prime. When $y = 5$, $6x+5$ has no immediately recognizable factors, so is the second candidate for primacy.

So all primes can be represented as either $6n + 1$ or $6n - 1$, and twin primes, since they are separated by two, will have to be $6n - 1$ and $6n + 1$.

This also means that five is the only prime that belongs to two pairs, as every other pair is at least a multiple of 6 apart, and so can not overlap. Another way to prove this is to note that every third odd number is divisible by 3, and therefore no three successive odd numbers can be prime unless one of them is 3. Five is therefore the only prime that is part of two twin prime pairs.

Listing 2.42 (twins.py) prints all twin primes between $a > 8$ and b. x is the first integer ≥ 2, and since a has the form $6n-1$, then x is initialized to 6 * int(a/6) + 5.

```
a,b=map(int,input("a b=? ").split())
x = 6*math.trunc(a/6) + 5
while x <= b-2:
  if isPrime(x) and isPrime(x+2):
    print(x, x+2)
  x += 6
```

Listing 2.42. Finding twin primes

The following output are all the twin primes between $a = 1000000$ and $b = 1003000$:

```
1000037 1000039    1000721 1000723    1001549 1001551    1002359 1002361
1000211 1000213    1000847 1000849    1001807 1001809    1002719 1002721
1000289 1000291    1000859 1000861    1001981 1001983    1002767 1002769
1000427 1000429    1000919 1000921    1002149 1002151    1002851 1002853
1000577 1000579    1001087 1001089    1002257 1002259    1002929 1002931
1000619 1000621    1001321 1001323    1002341 1002343
1000667 1000669    1001387 1001389    1002347 1002349
```

Note that between 1000000 and 1001000 (the first millennium), there are 11 twins. When we split this interval evenly into 10 centuries, just two centuries are empty, the second and the fourth. Is this significant?

Suppose we toss 11 balls at random into 10 buckets. What is the expected number of empty buckets? Any one bucket remains empty with a probability of 0.9^{11}, so the expected number of empty buckets is $10 * 0.9^{11} = 3.138$. However, we have too little data to show that these twin primes are too evenly spread out to be random, although we might suspect it. This is why the output above covers a range of 3000 (two more millennia), making the data more likely to be random.

Indeed, the distances between twin primes are much larger than the distances between primes, and there's little dependency between them. Certainly, twin primes do become increasingly rare at larger ranges, which is in keeping with the general tendency of gaps between adjacent primes. However, it's unknown whether there are infinitely many twin primes (the *twin prime conjecture*) or if there is a largest pair.

2.13.5 Prime Triples, Quadruples.

A prime triple of the form $(p, p+2, p+6)$ is a 2-4 *triple prime*. A 4-2 triple prime has the form $(p, p+4, p+6)$.

A 2-4-2 quadruple prime has the form $(p, p+2, p+6, p+8)$. A 4-2-4 quadruple prime has the form $(p, p+4, p+6, p+10)$. Listing 2.43 (quadruple.py) generates 2-4-2 quadruples.

```
a, b = map(int, input("a b=? ").split())
p = 6*math.trunc(a/6) + 5
while p <= b:
  if isPrime(p) and isPrime(p+2) and \
     isPrime(p+6) and isPrime(p+8):
    print(f"{p:5} {p+2:5} {p+6:5} {p
        +8:5}")
  p += 6
```

Listing 2.43. Finding 4-2-4 quadruple primes

The output for $a = 5$, $b = 20000$:

```
    5     7    11    13                 5651  5653  5657  5659
   11    13    17    19                 9431  9433  9437  9439
  101   103   107   109                13001 13003 13007 13009
  191   193   197   199                15641 15643 15647 15649
  821   823   827   829                15731 15733 15737 15739
 1481  1483  1487  1489                16061 16063 16067 16069
 1871  1873  1877  1879                18041 18043 18047 18049
 2081  2083  2087  2089                18911 18913 18917 18919
 3251  3253  3257  3259                19421 19423 19427 19429
 3461  3463  3467  3469
```

Apart from the first quadruple all the others end with the digits 1, 3, 7, 9, which is easy to prove (see Ex. 12). So the steps must be multiples of 10 as well as of 6, i.e. 30-steps. That is, as soon as we find the first quadruple in an interval, we can proceed in steps of 30.

This can also be expressed as all quadruplets except (5, 7, 11, 13) are of the form $(30n + 11, 30n + 13, 30n + 17, 30n + 19)$ for some integer n. This structure is necessary to ensure that none of the four primes are divisible by 2, 3, or 5. A quadruplet of this form is also called a *prime decade*.

All prime decades have center values of the form $210n + 15$, $210n + 105$, or $210n + 195$ since the centers must be -1, 0, or +1 modulo 7.

Suppose we want all quadruples between a and b. Where do we start? With the first term of the sequence $30n+11$ which is $\geq a$, this becomes 30*trunc((a+18)/30) + 11. Check this by plugging in some values for a.

We replace the equation for p in Listing 2.43 by

```
p = 30*math.trunc((a+18)/30)+11
```

The while-test becomes:

```
while p <= b-8:
```

and the increment changes to:

```
p += 30
```

This speeds up the program considerably, although (5,7,11,13) is no longer reported of course.

2.13.6 Relatively Prime Numbers.

The probability that two random numbers both have a given prime p as a factor is $1/p^2$. So, the probability that they do *not* have p as a common factor is $1 - 1/p^2$. Therefore, the probability that two numbers have no common prime factors is

$$P = (1 - 1/2^2)(1 - 1/3^2)(1 - 1/5^2)(1 - 1/7^2)(1 - 1/11^2)\cdots$$

Using

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots,$$

this can be rewritten as

$$P = \left((1 + 1/2^2 + 1/2^4 + \cdots)(1 + 1/3^2 + 1/3^4 + \cdots)\cdots\right)^{-1}.$$

By the Unique Factorization Theorem (every positive integer is expressible as the product of primes in exactly one way), this expression is equivalent to

$$P = (1 + 1/2^2 + 1/3^2 + 1/4^2 + 1/5^2 + 1/6^2 + \cdots)^{-1}.$$

In words, the expansion of the multiplications generates reciprocals whose denominators are a product of ever square power of every prime. For example, somewhere we can find $1/(2^2 \cdot 3^4 \cdot 5^2)$, which is the same as $1/(2 \cdot 3^2 \cdot 5)^2$, or $1/90^2$. Indeed, every integer will appear somewhere in P's sum as a squared denominator.

We have arrived at the Basel problem (`https://en.wikipedia.org/wiki/Basel_problem`) which asks for the precise summation of the reciprocals of the squares of the natural numbers:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots$$

Euler solved this, as we outline in Sec. 6.7.7:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

And so, the desired probability is

$$P = 6/\pi^2 \approx 61\%.$$

Listing 2.44 (coPrime.py) uses this probability to obtain an estimate for π by examining 1 million random integer pairs.

```
def estPi(nPairs):
  count = 0
  for i in range(0, nPairs):
    n1 = random.randint(1, MAX_NUM)
    n2 = random.randint(1, MAX_NUM)
    # increment the counter if both integers are coprimes
    if math.gcd(n1, n2) == 1:
      count += 1
  prob = float(count)/nPairs
  return math.sqrt(6 / prob)
```

Listing 2.44. π based on the probability of being relatively prime

The result is only correct to 2 decimal places:

```
> python coPrime.py
Generating 1000000 random pairs
 pi est: 3.1431831955327896
math.pi: 3.141592653589793
```

Recall that $\phi(n)$ = the number of integers less than n that are relatively prime to n (see Sec. 2.1.2). Then $\phi(n)/n$ = the probability that a randomly chosen integer is relatively prime to n. (This is true because any integer is relatively prime to n if and only if its remainder, when divided by n, is relatively prime to n.) The result above therefore tells us that the average value of $\phi(n)/n = 6/\pi^2$.

2.13.7 Prime Spirals. The Ulam spiral, or prime spiral, is a visualization of prime numbers devised by Stanisław Ulam in 1963, and popularized in Martin Gardner's 'Mathematical Games' column in *Scientific American* a year later (March 1964). The column, 'Patterns and Primes' is reprinted in [**Gar71**], chapter 9.

The spiral is constructed by writing the positive integers in a square spiral (see Fig. 2.15a) and then highlighting the prime numbers (see Fig. 2.15b).

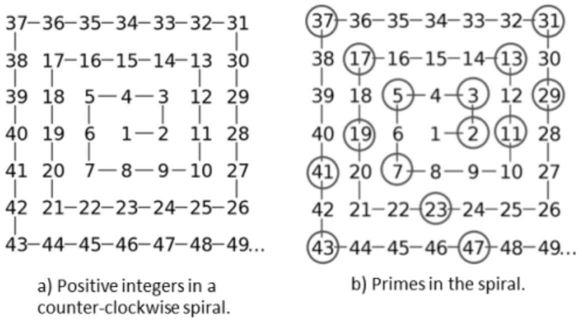


Figure 2.15. Constructing the Ulam spiral

Diagonal, vertical, and horizontal lines with a high density of primes become visible (see Fig. 2.16), which was so striking that the spiral was used as the cover illustration for that month's issue of *Scientific American*.

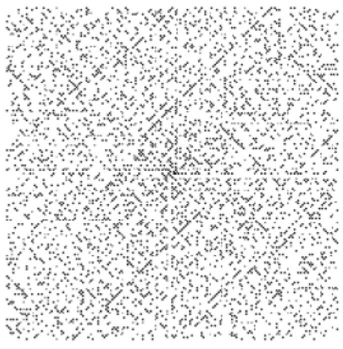


Figure 2.16. The Ulam spiral

Listing 2.16 was generated by ulamSpiral.py which uses isPrime() to identify the primes stored in a 2D list. The tricky part of the code is populating the array in the manner shown in Fig. 2.15a. An odd number sized array is created, so ensuring there's a single center cell. The code implements a 'state machine' based around the four states 'up', 'down', 'left', and 'right' corresponding to cell moves. The states are changed in an order that models counter-clockwise rotation, and a counter is used to decide how long each state is occupied. The counter is gradually incremented, which causes the turns to spiral outwards.

Diagonal, horizontal, and vertical lines in the number spiral correspond to polynomials of the form

$$f(n) = 4n^2 + bn + c$$

where b and c are integer constants. When b is even, the lines are diagonal, and either all the numbers are odd, or all even, depending on the value of c. This can be confirmed by manually calculating the equations for some of the diagonals around the center, as in Fig. 2.17. It's no surprise that all the primes other than 2 lie in alternate diagonals of the spiral.

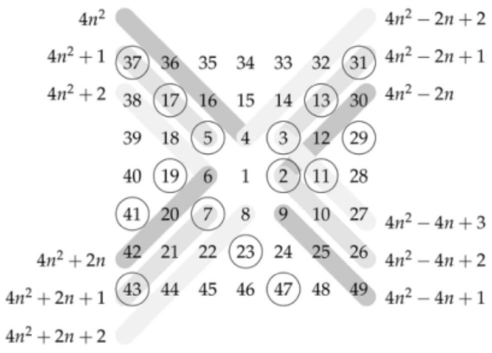


Figure 2.17. Diagonals as quadratic equations

The polynomial function can be obtained by looking at the differences between the values in typical sequences, such as the two shown in Fig. 2.18.

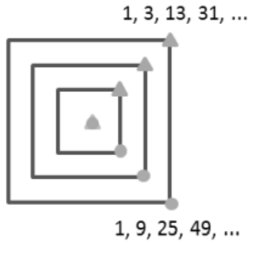


Figure 2.18. Two polynomial sequences in the Ulam spiral

The bottom right sequence in Fig. 2.18 is 1, 9, 25, 49, 81,..., whose difference sequence is 8, 16, 24, 32, ... In general, this b sequence has the form:

$$b_n = b_0 + 8n$$

b_0 is the starting value, which is 0 for this example.

This allows us to define the top-level sequence as:

$$a_n = a_0 + \sum_{k=1}^{n} b_k$$

a_0 is the starting value, which for this example is 1. It can be simplified:

$$\begin{array}{rcl} a_n & = & a_0 + \sum_{k=1}^{n}(b_0 + 8k) \\ & = & a_0 + nb_0 + 8\sum_{k=1}^{n} k \\ & = & a_0 + nb_0 + 8\left(\frac{n(n+1)}{2}\right) \\ & = & a_0 + nb_0 + 4n^2 + 4 \\ & = & 4n^2 + bn + c \end{array}$$

As to why some of the odd diagonals have a higher concentration of primes than others, consider $4n^2 + 6n + 1$ and $4n^2 + 6n + 5$ as examples. Plug in values for $n = 0, 1, 2,....$ into the first polynomial, and calculate the remainders modulo 3 to get 1, 2, 2, 1, 2, 2,.... The results for the second polynomial are 2, 0, 0, 2, 0, 0,.... This implies that in the second sequence two out of every three numbers are divisible by 3, and so not prime. However, the sequence for the first polynomial contains no numbers divisible by 3, which makes it more likely that it will have primes. It's only 'more likely' since we should also consider the remainders when the sequences are divided by other primes.

A more rigorous argument can use G.H. Hardy and J.E. Littlewood's *Conjecture F* which provides a formula for the number of primes of the form ax^2+bx+c (see `https://en.wikipedia.org/wiki/Ulam_spiral`). It implies that there's considerable variability in the prime density along different diagonals in the spiral. In particular, the density is highly sensitive to the discriminant of the polynomial, $b^2 - 16c$ (with $a = 4$).

Euler first noticed in 1772 that the quadratic polynomial

$$P(n) = n^2 - n + 41$$

is prime for the 40 integers $n = 0, 1, 2, ..., 39$, producing 41, 43, 47, 53, 61, 71, ..., 1601 respectively. This equation uses one of Euler's 'lucky' numbers – positive integers n such that for all integers k with $1 \leq k < n$, the polynomial $k^2 - k + n$ produces a prime number. Only six lucky numbers exist, namely 2, 3, 5, 11, 17 and 41. Note that they're all prime.

The spiralArr(a, N) function in ulamSpiral.py takes two arguments: the starting value a at the center of the 2D list and N for the size of the list. In Fig. 2.16, $a = 1$, but it's instructive to change it to one of Euler's lucky numbers. For example, Fig. 2.19 is the spiral generated when $a = 41$.

The prime density along several diagonals is markedly greater than in the earlier figure (Fig. 2.16).

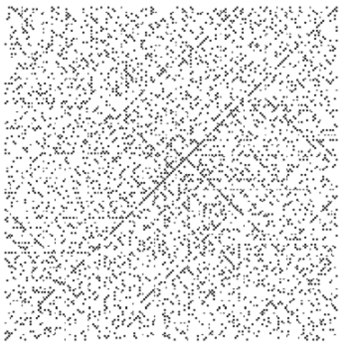


Figure 2.19. The Ulam spiral starting at 41

2.13.7.1 Prime Spiral Variants. Ulam wasn't the first to visualize primes in this way. In 1932, Laurence M. Klauber employed a triangle in which row n contains the numbers $(n-1)^2+1$ through n^2. As in the Ulam spiral, quadratic polynomials generate many primes that lie along diagonals. Also vertical lines, which correspond to numbers of the form k^2-k+M tend to have a high density of primes (see Fig. 2.20).

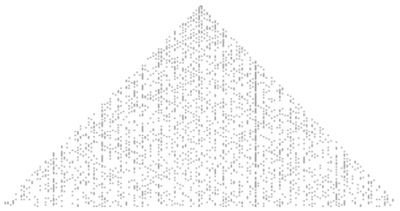


Figure 2.20. The Klauber triangle

Robert Sacks devised a variant of the Ulam spiral in 1994. Positive integers are plotted along an Archimedean spiral rather than Ulam's square spiral, spaced so that one perfect square occurs in each full rotation, as indicated in Fig. 2.21a.

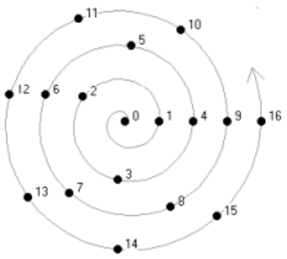
The spiral uses the following equations to generate polar coordinates (r,θ):

$$r = n^{1/2} \quad \text{and} \quad \theta = 2\pi \cdot 360 \cdot n^{1/2}.$$

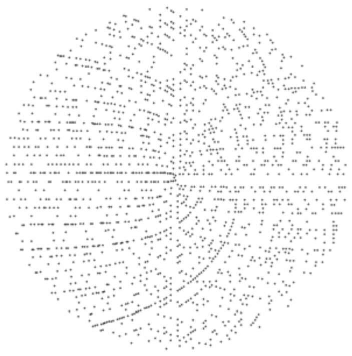
n is a positive integer. The angle calculation uses $n^{1/2}$ as the number of turns of the spiral, which has to be converted into degrees, and then to radians.

sacksSpiral.py draws the spiral using turtle graphics (Fig. 2.21b), and so its polar coordinates are converted to Cartesian (x,y):

$$x = r\cos\theta, \quad y = r\sin\theta$$



(a) Integer positions



(b) Drawing with a turtle

Figure 2.21. The Sacks spiral

The only prime-specific part of sacksSpiral.py is the if-test which decides whether an integer i should be drawn because it's a prime. It's interesting to replace this by other test functions to see how the spiral changes.

2.13.8 Factoring. We want to factorize an integer n, assuming that it's odd and greater than 1, so start with the trial divisor $d = 3$ (factor.py).

```
def factorsOdd(n):
  if n%2 == 0:
    print("Number must be odd")
    return []
  fs = []; d = 3
  while d*d <= n:
    while n == d*int(n/d):
      fs.append(d)
      n = n//d
    d += 2
  if n > 1:
    fs.append(n)
  return fs
```

Listing 2.45. Factorize an odd number

Let's factor some large odd numbers:

```
>>> from factor import *
>>> factorsOdd(123456789)
[3, 3, 3607, 3803]

>>> factorsOdd(999999999999)
[3, 3, 3, 7, 11, 13, 37,
 101, 9901]
```

All primes beyond 3 lie in one of two arithmetic progressions $6n - 1$ and $6n+1$. That is, from 5 onwards the trial divisor d can be incremented alternatively in steps of 2 and 4. This leads to a second version of the function shown in Listing 2.46 (factor.py).

```
def factors(n):
  fs = []
  while n == 2*int(n/2):
    fs.append(2)
    n = n//2
  while n == 3*int(n/3):
    fs.append(3)
    n = n//3
  d = 5; s = 2
  while d*d <= n:
    while n == d*int(n/d):
      fs.append(d)
      n = n//d
    d += s; s = 6-s
  if n > 1:
    fs.append(n)
  return fs
```

Listing 2.46. Factorizing any number

It first divides out the factors 2 and 3. Then it starts with $d = 5$ and step size $s = 2$. The assignment $s = 6 - s$ toggles s between 2 and 4. factors() has the advantage of supporting all positive integers:

```
>>> factors(22222222222222)
[2, 11, 239, 4649, 909091]

>>> factors(999999999999)
[3, 3, 3, 7, 11, 13, 37,
 101, 9901]
```

2.14 *n* as a Sum of Four Squares

Inspired by Diophantus' classical treatise *Arithmetica*, Bachet (Claude Gaspard Bachet de Méziriac) conjectured in 1621 that every natural number is the sum of at most four squares. He verified this up to $n = 351$. The first proof was given by Lagrange in 1770 (see https://en.wikipedia.org/wiki/Lagrange%27s_four-square_theorem). We can tell how difficult this theorem was from the fact that Euler, over the years, settled several special cases, but failed to prove rhe general form, although Lagrange's later proof did make use of Euler's four-square identity.

Listing 2.47 (fourSquares.py) finds all representations of n as $x^2 + y^2 + z^2 + u^2$ with $x \geq y \geq z \geq u$:

```
n = int(input("n=? "))
r = math.trunc(math.sqrt(n))
count = 0
for x in range(r, r//2-1, -1):
  for y in range(x, -1, -1):
    for z in range(y, -1, -1):
      for u in range(z, -1, -1):
        if x*x + y*y + z*z + u*u == n:
          print(f"{x:2} {y:2} {z:2} {u:2}    ", end='')
          count += 1
          if (count % 5 == 0):
            print()
```

Listing 2.47. Sum of four squares

For $n = 169$, 207 and 399, the program outputs:

```
> python.exe fourSquares.py
n=? 169
13  0  0  0   12  5  0  0   12  4  3  0   11  4  4  4   10  8  2  1
10  7  4  2    9  6  6  4    8  8  5  4
> python.exe fourSquares.py
n=? 207
14  3  1  1   13  6  1  1   13  5  3  2   11  9  2  1   11  7  6  1
11  6  5  5   10  9  5  1   10  7  7  3    9  9  6  3
> python.exe fourSquares.py
n=? 399
19  6  1  1   19  5  3  2   18  7  5  1   18  5  5  5   17 10  3  1
17  9  5  2   17  7  6  5   15 13  2  1   15 11  7  2   15 10  7  5
14 13  5  3   14 11  9  1   13 13  6  5   13 11 10  3   13 10  9  7
11 11 11  6
```

Jacobi's four-square theorem gives a formula (denoted by $r_4(n)$) for the number of representations of a natural number n as the sum of four squares:

$$r_4(n) = \begin{cases} 8\sum_{m|n} m & \text{if } n \text{ is odd} \\ 24\sum_{\substack{m|n \\ m \text{ odd}}} m & \text{if } n \text{ is even.} \end{cases}$$

In words, the number of ways to represent n is eight times the sum of the divisors of n if n is odd and 24 times the sum of the odd divisors of n if n is even.

This produces a much larger count than that suggested by the output from fourSquares.py since two representations are considered different if their terms are in a different order or if the integer being squared is negative. For example, $r_4(1) = 8$, three of which are:

$$\begin{gathered} 1^2 + 0^2 + 0^2 + 0^2 \\ 0^2 + 1^2 + 0^2 + 0^2 \\ (-1)^2 + 0^2 + 0^2 + 0^2. \end{gathered}$$

The theorem utilizes the divisors of a number n, which can be calculated using Listing 2.48 (jacobi.py).

```
def divisors(n):
  divs = [1]
  for i in range(2,
      int(math.sqrt(n))+1):
    if n%i == 0:
      divs.extend([i, n//i])
  divs.extend([n])
  divs = list(set(divs))
  divs.sort()
  return divs
```

Listing 2.48. Divisors of a number

```
def sumDivisors(n):
  odSums = 0; evSums = 0
  for i in range(1,
      int(math.sqrt(n))+1):
    if n%i == 0:
      if i%2 == 0:
        evSums += i
      else:
        odSums += i
      if n//i != i:
        if (n//i)%2 == 0:
          evSums += (n//i)
        else:
          odSums += (n//i)
  if n%2 == 1:  # odd
    return 8*(odSums + evSums)
  else:
    return 24*odSums
```

Listing 2.49. Divisors of a number (2)

However, Jacobi's formula only makes use of the sum of the divisors, and also distinguishes between odd and even divisors. This results in a modified divisor algorithm in Listing 2.49. It generates the following sequence of counts for $n = 0$ to 10:

```
1, 8, 24, 32, 24, 48, 96, 64, 24, 104, 144
```

This sequence is listed as A000118 at OEIS (`https://oeis.org/A000118`).

An interesting view of this topic appears in Chapter 4 of Polya [**Pol20**], where it's employed as an example of induction in number theory.

Exercises for Sections 2.13 to 2.14

(1) Denote by $G(2^n)$ the Goldbach count of 2^n. Check the following two-part table (2.11 and 2.12):

n	3	4	5	6	7	8	9	10	11	12
G(2^n)	1	2	2	5	3	8	11	22	25	53

Table 2.11. Goldbach count of 2^n.

(2) Plot $\ln(G(2^n))$ versus n. What do you get for large n?

(3) Write a program which checks if a number e satisfies the Goldbach conjecture. That is, it must print True as soon as it finds a representation of e as a sum of two odd primes $a + b$ with $a \leq b$. Also, find the quotient $\ln(e)/\ln(a)$.

n	13	14	15	16	17	18	19	20	21	22
$G(2^n)$	76	151	244	435	749	1314	2367	4239	7471	13705

Table 2.12. Goldbach count of 2^n continued.

(4) Write a program which prints a and e (see the previous question) only if a is larger than all the a's of the preceding e's. Once again find $\ln(e)/\ln(a)$.

(5) Print the first 20 emirps. An *emirp* ('prime' spelled backwards) are primes that when reversed (in their decimal representation) become different primes. (This rules out palindromic primes.)

(6) Find all positive integer solutions (x_n, y_n) of $x^2 - dy^2 = 1$ for $d = 2, 3, 5, 7, 10$ in a sufficiently large interval. From the data, guess recurrences $x_{n+i} = f(x_n, y_n)$, $y_{n+1} = g(x_n, y_n)$ and prove them by induction.

(7) Write programs which compute prime triples of type 2-4 as well as 4-2.

(8) Find the first prime p beyond a max value where $2p + 1$ is also a prime. Use max = 1000, 10000, 100000, 1000000, 10000000, 100000000.

(9) Are there quadratic polynomials which assume unusually many prime values? Count the primes of the form $x^2 + 1, x^2 + x + 41, x^2 + x + 1$ up to $x = 4000$.

(10) Rewrite isPrime() (Listing 2.39) more efficiently as follows: starting with 5 you can proceed alternately in steps of 2 and 4. If we set initially $s = 2$ then the counter $s = 6 - s$ oscillates between 2 and 4.

(11) Rewrite the function isPrime() in Ex. 10 to start from 7, with suitably changed step sizes, and find out if there is a further speedup.

(12) A *superprime* is an integer that is prime, *and* so is every integer obtained by deleting a digit from its right-hand side. For example, 7331 is prime, and so are 733, 73, and 7.

a) What 2- and 3-digit superprimes exist?

b) Which digits of a superprime can be a 1, 2, 4, 5, 6, 8, and 0?

(13) Write programs for computing 4-2-4 prime quadruples.

(14) A *Wilson prime of order n* is a prime number p such that p^2 divides $(n-1)! \cdot (p-n)! - (-1)^n$. For example, if $n = 1$, the formula reduces to $(p-n)! + 1$ where the only known primes are 5, 13, and 563.

Generate the Wilson primes for orders $n = 1$ to 11 and $p < 11000$.

(15) Factor the Fermat number $F_5 = 2^{2^5} + 1 = 4294967297$ with the two factor functions (Listings 2.45 and 2.46) and compare their run times. Use the following function for generating this Fermat number (and others):

```
def fermat(n):
  return 2**(2**n) + 1
```

(16) Compute a list of natural numbers which are not the sums of three squares (0 is allowed) and make a conjecture about the form of these numbers.

(17) Prove that the numbers of the form identified in Ex. 16 are indeed not the sums of three squares. *Remark*: it is a deep theorem that every positive integer not of the form given in the solution of Ex. 16 is the sum of three squares.

(18) a) Prove that a number of at least four digits cannot be equal to the sum of the cubes of its digits.

b) Find all the numbers with at most three digits which are equal to the sum of the cubes of their digits.

(19) *Wheels*. To factor a large number n we first divide it by 2, 3, 5, 7. From then on we can use step sizes 4, 2, 4, 2, 4, 6, 2, 6 of period 8 to get the trial divisors 11, 13, 17, 19, 23, ... which contain all the remaining primes. Use this to write a factoring function.

(20) Find the smallest integer which can be represented as a sum of three 4th powers in two different ways.

(21) Sieve out the positive integers which cannot be represented in the form: a) $x + y + xy$; b) $x + y + 2xy$ with positive integers x, y. Comment.

(22) Find all the numbers n up to 1000 so that all the numbers of the form $n - 2^k$ with $2 \leq 2^k < n$ are prime. The six numbers you will find are the only known numbers with this property.

2.15 The Maximum of a Unimodal Function

A function which is strictly increasing up to a point in an interval and strictly decreasing past that point is called *unimodal* in that interval. We want to find the maximum of a unimodal function f in an interval $[x1, x3]$.

The *golden-section* search method is suitable for when we have no information about the derivative of f, and must find a maximum or a minimum. Its name is derived from the fact that it calculates new points, $x2$ and $x4$, by using ϕ, the golden ratio. This guarantees that the interval width shrinks by the same constant proportion in each step, which means that the algorithm avoids the generation of very small intervals which may cause rounding errors (at least until it gets very close to the answer).

The starting situation is illustrated in Fig. 2.22. We are supplied with the starting interval $[x1, x3]$ and must calculate two new points, $x2$ and $x4$.

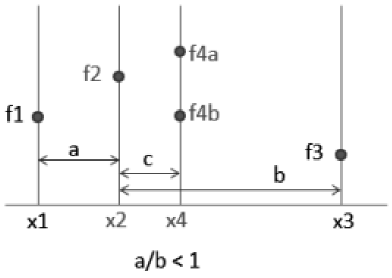


Figure 2.22. Starting a golden-section search

$x2$ will be chosen so it is a units from $x2$ and b units from $x3$, such that the intervals ratio $a/b < 1$. $x4$ will be c units from $x2$, with c determined by ensuring that the shrinking intervals ratios stay the same as the original a/b.

Each of the x points has a y value represented by $f1$, $f2$, etc. in Fig. 2.22. Note that there are two possible y-values for $x4$ – when the gradient at $f2$ is decreasing, $x4$ will have the y-value $f4b$. But when $f2$'s gradient is increasing then $x4$ will be assiged $f4a$. The two cases are illustrated in Fig. 2.23.

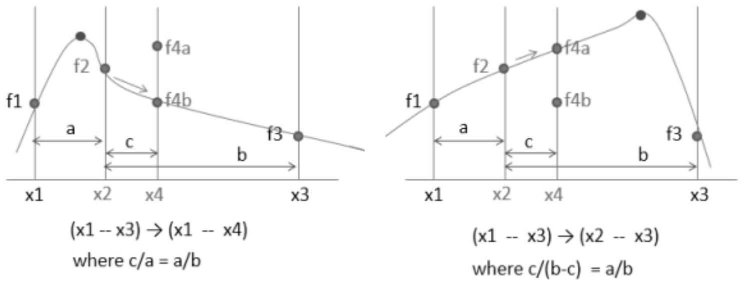


Figure 2.23. Reducing the golden-section search interval

In the first case (gradient decreasing at $f2$), then the maximum must be somewhere between $x1$ and $x2$, and so the search interval is reduced from $[x1, x3]$ to $[x1, x4]$, but with the proviso that

$$\frac{c}{a} = \frac{a}{b}.$$

In the second case (gradient increasing at $f2$), then the maximum must be somewhere between $x4$ and $x3$, and so the interval is reduced from $[x1, x3]$ to $[x2, x3]$, but with the proviso that

$$\frac{c}{b-c} = \frac{a}{b}.$$

These two constraints on how the interval is reduced are enough for us to derive equations for calculating $x2$ and $x4$.

Eliminating c from these two simultaneous equations yields

$$\frac{b}{a}^2 - \frac{b}{a} = 1.$$

If we substitute $x = b/a$ and solve $x^2 - x - 1 = 0$, the positive root is $x = \frac{1+\sqrt{5}}{2} = \phi$. Therefore, to maintain the same interval ratio throughout the search, we must set $a/b = 1/\phi$.

The code implements this idea when calculating $x2$ and $x4$. The key lines are:

```
PHI = (math.sqrt(5) + 1) / 2
x2 = x3 - (x3-x1)/PHI
x4 = x1 + (x3-x1)/PHI
```

We can show these assignments are correct by rewriting them in terms of the intervals used in Fig. 2.22, and the required intervals ratios:

$$\frac{a}{b} = \frac{1}{\phi} \quad \text{or} \quad \frac{b}{a} = \phi \tag{2.11}$$

$$\frac{c}{a} = \frac{1}{\phi} \quad \text{or} \quad \frac{a}{c} = \phi \tag{2.12}$$

$$\frac{c}{b-c} = \frac{1}{\phi} \quad \text{or} \quad \frac{b-c}{c} = \phi \tag{2.13}$$

Considering the $x2$ assignment:

$$\begin{aligned}
x2 &= x3 - (\tfrac{x3-x1}{\phi}) \\
\phi &= \tfrac{x3-x1}{x3-x2} = \tfrac{b+a}{b} \quad \text{from Fig. 2.22} \\
&= 1 + \tfrac{a}{b} \quad \text{using Equ. 2.11} \\
&= 1 + \tfrac{1}{\phi} \\
\phi^2 &= \phi + 1
\end{aligned}$$

Considering the $x4$ assignment:

$$\begin{aligned}
x4 &= x1 + (\tfrac{x3-x1}{\phi}) \\
\phi &= \frac{x3-x1}{x4-x1} = \frac{b+a}{c+a} \quad \text{from Fig. 2.22} \\
&= \frac{a\phi+a}{\frac{a}{\phi}+a} \quad \text{using Equs. 2.11 and 2.12} \\
&= \frac{\phi+1}{\frac{1}{\phi}+1} = \phi\frac{\frac{1}{\phi}+1}{\frac{1}{\phi}+1} = \phi
\end{aligned}$$

The preceding lines only caoture half of the rewrite since c can alternatively be replaced by Equ. 2.13 on the third line. We'll leave the details to you, but it uses the equality $c/a = b/(a+b)$ (a combination of Equs. 2.11 and 2.13) and ends in the same way as the $x2$ assignment rewrite, with $\phi^2 = \phi + 1$.

```
def goldSrch(fn, x1, x3):
  while abs(x3-x1) > EPS:
    x2 = x3 - (x3-x1)/PHI
    x4 = x1 + (x3-x1)/PHI
    if fn(x2) > fn(x4):
      x3 = x4
    else:
      x1 = x2
  return (x3+x1)/2

fn = lambda x: 1 - ((x-2)**2)
xMax = goldSrch(fn, 1, 5)
print(f"Maximum x: {xMax:.8f}")
```

Listing 2.50. Using the golden-section search

Listing 2.50 (goldSrch.py) uses goldSrch() to find the maximum of $y = 1-(x-2)^2$ in the interval $[1,5]$. The result:

```
> python goldSrch.py
Maximum x: 2.00000001
```

Golden-section search was first reported by J. Kiefer in 'Sequential Minimax Search for a Maximum', *Proc. Amer. Math. Soc.*, Vol. 4, No 3, pages 502–506, June 1953. He also developed the closely related Fibonacci search technique which finds the maximum/minimum of a sequence of values that has a single local maximum/minimum. A golden-section search is approximated using Fibonacci numbers applied to the indices of an integer sequence.

Note that the bisection method is a similar algorithm, but for finding a zero of a function (see Sec. 1.5). Also, bisection narrows in on the root using only two points, while golden-section search employs four, and bisection reduces the interval by a factor of two in each step, rather than by the golden ratio.

2.16 A Very Difficult Problem

We want to show that if a, b, and q are positive integers with $a^2 + b^2 = q(ab+1)$, then q is a perfect square.

This problem was proposed by the German Federal Republic group at the XXIX. IMO in Canberra, 1988. The Australian Problem Committee liked it a lot, but nobody could solve it, including George Szekeres and his wife, both famous problem solvers and creators. So it was passed to the four most eminent number theorists in Australia, but after six hours none of them had solved it either. To cut a long story short, the problem was chosen for the Olympiad, and eleven high school students produced complete solutions!

Suppose we are mathematicians of average ability, and have Python at our disposal, then the problem becomes comparatively simple. First, collect data, and study it for clues. When we see how to generate all the solutions output by programming, an elegant proof may suggest itself to us.

a) Due to the symmetry in a and b, we can assume that $a \leq b$, and it's also easy to see that $a = b$ only when $a = b = q = 1$. So we may additionally assume that $a < b$.

b) At this stage we should write a short piece of code (see Ex. 1) to generate solutions in a small range, such as $a \leq 150, b \leq 1000$. We get:

a	1	2	3	4	5	6	7	8	8	9	10	27	30	112
b	1	8	27	64	125	216	343	30	512	729	1000	240	112	418
q	1	4	9	16	25	36	49	4	64	81	100	9	4	4

Table 2.13. Solutions to the difficult problem

c) An examination of Table 2.13 suggests that $(a, b, q) = (c, c^3, c^2)$. Indeed:

$$c^2 + c^6 = c^2(c^4 + 1).$$

We have found one solution for each square.

d) Now turn to the triples that produce the same q, such as when $q = 4$:

2	8	30	112	a_1	b
8	30	112	418	b	a_2
4	4	4	4	q	q

Table 2.14. Solutions when $q = 4$

The second value of each triple (a, b, q) is the first value of the next triple, which suggests the transformation

$$(a_1, b, q) \rightarrowtail (b, a_2, q).$$

We can rewrite $a^2 + b^2 = q(ab + 1)$ as an explicit diophantine equation

$$a^2 - qba + b^2 - q = 0 \tag{2.14}$$

This can be viewed as a quadratic in a, with two solutions a_1 and a_2, which satisfy

$$a_1 + a_2 = qb \tag{2.15}$$

$$a_1 \cdot a_2 = b^2 - q \tag{2.16}$$

Equ. (2.15) shows that both a_1 and a_2 are integers, and we have

$$a_2 = qb - a_1. \tag{2.17}$$

Since $q \geq 2$ and $b > a_1$, we have $a_2 >$ b. So for any b, the two solutions a_1 and a_2 of Equ. (2.14) straddle b. By symmetry, the two solutions b_1, b_2 of Equ. (2.14) for a given a also straddle a. Thus one can get ever larger pairs a and b of integers which satisfy Equ. (2.14) for a fixed q.

We shall now prove that q is a square by going downwards through this family of solutions.

The original equation $a^2 + b^2 = q(ab + 1)$ shows that ab can not be negative. Hence a and b must have the same sign (or possibly one of them is 0). Using the straddling property we can alternately replace the larger of a or b by a smaller non-negative integer. Eventually, one of the numbers must reach 0, and q will be the square of the other.

Exercises

(1) Write a program which prints the triples a, b, and q in Table 2.13.

(2) Let a and b be positive integers such that $ab + 1$ divides $a^2 + ab + b^2$. Show that $(a^2 + ab + b^2)/(ab + 1)$ is the square of an integer. Explore the situation as in the IMO problem.

(3) Consider the general term $q = (a^2 + r \cdot ab + b^2)/(s \cdot ab + t)$ with r = -1, 0, or 1, and positive integers s and t. What condition between s and t must be satisfied in order that a) q is a square; b) $q \cdot t$ is a square.

(4) *Empirical Exploration.* Start with a stack of n coins, each having their 'head' face up. Now turn over the top i-substack and place it back on top for $i = 1, 2, ..., n, 1, 2, ..., n, 1, 2, ...$ For each i, check if the initial state (all 'heads') has re-occurred. Let $f(n)$ be the number of reversals until this happens. Write a program to calculate $f(n)$. Can you also discover a formula for $f(n)$?

A variant of this problem is concerned with *burnt pancakes*, where each pancake has a burnt side and all pancakes must end up with the burnt side on the bottom.

Pancake sorting is a recognized sorting type, where the aim is to put a disordered stack of pancakes of different sizes into size order by inserting a spatula at any point in the stack and using it to to flip all the pancakes above it (see `https://en.wikipedia.org/wiki/Pancake_sorting`). The problem is notable as the topic of a paper by Microsoft founder Bill Gates, co-authored with Christos Papadimitriou.

Additional Exercises for Chapters 1 to 2

(1) If n runs from 1 to a^2, the function $f(n) = \lfloor n + \sqrt{2n} + 0.5 \rfloor$ runs up to $a^2 + a$, leaving out exactly a numbers.

a) Write a program which prints the skipped values.

b) Make a conjecture and try to prove it.

This question, and the next, comes from Essay 12 in Honsberger [**Hon70**].

(2) If n runs from 1 to some upper bound max, then $f(n) = \lfloor n + \sqrt{2n} + 0.5 \rfloor$ will skip some values.

a) Write a program which prints the skipped values.

b) Make a conjecture and try to prove it.

(3) a) For $k = 2, 3, 4, \ldots$ investigate the values left out by the functions $f_k(n) = \lfloor n + \sqrt{n/k} + 0.5 \rfloor$.

b) Try to guess a closed formula for these values.

c) Prove your conjecture.

(4) a) Find the values skipped by $f(n) = \lfloor n + \sqrt{kn} + 0.5 \rfloor$. Develop a closed formula for the skipped values and prove your conjecture.

b) Find the values skipped by $f(n, p, q) = \lfloor n + \sqrt{np/q} + 0.5 \rfloor$ for positive integers p, q.

(5) *Stirling Numbers of the Second Kind.* Let $S(n, k)$ be the number of partitions of the set {1, 2, ..., n} into exactly k nonempty subsets. Show that

$$S(n,1) = S(n,n) = 1, \quad S(n,k) = 0 \text{ for } k > n,$$
$$S(n,k) = S(n-1,k-1) + kS(n-1,k).$$

Write recursive and iterative programs for $S(n, k)$.

(6) *Stirling Numbers of the First Kind.* Let $f(n,k)$ be the number of permutations of {1, 2, ..., n} with exactly k cycles. Show that

$$f(n,k) = f(n-1,k-1) + (n-1)f(n-1,k).$$
$$f(n,1) = (n-1)! \quad f(n,n) = 1 \quad f(n,k) = 0 \text{ for } k > n.$$

Write recursive and iterative programs for $f(n,k)$.

(7) Find all the representations of n in the form $x^3 + y^3 + z^3$ with non-negative integers x, y, z. Explain why numbers of the form $9k \pm 4$ cannot be the sums of three cubes.

(8) Find all the representations of n as a sum of four non-negative integer cubes.

(9) Write a program for listing the integers from 1 to some maximum which can be represented in the form $x^2 + 2y^2$.

(10) Show empirically that any number from 1 to 10000 can be represented as a sum of at most three triangular numbers, i.e. numbers of the form $\binom{n}{2}$. Do it by sieving.

In 1796, Gauss showed that any positive integer is the sum of three triangular numbers. It's known as the *Eureka theorem* because of what he wrote in his diary: EYPHKA! Num $= \triangle + \triangle + \triangle$

(11) *The mode* (*plateau problem*). Let a be a list of n integers sorted into increasing order. The mode m is the most frequent value. Construct a function which finds m and its frequency f. The program should also find the length of the longest plateau (stretch of equal values) for any list, even if it is not sorted.

(12) *Common elements of two sequences.* We are given two lists f and g of natural numbers sorted into increasing order. Find the terms which occur in both sequences and the number k of common terms.

(13) A sequence $g(n)$ is defined by $g(0) = 0$ and $g(n) = n - g(g(n-1))$ for $n > 0$.

a) Construct a recursive program for $g(n)$ and compute its values up to $g(40)$.

b) Write an iterative version.

c) Plot the sequence and try to guess a closed formula for $g(n)$.

d) Prove your conjecture.

This question, and the next two, come from Hofstadter [**Hof99**].

(14) A sequence $h(n)$ is defined by

$$h(0) = 0 \text{ and } h(n) = n - h(h(h(n-1))) \text{ for } n > 0.$$

a) Write a recursive program for $h(n)$ and find its values up to $h(40)$.

b) Write an iterative version.

c) Plot the sequence and make conjectures about a closed formula for $h(n)$.

(15) A sequence $q(n)$ is defined by

$$q(1) = q(2) = 1 \text{ and } q(n) = q(n - q(n-1)) + q(n - q(n-2)) \text{ for } n > 2.$$

a) Write a recursive program and compute the values of $q(n)$ up to $q(30)$.

b) Write an iterative version and compute the values up to $q(2000)$.

c) Print the integers 7, 13, 15, 18, ... that get left out. Are there infinitely many of these? (This is a hard unsolved problem.) See also Ex. 23, below.

(16) Find all 4-2-4-2-4 sextuples of primes below 50000.

(17) Find numerical evidence for the following theorem due to Euler:

Every number 2^n for $n \geq 3$ can be represented in the form $2^n = 7x^2 + y^2$ with odd x and y.

From the numerical evidence deduce a proof.

This problem comes from the 48th Moscow Mathematics Olympiad, 1985. A book containing over 60 years of these problems (with hints and solutions), edited by D. Leites, can be downloaded from `https://diendantoanhoc.org/index.php?app=core&module=attach&section=attach&attach_id=22389`. An alternative source is AoPS' Olympiad Archive at `https://artofproblemsolving.com/wiki/index.php?title=Olympiad_Archive`.

(18) Consider again Perrin's sequence v_n defined by $v_0 = 3$, $v_1 = 0$, $v_2 = 2$, $v_n = v_{n-2} + v_{n-3}, n \geq 3$ from Sec. 1.5. Write a program which for each n from 3 to 1 million, tests if $n|v_n$. If the test is positive and n is not prime, it should print n. Do you now have enough evidence for the conjecture $n|v_n \Leftrightarrow n$ is a prime?

(19) On visiting Srinivasa Ramanujan, G.H. Hardy mentioned that he took taxi #n, but this number did not look remarkable to him. "On the contrary", replied Ramanujan, "it is the smallest natural number which can be represented as a sum of two cubes in two different ways" Find n. (Next problem is a continuation.)

(20) Hardy then asked if Ramanujan knew the answer to the problem for fourth powers. After some deliberation, he replied: "I cannot find an example, I think the first such number must be large". Indeed, Euler had earlier discovered that $635318657 = x^4 + y^4 = u^4 + v^4, x < y, u < v, x \neq u$. Find x, y, u, v. Is this the smallest such number?

(21) *The smallest common element of three ordered lists.* Given are three increasingly ordered lists F, G, and H. Find the smallest common element. Since the lists are ordered, we need the smallest i, such that $F[i] = G[j] = H[k]$. Apply the algorithm to the Fibonacci sequence F, to the squares G, and to the multiples H of 9. Print i, i, k, and $F[i]$.

(22) A sequence is defined as

$$f(1) = f(2) = 1, \quad f(n) = f(f(n-1)) + f(n - f(n-1)) \text{ for } n > 2.$$

Table 2.15 shows some values of this sequence. We observe that $f(13) = \cdots = f(16) = 8$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
f(n)	1	1	2	2	3	4	4	4	5	6	7	7	8	8	8	8	9	10	11	12	12

Table 2.15. The $f(n)$ sequence.

There is an *interval of constancy* of length 4 starting at 13. On the other hand, f(16) to f(20) is a run of length 5. Between 1 and some maximum, find:

a) the leftmost (rightmost) longest interval of constancy,

b) the leftmost (rightmost) longest run.

(23) Let's return to the sequence in Ex. 15., defined by $q(1) = q(2) = 1$ and $q(n) = q(n-q(n-1)) + q(n-q(n-2))$ for $n > 2$. Hofstadter [**Hof99**] calls its growth pattern chaotic, but that only applies to local growth. Find a simple global regularity in its growth.

(24) *McCarthy's Function.* A function G is defined on the integers as follows:

$$G(n) = \begin{cases} n - 10 & \text{for } n > 100 \\ & \text{else } G(G(n+11)). \end{cases}$$

Write a recursive program which finds $G(n)$.

(25) A function Q is defined on the integers by

$$Q(a,b) = \begin{cases} 0 & \text{if } a < b \\ Q(a-b,b)+1 & \text{if } a \geq b. \end{cases}$$

Find $Q(a,b)$.

(26) There is a conjecture that every large integer n is either a square, or the sum of a prime and a square. Write a program which finds all the numbers from 0 to 32000 which are not representable in this way. Is the conjecture plausible? Also write a program, which for input n, prints the number of representations $n = m^2 + p$. For $n = 11$ it should print 3, since $11 = 0^2 + 11 = 2^2 + 7 = 3^2 + 2$.

(27) Find the function L defined on the integers by

$$L(n) = \begin{cases} 0 & \text{if } n = 1 \\ L(n \operatorname{div} 2) + 1 & \text{if } n > 1. \end{cases}$$

(28) What does Listing 2.51 (product.py) do for the inputs x, y?

```
def prod(x,y):
  if y == 1:
    return x
  elif (y%2 == 1):
    return x + prod(x, y-1)
  else:
    return prod(x+x, y//2)

x,y = map(int, input("x y=? ").split())
print(prod(x,y))
```

Listing 2.51. Not a prime ordering function

(29) Find a permutation of the set $\{1, 2, ..., 9\}$ so that the number consisting of the first n digits is divisible by n for every n from 1 to 9. The most obvious coding approach is rather slow; there are several ways to speed up the search.

(30) Let B_n be the number of partitions of an n-set. The numbers B_n can be computed by means of the "Bell triangle" below. The first column is $B_0, B_1, B_2, ...$ and the last entry in each row is the next Bell number. Here $B_0 = 1$ by definition.

```
  1
  1   2
  2   3   5
  5   7  10  15
 15  20  27  37  52
 52  67  87 114 151 203
203 255 322 409 523 674 877
877
```

How is this triangle formed? Write a program which for input n prints the Bell triangle up to line n.

(31) Write a program based on $S(n, k)$, the *Stirling Numbers of the Second Kind* (see Ex. 5) which for input n prints B_n.

(32) a) Write a program which prints

```
1
2 2
3 3 3
4 4 4 4
.........
```

b) Show that $a_n = \lfloor \sqrt{2n} + 0.5 \rfloor$ is a closed expression for the n-th term of this sequence when written linearly as 1, 2, 2, 3, 3, 3, 4, 4, 4, 4,....

c) It is claimed that $a_n = \lfloor \frac{1}{2}(1 + \sqrt{8n - 7}) \rfloor$ and $a_n = \lceil \frac{1}{2}(\sqrt{8n + 1} - 1) \rceil$ are also closed expressions for the n-th term. Check if these claims are true for $n = 1..1000$.

(33) a) Write a program which prints the sequence

```
1
2 4
5 7 9
10 12 14 16
.............
```

b) Show that $a_n = 2n - \lfloor \sqrt{2n} + 0.5 \rfloor$ is a closed expression for the n-th term of the sequence when written linearly.

(34) Let X and Y be natural numbers. We say that X is contained in Y if the binary representation of Y goes over into that of X when some (or possibly no) digits are omitted. For example, $X = 1010$ is contained in $Y = 1001100$. Construct an algorithm which, for given natural numbers A, B, finds the largest C contained in A as well as in B.

This question comes from the International Collegiate Programming Contest (ICPC), 1987. Archives dating back to 2008 are online at `https://icpcarchive.github.io/`, and questions for the years 1991 to 2006 can be found in Poucher [**PR09**]. For older problems, you may need to use the Internet Archive's Wayback Machine. we had success with the link `http://icpc.baylor.edu/past/`, then clicking on the tab "Past Problems".

(35) A sequence is defined by $a_1 = 1, a_n = \lfloor \sqrt{a_1 + ... + a_{n-1}} \rfloor, n > 1$. Explore this sequence.

(36) *Ulam's lucky number.* From the list 1, 2, 3, 4,... of all positive integers remove every second number, leaving 1, 3, 5, 7, 9,.... Since 3 is the first number (above 2) that has not been used as a sieving number, we remove every third number from the remaining numbers, yielding 1, 3, 7, 9, 13, 15, 19, 21, Now every seventh number is removed, leaving 1, 3, 7, 9, 13, 15, 21, Numbers that are never removed are considered to be "lucky". Write a program that prints the lucky numbers up to some maximum.