1

Introductory Problems

This chapter is introductory in the sense that it focuses on the parts of Python that confuse beginners, and also on visualization tools (matplotlib and turtle). One of the most confusing areas of Python, which it shares with other programming languages, is recursion, which has motivated our choice of factorial, Fibonacci, the Towers of Hanoi, and the other examples. Another area, more specific to Python, are the use of generators. Other topics discussed here include a comparison of recursion vs. iteration, different ways of measuring running time, techniques for reducing running time, and integers vs. reals (floats and decimals in Python).

1.1 The Factorial

The factorial function is defined on non-negative integers as:

$$1! = 1, \quad n! = n(n-1)!$$

This can be readily translated into the recursive function in Listing 1.1.

```
def factRec(n):
  if n <= 1:
    return 1
  else:
    return n*factRec(n-1)
```

Listing 1.1. Recursive factorial

Note that this is not an exact conversion since the base case (the if-case) has been modified to handle 0 and negative numbers (`factsLib.py`).

factRec() can be called like so from inside Python's IDE:

```
>>> from factsLib import *
>>> factRec(5)
120
```

A series of function instantiations deal with progressively decreasing values, until the base case is encountered at $n = 1$ (see Fig. 1.1). These are stored as *stack frames* by the run time system until later instantiations complete and return values to them.

One problem for a beginner is recursion's dynamic nature – Fig. 1.1 only depicts the moment when the maximum number of function instantiations exist, just prior to the base case returning. The parent function will resume, return a value, and disappear.

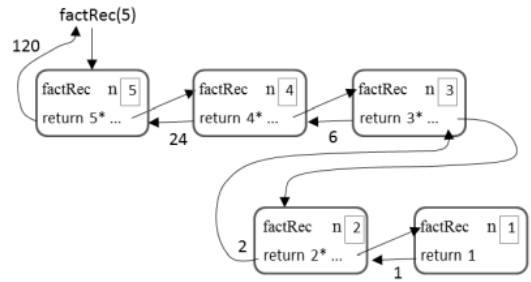


Figure 1.1. Executing fact(5)

Another difficult aspect is that the call to factRec(5) eventually triggers the creation of five copies of the function's n variable, whereas only a single n is present in the factRec() source. Indeed, the number of variables created at run time varies linearly with the size of the initial n.

1.1.1 Iterative Factorials.

It's possible to define factorial as a product:

$$n! = \prod_{i=1}^{n} i$$

This is easily translated into iterative code (factsLib.py), although there are several possibilities:

```
def factIter(n):
  fact = 1
  while n > 0:
    fact *= n
    n -= 1
  return fact

def factIterUp(n):
  fact = 1; count = 0
  while count < n:
    count += 1
    fact *= count
  return fact

def factRange(n):
  fact = 1
  for i in range(1, n+1):
    fact *= i
  return fact
```

Listing 1.2. Three iterative factorials

They all produce the same result as the recursive version:

```
>>> factIter(5)
120
>>> factIterUp(5)
120
>>> factRange(5)
120
```

But this may not be the case when n is large:

```
>>> factRange(1000)
402387260077093773543702433923003985719374864210 71
463254379991042993851239862902059204420848696 9...
700277534720000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
>>>
>>> factRec(1000)
Traceback:
  File "factsLib.py", line 14, in factRec
    if n < 2:
RecursionError: maximum recursion depth exceeded in comparison
```

The space used to represent Python integers automatically grows, so factRange() has no trouble calculating 1000!. This is also the case for factRec(), but it falls foul of Python's default recursion depth limit (100). The depth required by factRec() depends linearly on the size of n, and so factRec(1000), needing at most 1000 stack frames, crashes.

The depth limit can be changed:

```
>>> import sys
>>> sys.setrecursionlimit(10**6)
>>> factRec(1000)
402387260077093773543702433923003985719374864210 71
463254379991042993851239862902059204420848696 9...
700277534720000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
>>>
```

Another problem that may arise in Python version 3.10.7 or later is due to its integer string conversion length limit, which defaults to 4300 characters. It will crop up when a very large factorial is converted to a string before being printed:

```
>>> factRange(10000)
Traceback:
ValueError: Exceeds the limit (4300 digits) for
integer string conversion;
use sys.set_int_max_str_digits() to increase the limit
```

This limit can be removed:

```
>>> import sys
>>> sys.set_int_max_str_digits(0)
>>> factRange(10000)
28462596809170545189...
   :  # many lines not shown
000000000000000
```

1.1.2 Recursion vs. Iteration. Recursive and iterative programs can be compared by looking at their running times and amount of memory used. factCompare.py employs Python's time and tracemalloc modules, running factRec() and factRange() with a range of values.

The times and memory allocations are plotted in the graphs shown in Fig. 1.2. (If you're unfamiliar with Python's Matplotlib module, please read Appendix F.)

The left-hand chart shows that there's essentially no difference between the recursive and iterative running times; both are O(n). The significant difference is highlighted by the right-hand graph – the memory used by the iterative code is constant (it only requires two variables and a range() iterator), whereas the memory used by the recursive function increases linearly since a new stack frame is created for each recursive call. This was illustrated in Fig. 1.1.

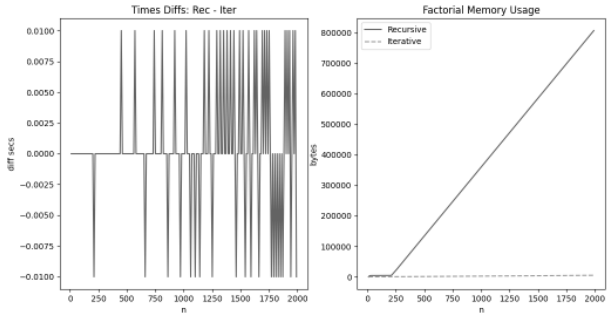


Figure 1.2. Time and memory usage comparisons

1.1.3 Functional Factorials. Python utilizes a number of functional programming ideas, including list comprehensions, anonymous functions, and higher-order operators such as map(), reduce() and apply(). This allows (at least) two other ways to write factorial (factsLib.py):

```
def factProd(n):
  return math.prod(range(1, n+1), start=1)

def factReduce(n):
  return reduce(lambda x, y: x * y, range(1, n+1), 1)
```

Listing 1.3. Two functional factorials

The product definition for factorial is implemented succinctly using reduce() in factReduce(). It applies the supplied function (multiplication) to an iterable, reduces it to a single cumulative value.

factTimings.py calculates the average running times for all of these variants (factRec(), factIter(), factRange(), factProd(), factReduce()) and Python's own built-in, math.factorial(). Unlike the timing code in Listing 1.2 (factCompare.py), Python's timeit module is employed (`https://docs.python.org/3/library/timeit.html`). It provides a convenient way to run functions several times over, in order to calculate an average time.

The resulting chart is in Fig. 1.3 (factTimes.py). The unsurprising result is that math.factorial() is the fastest (it's a small wrapper around a high-speed C library). Also, range() is faster than a while-loop, but the timing difference is marginal. Although the reduce and recursive versions are arguably the closest to the math definitions, they're also the slowest.

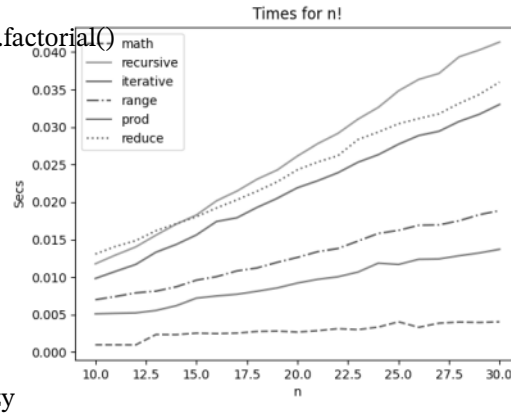


Figure 1.3. Factorial times

1.1.4 Lazy Evaluation. Lazy evaluation debuted in the lambda calculus, and was first utilized in functional programming languages. It lets the evaluation of an expression be suspended until required by another part of the computation.

Lazy evaluation can be employed in Python with *generator* functions. Generator code looks like an ordinary function but utilizes 'yield' rather than 'return'. When the generator is invoked, an iterator is created, which will yield a new result whenever it is called with next().

factGen() (factsLib.py) shown below is an *infinite* generator since it can keep producing factorials forever (if requested).

```
def factGen():
  fact = 1
  i = 0     # fact(0) = 1
  while True:
    yield fact
    i += 1
    fact *= i
```

Listing 1.4. A factorial generator

```
>>> fg = factGen()
>>> next(fg)
1
>>> next(fg)
1
>>> next(fg)
2
>>> next(fg)
6
>>> next(fg)
24
```

The call to factGen() creates an iterator (called fg), which returns its first value when next() is called. The iterator wakes up and executes up to the yield line which returns the fact value *and then suspends*. When next() is called again, the iterator resumes from where it was suspended and progresses until it next encounters a yield.

Python's itertools module offers higher-level operations for working with iterators, many of which will be familiar to functional programmers (`https://docs.python.org/3/library/itertools.html`). Listing 1.5 (factGens.py) contains a few examples:

```
import itertools
from factsLib import *

n = int(input("n=? "))
print("Fact("+str(n)+"):",next(itertools.islice(factGen(),n,None)))
lessFacts = list(itertools.takewhile(lambda i: i < n*n,factGen()))
print("\nFacts < :", *lessFacts)
print("\nnum Facts: ", *facts(n))
print()
gen = factGen()
for i in range(n+1):
  print(f"{next(gen):12d}", end = '')
  if (i+1)%5 == 0:
    print()
```

Listing 1.5. Using the factorial generator

Listing 1.5 uses itertools.islice() and itertools.takewhile(). islice allows a slice of several values to be returned from the iterator, although our example only gets the n-th value. takewhile() keeps extracting data from an iterator while its condition (in this case, $i < n^2$) is true.

The third example calls facts(n), a function from factsLib.py, which constructs a list of the first n numbers of factGen(). The code at the end of Listing 1.5 shows how to embed next() inside a loop.

```
> python factGens.py
n=? 5
Fact(5): 120
Facts < : 1 1 2 6 24
num Facts:  1 1 2 6 24 120
```

1.1.5 Factorial Approximations. Stirling's series (`https://mathworld.wolfram.com/StirlingsSeries.html`) is an asymptotic approximation for n!:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} + O(\frac{1}{n^4})\right)$$

The approximation error is equal to the first omitted term, which for the above is $O(\frac{1}{n^4})$. Another source of errors are the use of floats to represent π, square roots, and real-number division; see Appendix E for more discussion of these issues.

Listing 1.6 (factsLib.py) contains two implementations of Stirling's approximation: stirling() utilizes only the first term of the series, while stirling2() includes the second.

```
def stirling(n):
  return math.sqrt(2*math.pi*n)*(n/math.e)**n

def stirling2(n):
  return  (1 + 1/(12*n)) * math.sqrt(2*math.pi*n)*(n/math.e)**n
```

Listing 1.6. Two Stirling approximations

factApproxs.py runs these two on a variety of inputs, and charts their *relative* errors compared to math.factorial() (see Fig. 1.4).

Although the relative errors are small, the enormous size of the factorials means there's very few correct digits in the results. For instance, consider 20! and 100!:

```
>>> factRange(20)
2432902008176640000
>>> stirling(20)
2.422786846761135e+18
>>> stirling2(20)
2.432881791955973e+18
```

```
>>> factRange(100)
933262154439441526816992388563...
168640000000000000000000000000
>>> stirling(100)
9.32484762526942e+157
>>> stirling2(100)
9.332618331623811e+157
```

stirling() only manages two correct digits, while stirling2() produces four or five. It might be tempting to switch to an implementation of Stirling's series using decimals (see Appendix E for more on Python's decimal), which replaces float's 15 digit precision with 28 digits (factsLib.py). Let's try it:

```
def stirlingD(n):
  root = (2*PI*D(n)).sqrt()
  return root*((D(n)/E)**D(n))

def stirling2D(n):
  term12 = D(1) + D(1)/(D(12) * D(
      n))
  root = (2*PI*D(n)).sqrt()
  return term12*root*((D(n)/E)**D(
      n))
```

Listing 1.7. Stirling approx using decimals

```
>>> stirlingD(50)
Decimal('3.036344593938155820798372692E+64')
>>> stirlingD(100)
Decimal('9.324847625269343247764756226E+157')

>>> stirling2D(50)
Decimal('3.041405168261386080499703315E+64')
>>> stirling2D(100)
Decimal('9.332618331623734367137893520E+157')
```

The number of digits in the answers increases, but the number of correct digits does not. The reason is that the greater part of the error is due to the series approximations not the use of floating point. In the case of stirling() (and stirlingD()), the error is $O(\frac{1}{n})$ which for n = 100 means an error of about 1%, or two-digit precision. For stirling2() (and striling2D()), the error is $O(\frac{1}{n^2})$ which for n = 100 is an error of 0.01% and a precision of four digits.

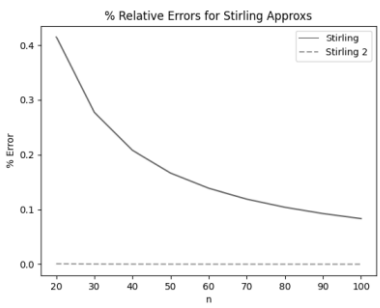


Figure 1.4. Relative errors when using Stirling

1.1.6 Using Log Gamma. In languages that don't have the benefit of Python's unlimited size integers, a commonly suggested alternative for calculating large factorials is to manipulate them

as logarithms. For example, this would replace the division of two massive factorials (e.g. 100!/99!) with the subtraction of two much smaller logarithms (although a wily student would just simplify the expression directly).

Python doesn't have a logarithmic factorial function, but it does offer logarithmic gamma – math.lgamma(). The gamma (Γ) function extends factorial to complex numbers, excluding negative integers, but the factorial of a positive integer n is simply $n! = \Gamma(n+1)$, so that $\log n! = \log \Gamma(n+1)$. 100!/99! can be expressed as:

```
>>> import math
>>> math.exp( math.lgamma(101) - math.lgamma(100))
99.9999999999981
```

Since these functions utilize floats, we can expect them to be accurate to at most 15 digits, as here. This may be satisfactory, but it illustrates that using Python's gamma() or lgamma() is *not* the same as calculating factorial with integers.

In general, it's much better to stick with integer-based calculations if possible. If you must use floats, then be aware of the problems with rounding errors and precision. These issues are explained at some length in Appendix E. Python's decimal type may help, but it depends on the source of the approximation errors (e.g. do they stem from the math, or from the algorithm, or both).

1.2 Fibonacci's Sequence: A Two-Term Recursion

The Fibonacci sequence is given by the recursive relation:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \quad n > 1 \end{aligned}$$

An alternative is to start the base cases at F_1 and define $F_2 = 1$.

Leonardo Pisano (1170-1250), or Leonardo of Pisa, but better known by his nickname, Fibonacci, used the sequence in his *Liber Abaci* ('Book of Calculations') published in 1202, although it was already known to Indian mathematicians as early as 200BC. The series is ubiquitous in nature, and has been eagerly connected to many forms of art and architecture, although some of these seem a little dubious. For a good account, see Vorobev [**Vor11**].

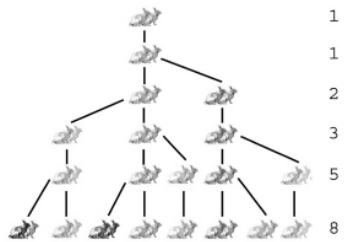


Figure 1.5. Fibonacci's rabbits

The *Liber Abaci* introduced Hindu-Arabic digits to Europe, but also debuted Fibonacci's rabbit problem:

> A pair of rabbits is born and produces a further pair in the second month of its existence, and in each subsequent month. Each succeeding generation reproduces in the same way. How many pairs are there after a year?

The situation is depicted in Fig. 1.5.

A quick calculation produces Table 1.1. After a year (i.e. at the start of the 13th month), there will be 233 pairs of rabbits.

Month	1	2	3	4	5	6	7	8	9	10	11	12	13
No. Pairs	1	1	2	3	5	8	13	21	34	55	89	144	233

Table 1.1. Fibonacci's rabbits

The recurrence relation is readily translated into recursive code (Listing 1.8; fibsLib.py).

```
def fibRec(n):
  if n <= 0:  # base cases
    return 0
  elif n == 1:
    return 1
  else:
    return fibRec(n-1) + fibRec(n-2)
```

Listing 1.8. Recursive Fibonacci numbers

Happily, the function gives the same answer for the number of rabbits:

```
>>> from fibsLib import *
>>> fibRec(13)
233
```

The call graph for fibRec(5) (Fig. 1.6) reveals a worrying duplication of effort in different parts of the execution tree (or *call graph*). Note, we've abbreviated the function name to fib() to make the diagram less cluttered.

fibRec(2) is evaluated three times, and fibRec(3) twice. The amount of repeated work only becomes more serious as the number supplied to the initial fibRec() call is increased.

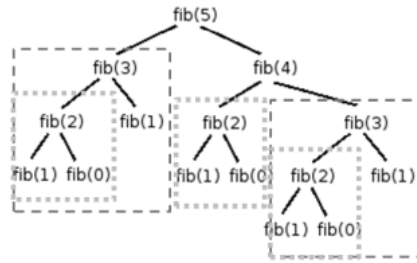


Figure 1.6. Call graph for fibRec(5)

The call graph also gives an indication of the running time for the function. Each call results in *two* further calls, causing a doubling of the computational effort at each level. Of course, once a base case is reached, the calculations stop, and the tree isn't the same height across its breadth. Nevertheless, this doubling leads to an exponential running

time of $O(2^n)$, which will bring any computer to a grinding halt when n is even modestly large.

In one respect the call graph is misleading: it suggests that the entire tree is in memory at once. This is not the case since execution progresses down one branch at a time, reaches a base case, returns, and the call frames are deleted. This means that the memory costs are related only to the length of a single branch, which will be $O(n)$. The exponential problem is confined to the running time.

The running time can be reduced to $O(n)$ by switching to an iterative solution (see Listing 1.9; fibsLib.py). The algorithm relies on the fact that the current Fibonacci number only depends on the two previous values, and so a loop using two variables to store those numbers is sufficient.

```
def fibIter(n):
  a = 0; b = 1  # F0, F1
  if n <= 0:
    return a
  elif n == 1:
    return b
  else:
    for i in range(2, n+1):
      a, b = b, a + b
    return b
```

Listing 1.9. Iterative Fibonacci

1.2.1 Memoization. Memoization is the (rather obvious) idea of storing answers to sub-calculations so that if the same calculations reappear later, then the stored values can be used instead of repeating the computations. fibRec() suffers from a great deal of recalculation, and so will benefit from memoing; the outcome is Listing 1.10 (fibsLib.py).

```
fibsStore = [0, 1]
numCalls = 0

def fibMemo(n):
  global numCalls
  numCalls += 1

  if n < len(fibsStore):
    return fibsStore[n]
  else:
    res = fibMemo(n-1) + fibMemo(n-2)
    fibsStore.append(res)
    return res
```

Listing 1.10. Memoing Fibonacci numbers

fibMemo() is still recursive, with the same structure as fibRec(), but also accesses fibsStore, a global variable that's retained in memory even after fibMemo() returns. The numCalls variable isn't needed for the memoing, and is included to total up the number of function calls made in fibMemo().

Listing 1.11 (fibMemo.py) calls fibMemo() three times: fib(50) is calculated twice, and then fib(60).

```
def useFibMemo(n):
  fibsLib.numCalls = 0
  print("\nfib of", n, "=",
      fibMemo(n))
  print("No of calls:", fibsLib.
      numCalls)
  print("fibsStore size:", len(
      fibsStore))


useFibMemo(50)
useFibMemo(50)
useFibMemo(60)
```

Listing 1.11. Using memoing

```
> python fibMemo.py

fib of 50 = 12586269025
No of calls: 99
fibsStore size: 51

fib of 50 = 12586269025
No of calls: 1
fibsStore size: 51

fib of 60 = 1548008755920
No of calls: 21
fibsStore size: 61
```

Memoization has an immediate benefit in the first call to fib(50). Instead of generating around 2^{50} calls, there's only 99. The savings continue with the second fib(50): since fibsStore is still in memory, there's no need for any calculations.

fib(60) does require new work, but only for the Fibonacci numbers between 51 and 60, which takes linear rather than exponential time.

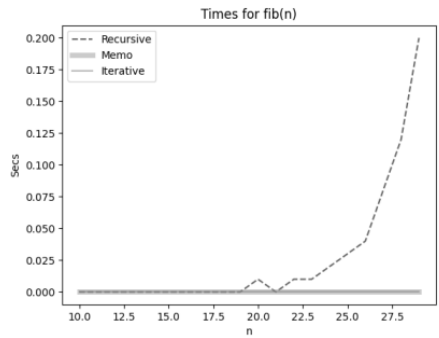


Figure 1.7. Timings for Fib variants

fibTimes.py calls fibRec(), fibIter(), and fibMemo() for a variety of values, and plots the running times. The results are shown in Fig. 1.7.

The plot for fibMemo() is linear (the same as fibIter()) even though the function is recursive. However, the curve for fibRec() has a characteristic exponential rise.

1.2.1.1 Memoization using Lazy Evaluation. Python's lazy evaluation can be utilized as a form of memoization. fibsGen() in Listing 1.12 (fibsLib.py) is a version of fibIter() using yield rather than return.

```
def fibGen():
  # generator that never stops
  current = 0 # first fibonacci numbers
```

```
  b = 1
  while True:
    yield current
    current, b = b, current+b
```

Listing 1.12. Generating Fibonacci numbers

Once the iterator has been created, it stays in memory, waking up when the next Fibonacci number is requested:

```
>>> fg = fibGen()
>>> next(fg)
0
>>> next(fg)
1
>>> next(fg)
1
>>> next(fg)
2
>>> next(fg)
3
>>> next(fg)
5
>>> next(fg)
8
```

However, iterators are *single-use* which means that `fg` can only generate a single sequence of Fibonacci numbers. For instance, we can't request fib(50) twice, or jump from fib(50) to fib(60) without generating all the intermediate numbers with calls to next().

1.2.2 Fibonacci Approximations. The Fibonacci sequence can be defined as a closed-form expression called Binet's formula (`https://en.wikipedia.org/wiki/Fibonacci_sequence#Binet's_formula`), which uses powers of the golden ratio ϕ:

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

ϕ is equal to $\frac{1+\sqrt{5}}{2}$.

One way to derive this formula is by considering the quadratic $x^2 - x - 1 = 0$. Its roots are $\frac{1\pm\sqrt{5}}{2}$, so one of them is ϕ. The other, $\frac{1-\sqrt{5}}{2}$, is sometimes called τ.

The quadratic can be rearranged as $x^2 = x + 1$ and then used to produce increasing powers of x^n in terms of multiples of x:

$$\begin{aligned} x^3 &= x \cdot x^2 = x \cdot (x+1) \\ &= x^2 + x = (x+1) + x \\ &= 2x + 1 \end{aligned}$$

$$\begin{aligned} x^4 &= x \cdot x^3 = x \cdot (2x+1) \\ &= 2x^2 + x = 2(x+1) + x \\ &= 3x + 2 \end{aligned}$$

$$\begin{aligned} x^5 &= 5x + 3 \\ x^6 &= 8x + 5 \end{aligned}$$

The prevalence of the Fibonacci numbers is quite noticeable, which lets us make the leap of faith to:

$$x^n = F_n x + F_{n-1}$$

Since ϕ and τ are roots of the quadratic, they must both satisfy x^n, permitting us to say:

$$\phi^n = F_n\phi + F_{n-1} \quad \text{and} \quad \tau^n = F_n\tau + F_{n-1}$$

Subtracting the second equation from the first equation yields:

$$\phi^n - \tau^n = F_n(\phi - \tau) + F_{n-1} - F_{n-1}$$

and so:

$$F_n = \frac{\phi^n - \tau^n}{\phi - \tau} = \frac{\phi^n - \tau^n}{\sqrt{5}}$$

Returning to the quadratic, $x^2 - x - 1 = 0$, it can also be expressed as:

$$\begin{aligned} x - 1 - x^{-1} &= 0 \\ \frac{1}{x} &= x - 1 = \phi - 1 \\ &= -\frac{1-\sqrt{5}}{2} = -\tau \\ \tau &= -x^{-1} = -\phi^{-1} \end{aligned}$$

This value for τ in terms of ϕ (they are in fact complex conjugates), can be substituted into the F_n expression to give:

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

Binet's formula is coded using floats and decimals in `fibsLib.py`, and as with the factorial approximations, we need to consider precision.

```
import math
from decTrig import *

PHI_F = (1 + math.sqrt(5)) / 2

def fibPhi(n): # Binet's formula
  return (PHI_F**n - \
      (1-PHI_F)**n)/math.sqrt(5)

def fibPhiD(n):
  return (PHI**D(n) - \
     (D(1)-PHI)**D(n))/D(5).sqrt()
```

```
>>> fibIter(13)
233
>>> fibPhi(13)
233.00000000000006
>>> fibPhiD(13)
Decimal('233.000000000000000000000000')
>>>
>>> fibIter(100)
354224848179261915075
>>> fibPhi(100)
3.542248481792631e+20
>>> fibPhiD(100)
Decimal('354224848179261915075.0000000')
```

Binet's formula doesn't involve series approximations like Stirling's formula, and so the only errors are those associated with floating point and decimal accuracy (15 digits and 28 digits respectively). These can be seen in the results produced by fibPhi() and fibPhiD().

One of the consequences of Binet's formula is that the ratio of two consecutive Fibonacci numbers tends to ϕ as n approaches infinity:

$$\lim_{x \to \infty} \frac{F_{n+1}}{F_n} = \phi$$

The $(-\phi)^{-n}$ in Binet progressively approaches 0 so the remaining ratio approaches $\frac{\phi^{n+1}}{\phi^n}$, or ϕ. This limit has nothing to do with the choice of base cases (i.e $F_0 = 0$ and $F_1 = 1$) but is due to the recurrence relation. Listing 1.13 (golden.py) emphasizes this by generating a Fibonacci series beginning with two random integers between 1 and 10000, and calculating the ratio of successive terms:

```
n1 = random.randint(1, 10000)
n2 = random.randint(1, 10000)
print(n1, n2)
sums = [n1, n2]

ratios = []
for i in range(20):
  sumCurr = sums[-1] + sums[-2]
  sums.append(sumCurr)
  ratio = sums[-1]/sums[-2]
  print(f"{i:2d} {sumCurr:10d}  {(ratio-PHI)/PHI: 15.6%}")
  ratios.append(ratio)
```

Listing 1.13. Generating Fibonacci ratios

No matter what values are chosen for n1 and n2, after 20 or so iterations, the ratio is equal to ϕ to at least six decimal places.

1.2.3 The Golden Spiral. A golden spiral is a logarithmic spiral whose growth factor is set to ϕ. It can be approximated using a Fibonacci spiral, which we'll construct in two different ways – using turtle graphics and through more conventional curve plotting. (If you're unfamiliar with Python's turtle module, please read Appendix G.)

A logarithmic spiral is distinguished from other types, such as the Archimedean spiral, by the way that the distances between its turnings increase in a geometric progression (an Archimedean spiral's turning distances remain constant). A natural way to implement this is by using a turtle to do the turning.

A golden spiral's ϕ growth factor is approximated by using the Fibonacci series (F_1, F_2, F_3, ..., F_n) since the ratio F_n/F_{n-1} approaches ϕ as n increases.

The crucial part of `fibSpiral.py` is:

```
fibs = fibs(n+1)[1:]   # drop F0 value
  :
for i in range(len(fibs)):
  dist = math.pi/180 * fibs[i] * scale
  for j in range(90):  # quarter circle
    t.forward(dist)
    t.left(1)
```

The `dist` value is used to draw a quarter circle. In effect we are approximating the growth steps of the spiral by a series of quarter turns centered inside squares with Fibonacci side lengths.

This is much more apparent in Fig. 1.8 – a box is drawn around each quarter turn and boxes' side lengths are labeled.

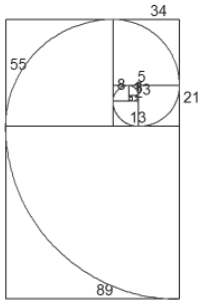


Figure 1.8. Fib squares and spiral

Although Fibonacci is a commonly utilized approximation, it isn't all that close to the actual golden spiral, as we'll see shortly.

The polar coordinate (r, θ) for a golden spiral is $r = a\phi^{2\theta/\pi}$. where θ is in radians, and a some constant. This can be compared to the polar equation for a general logarithmic spiral $r = ce^{b\theta}$, where b is the growth factor, and c some constant. They can be seen to be equivalent by changing the base of the power:

$$\begin{aligned} r &= a\phi^{2\theta/\pi} \\ \log r &= \log a + \frac{2\theta}{\pi} \cdot \log \phi \\ &= \log a + \theta b \\ r &= ae^{b\theta} \end{aligned}$$

Listing 1.14 (goldSpiral.py) doesn't use the Fibonacci series. Instead, it loads `PHI_F` from `fibsLib.py` to calculate the polar coordinates directly. These are converted to Cartesian form and plotted by Matplotlib.

```
NUM_PTS = 1000

def plotGolden(n):
  angles = linspace(0, n*math.pi, NUM_PTS)
  # Convert to (r,theta) to (x,y)
  xs = [PHI_F**(2*ang/math.pi) * math.cos(ang) for ang in angles]
  ys = [PHI_F**(2*ang/math.pi) * math.sin(ang) for ang in angles]

  plt.figure(figsize=(8, 8))
  plt.plot(xs, ys, color="blue", linewidth=2)
  plt.axis("equal")
```

```
  plt.axhline(0)
  plt.axvline(0)
  plt.title("Golden Spiral")
  plt.show()

n = float(input("n? "))
plotGolden(n)
```

Listing 1.14. Plotting the golden spiral

The result is in Fig. 1.9. The easiest-to-see differences between Figs. 1.8 and 1.9 are at the four compass points: the Fibonacci spiral is always a tangent to these directions whereas the golden spiral is pointing slightly outwards.

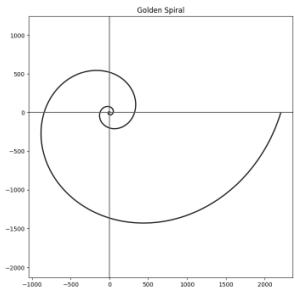


Figure 1.9. Golden spiral

1.2.4 Golden Spirals of 2D Points. Another popular way of drawing the golden spiral is by rendering multiple spirals as points filling the plane. We generate n polar coordinates, where $0 \leq i < n$:

$$r_i = (i/n)^{\frac{1}{2}} \quad \text{and} \quad \theta_i = 2\pi i\phi$$

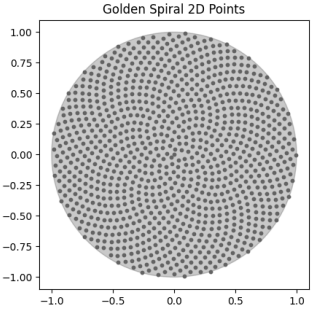
The square-root of the radius has nothing to do with ϕ but is required to distribute the points uniformly over a circular area. If the square-root wasn't applied then the points would tend to congregate near the center.

Listing 1.15 (goldScatter2D.py) also draws a pale circle behind the points to make it easier to see their relative positions.

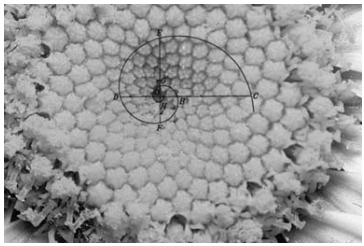
```
NUM_PTS = 1000

circle = plt.Circle((0, 0), 1, color="red", alpha=0.3)
plt.gca().add_patch(circle)

xs = [0]*NUM_PTS
ys = [0]*NUM_PTS
for i in range(NUM_PTS):
  r = (i/NUM_PTS)**0.5
  theta = 2*math.pi*(i*PHI_F)
  xs[i] = r*math.cos(theta)
  ys[i] = r*math.sin(theta)

plt.scatter(xs, ys, s=10)
plt.axis('scaled')
plt.title("Golden Spiral 2D Points")
plt.show()
```

Listing 1.15. Plotting the golden spiral as 2D points

The result is in Fig. 1.10a.



(a) The golden spiral as 2D points



(b) Spirals in flower seeds

Figure 1.10. The golden spiral

Inevitably, images like Fig. 1.10a are compared to the position of seeds on flower heads (e.g. Fig. 1.10b), but it's quite unclear whether these spirals are specifically golden (i.e. based on ϕ) or just various logarithmic spirals.

It's also worth mentioning that even if the golden spiral is most commonly used in nature, that doesn't imply that there's a Fibonacci sequence behind it. Any series that satisfies the linear recurrence

$$G_{n+2} = G_{n+1} + G_n$$

such as Lucas numbers (`https://en.wikipedia.org/wiki/Lucas_number`), would work just as well.

The strongest argument in favor of nature's use of the golden spiral is that the points in Fig. 1.10a possess the densest packing for same-sized circles within a circular region.

1.2.5 Spirals of 3D Points. We don't have to stop at two dimensions; its entirely possible to map the golden spiral to higher dimensions. As a lead-in, let's start with randomly placed points on the surface of a sphere.

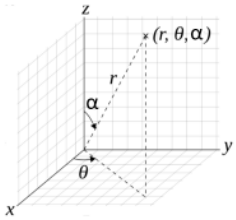


Figure 1.11. Spherical coordinates

1.2.5.1 Randomly Distributing Points on a Sphere. In spherical coordinates, r is the radius, $\theta \in [0, 2\pi]$ is the azimuthal angle, and $\alpha \in [0, \pi]$ is the polar angle, as in Fig. 1.11.

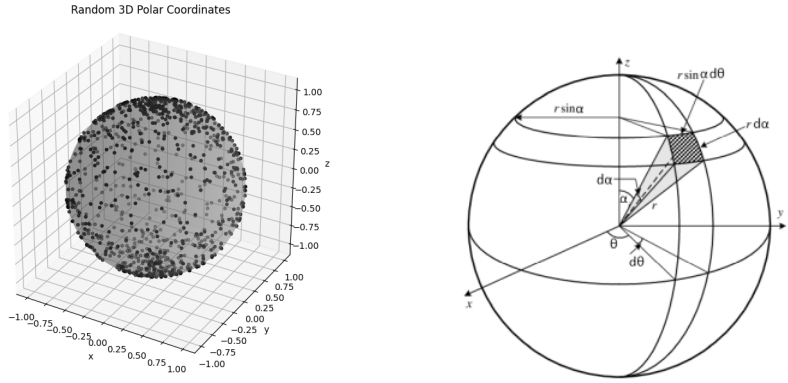
This is called the 'mathematics' convention (to distinguish it from the 'physics' convention). Also,

we've replaced the more usual polar angle variable, ϕ, with α to avoid confusion with the golden ratio.

This means that:

$$\begin{aligned} x &= r\sin(\alpha)\cos(\theta) \\ y &= r\sin(\alpha)\sin(\theta) \\ z &= r\cos(\alpha) \end{aligned}$$

A tempting way to obtain the points is to generate a uniform distribution of θ and α, then convert them to (x, y, z) points, as in randScatter3D.py



(a) Poorly randomized points on the surface of a sphere

(b) Defining dA on a sphere

Figure 1.12. Spherical Points

Fig. 1.12a shows that this approach does not lead to a good random distribution of the points. Instead, they tend to cluster around the poles ($\alpha = 0$ and $\alpha = \pi$) and are sparse around the equator ($\alpha = \pi$).

The reason for this is that the area dA of a small surface element is a square of dimension $rd\alpha$ by $r\sin(\alpha)\, d\theta$, as illustrated in Fig. 1.12b. This implies that close to the poles (when $\alpha = 0$ and $\alpha = \pi$), the surface area element determined by $d\theta$ and $d\alpha$ will get smaller since $\sin(\alpha) \to 0$. Thus, we should put less points near $\alpha = 0$ and $\alpha = \pi$ and more points near $\alpha = \pi/2$ to achieve better randomness.

Let v be a point on the unit sphere S. We want the probability density function (PDF) $f(v)$ to be constant for a uniform distribution.

The surface area of a unit sphere is 4π, which can be expressed as $\int\int_S dA = 4\pi$. Also, $f(v)\, dA$ is the probability of finding a point dA in an area about v on the sphere. Since $\int\int_S f(v)\, dA = 1$ then $f(v) = \frac{1}{4\pi}$.

However, we really want to represent points v using a parameterization of θ and α and so want the PDF in the form $f(\theta, \alpha)$. We can obtain it by equating:

$$f(v)\, dA = \frac{1}{4\pi}\, dA = f(\theta, \alpha)\, d\theta\, d\alpha,$$

We know that $dA = \sin(\alpha)\, d\alpha\, d\theta$ (assuming $r = 1$), so

$$\begin{aligned} f(\theta, \alpha)\, d\theta\, d\alpha &= \tfrac{1}{4\pi}\, dA \\ &= \tfrac{1}{4\pi}\, \sin(\alpha)\, d\alpha\, d\theta \\ f(\theta, \alpha) &= \tfrac{1}{4\pi}\, \sin(\alpha) \end{aligned}$$

Now separate this joint distribution to get the PDFs of θ and α:

$$f(\theta) = \int_0^{\pi} f(\theta, \alpha) d\alpha = \frac{1}{2\pi}$$
$$f(\alpha) = \int_0^{2\pi} f(\theta, \alpha) d\theta = \sin(\alpha)/2.$$

We see that θ is already uniform (so needs no changes), but $f(\alpha)$ scales with $\sin(\alpha)$. This means that we want more points around the equator, $\alpha = \pi/2$, which is where $\sin(\alpha)$ takes its maximum. In that case, how can we sample numbers α that follow the distribution $f(\alpha)$?

Inverse Transform Sampling allows us to sample a general PDF using a uniform random number. For this, we need the cumulative distribution function (CDF) of the random variable (α in our case).

The CDF of a random variable X is the function

$$F_X(x) = P(X \leq x)$$

The CDF can also be expressed as the integral of the PDF, $f(x)$:

$$F(x) = \int_{-\infty}^{x} f(x)\, dx.$$

This allows us to write the CDF of α as:

$$F(\alpha) = \int_0^{\alpha} f(\alpha)\, d\alpha = \int_0^{\alpha} \sin(\alpha)/2\, d\alpha = \frac{1}{2}(1 - \cos(\alpha))$$

Let U be the uniform random number in $[0, 1]$. Note that

$$Pr(U \leq F(\alpha)) = F(\alpha).$$

As F is invertible and monotone, we can preserve this inequality by writing:

$$Pr(F^{-1}(U) \leq \alpha) = F(\alpha).$$

This shows that $F(\alpha)$ is the CDF for the random variable $F^{-1}(U)$. Thus, $F^{-1}(U)$ follows the same distribution as α.

The algorithm for sampling the distribution using inverse transform sampling becomes:

- Generate a uniform random number U from the distribution $U[0,1]$. (We use Python's random.random() for this.)
- Compute $\alpha = F^{-1}(U)$ such that $F(\alpha) = U$.
- Use this α as a random number drawn from the distribution $f(\alpha)$.

In our case, $F(\alpha) = U = \frac{1}{2}(1 - \cos(\alpha))$, and so $\alpha = F^{-1}(U) = \arccos(1 - 2U)$.

This requires the change of one line in `randScatter3D.py` – the commenting out of the old assignment to alpha, replacing it with

```
alpha = math.acos(1-2*random.random())
```

The resulting globe is much more smoothly random (see Fig. 1.13).

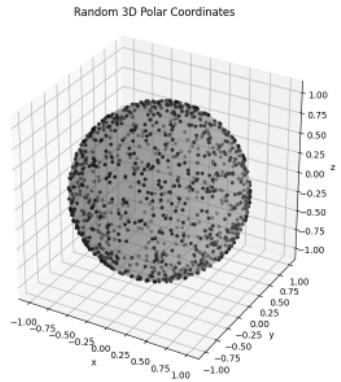


Figure 1.13. Uniformly randomized points on the surface of a sphere

1.2.5.2 Evenly Distributing Points on a Sphere. A much harder task is to *evenly* distribute points over a sphere's surface. Unfortunately, you can only do this if the points are the vertices of a regular solid. So you can exactly evenly space 4, 6, 8, 12, or 20 points, but if you want to position more than 20, then some will necessarily be closer to their neighbors than others. Of course, there are ways to approximately position the points, and the golden spiral offers both a simple, and very accurate, solution.

The implementation begins by evenly distributing n points inside a rectangle, where $0 \leq i < n$:

$$x_i = i\phi \quad \text{and} \quad y_i = i/n$$

Each (x_i, y_i) pair is mapped to the angles we used in randScatter3D.py:

$$\theta_i = \pi x_i \quad \text{and} \quad \alpha_i = \arccos(1 - 2y_i)$$

Listing 1.16 shows the part of goldScatter3D.py where the coordinates are generated. The resulting sphere is in Fig. 1.14.

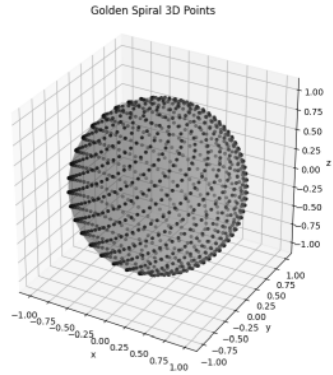


Figure 1.14. Golden spiral points on the surface of a sphere

```
xs = [0]*NUM_PTS
ys = [0]*NUM_PTS
zs = [0]*NUM_PTS
radius = 1
for i in range(NUM_PTS):
  theta = math.pi*(i*PHI_F)
  alpha = math.acos(1-2*(i/NUM_PTS))   # for uniformity
  xs[i] = radius*math.cos(theta)*math.sin(alpha)
  ys[i] = radius*math.sin(theta)*math.sin(alpha)
  zs[i] = radius*math.cos(alpha)
```

Listing 1.16. Coordinates for the golden spirals on the surface of a sphere

One way to measure the distance between points is to construct a line from the center to each one. We can measure the angle between two lines by taking their dot product.

E. B. Saff and A. B. J. Kuijlaars, 'Distributing many points on a sphere', *The Mathematical Intelligencer*, Vol. 19, No. 1, pp. 5-11, 1997 is a great paper on this topic. Another is 'Measurement of areas on a sphere using Fibonacci and latitude-longitude lattices' by Álvaro González, available at `https://arxiv.org/pdf/0912.4540.pdf`. A very nice online resource is Martin Roberts's blog post at `https://extremelearning.com.au/how-to-evenly-distribute-points-on-a-sphere-more-effectively-than-the-canonical-fibonacci-lattice/`.

1.2.6 Generalizing the Fibonacci Series. There are numerous ways that the Fibonacci series can be modified (`https://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers`). The Lucas sequence is an important one, and appears in one of the exercises at the end of this part. Another generalization is to higher order sequences, where each element is the sum of the previous n elements. The *tribonacci* numbers starts with *three* predefined terms and each subsequent term is the sum of the preceding *three*. *Tetranacci* numbers start with four terms, and each following term is the sum of the preceding four. Of course, Fibonacci numbers follow this pattern, with order 2.

These sequences start with $(n-1)$ 0's and a single 1. So the first few tribonacci numbers are:

```
0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, ...
```

The tetranacci numbers begin as:

```
0, 0, 0, 1, 1, 2, 4, 8, 15, 29, 56, 108, 208, 401 ...
```

Code for generating these sequences is in higherFibs.py. It employs the user's supplied n value to create a list filled with $(n-1)$ 0's and a 1. A new value is calculated by summing the list, then the list's data is shifted one element to the left, causing the oldest value to be dropped. The space opened up at the end is used to store the new value, and the program repeats.

1.3 The Tower of Hanoi

In 1883 the French mathematician, Édouard Lucas, unveiled 'The Tower of Hanoi' (La Tour D'Hanoï) puzzle. There are eight wooden disks with holes in their centers, stacked in order of decreasing size on one pole in a row of three (see Fig. 1.15).

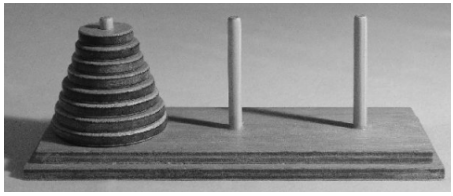


Figure 1.15. The Tower of Hanoi

The objective is to move the entire stack to one of the other poles, while following three rules:

(1) Only one disk may be moved at a time.

(2) Each move consists of taking the top disk from one of the stacks and placing it on top of another stack or on an empty pole.

(3) No disk may be placed on top of a smaller disk.

Fig. 1.16 shows the seven moves required to move three disks to another pole.

Lucas' instructions claim that the puzzle was based on an Indian legend:

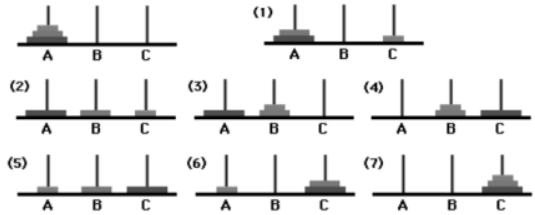


Figure 1.16. Moving a stack of three disks

> On the steps of the altar in the temple of Benares, for many, many years Brahmins have been moving a tower of sixty-four golden disks from one pole to another; one by one, never placing a larger on top of a smaller. When all the disks have been transferred the Tower and the Brahmins will fall, and it will be the end of the world.

Lucas offered a prize of ten thousand francs (about $200,000 today) to anyone who could move a tower of 64 disks by hand while following the game rules. What he didn't reveal was that the minimal number of required moves is $2^n - 1$, where n is the number of disks. So, if the eager contestant was able to move disks at a rate of one per second, it would take them $2^{64} - 1$ seconds or roughly 585 billion years to finish, which is some time after the end of the universe. Indeed, the Earth will have become uninhabitable due to the Sun's decay after a mere billion years. This constrains the problem to around a maximum of 55 disks.

A program for solving the puzzle arises naturally by expressing the solution recursively. The poles are labeled A, B, and C, with A the home of the stack of disks at the start, and the aim being to move them to pole C.

The base case, when there's just one disk, involves moving that disk directly from pole A to C.

The recursive case, involving the transfer of n disks from A to C, is divided into three stages:

(1) Move the top $n - 1$ disks from A to B. Note that this is a recursive version of the original goal with a smaller stack of disks, and using pole B as the destination. Pole C will be utilized as an auxiliary.

(2) Move the bottom disk from A to C.

(3) Move the $n - 1$ disks from B to C (using pole A as the auxiliary this time). This is another recursive task.

A representation of these three stages for $n = 4$ disks is shown in Fig. 1.17.

A key insight is that the recursive stages 1 and 3 involving $n - 1$ disks are possible because the largest disk is at the bottom of a pole, and so cannot limit the choice of moves for the other disks.

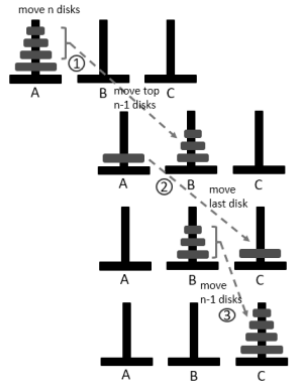


Figure 1.17. Moving n disks in three stages

The resulting code is in Listing 1.17 (hanoi.py).

```
def moves(n, src, dest, aux):
  if n == 1:
    print("Disk 1 :", src, "-->", dest)
  else:
    moves(n-1, src, aux, dest)
    print("Disk", n, ":", src, "-->", dest)
    moves(n-1, aux, dest, src)


n = int(input("n? "))
print("Moving", n, "disks from A to C")
moves(n, "A", "C", "B")
```

Listing 1.17. Recursive Tower of Hanoi

The following output replicates the steps shown in Fig. 1.16.

```
> python hanoi.py
n=? 3
Moving 3 disks from A to C
Disk 1 : A --> C
Disk 2 : A --> B
Disk 1 : C --> B
Disk 3 : A --> C
Disk 1 : B --> A
Disk 2 : B --> C
Disk 1 : A --> C
```

For larger stacks it's useful to print a textual representation of the poles state, which is done by hanoiShow.py. It stores the poles data in a Python dictionary, with the pole name as the key attached to a list of disks currently on that pole.

The output for the 3-disk puzzle is now:

```
> python hanoiShow.py
n? 3
Moving 3 disks from A to C
                    A         B         C
                    [ 3 2 1   [         [
Disk 1: A --> C   : [ 3 2     [         [ 1
Disk 2: A --> B   : [ 3       [ 2       [ 1
Disk 1: C --> B   : [ 3       [ 2 1     [
Disk 3: A --> C   : [         [ 2 1     [ 3
Disk 1: B --> A   : [ 1       [ 2       [ 3
Disk 2: B --> C   : [ 1       [         [ 3 2
Disk 1: A --> C   : [         [         [ 3 2 1
```

1.3.1 Calculating the Running Time. A recurrence relation can be used to calculate the big-Oh running time. T_n is the time to move n disks from the starting pole to the destination, and is defined as:

$$\begin{aligned} T_n &= b + 2T_{n-1}, \quad n > 1 \\ &= a, \quad n = 1 \end{aligned}$$

where a and b are constant times for a single move. The $b + 2T_{n-1}$ sum is the cost for the three stages. The recursive stages 1 and 3 each take time T_{n-1} since they are moving $n - 1$ disks. This relation can be solved by substitution:

$$\begin{aligned} T_n &= b + 2T_{n-1} \\ &= b + 2(b + 2T_{n-2}) \\ &= b + 2b + 2^2 T_{n-2} \\ &= b + 2b + 2^2(b + 2T_{n-3}) \\ &= (1 + 2 + 2^2)b + 2^3 T_{n-3} \\ &\vdots \\ &= (1 + 2 + 2^2 + \cdots 2^{n-2})b + 2^{n-1} T_1 \\ &= (1 + 2 + 2^2 + \cdots 2^{n-2})b + 2^{n-1} a \\ &= (2^{n-1} - 1)b + 2^{n-1} a \\ &= 2^{n-1}(a + b) - b \end{aligned}$$

Since a and b are constants, T_n is $O(2^{n-1}) + O(1)$, which can be simplified to T_n is $O(2^n)$, or exponential running time.

1.3.2 An Iterative Version. In "The Tower of Hanoi for Humans" [**Sto18**], Paul K. Stockmeyer describes three iterative algorithms, which he classifies as 'easy for humans' due to the rules for disk moves only utilizing a bounded amount of memory, augmented by configuration information such as disk size or color.

The third method is by Scorer, Grundy, and Smith from their paper 'Some Binary Games', *Math. Gaz.*, 28 (1944), no. 280 (July) pp.96-103. For $X, Y \in$

$\{A, B, C\}$, $(X \leftrightarrow Y)$ denotes a disk move involving poles X and Y in either direction. There is one rule:

- Move disks cyclically in the order:

$$(A \leftrightarrow B), \quad (A \leftrightarrow C), \quad (B \leftrightarrow C), ...$$

(The relative sizes of the disks on the poles determine the direction of the move.)

Note: if the number of disks is odd, then the cyclic order changes to

$$(A \leftrightarrow C), \quad (A \leftrightarrow B), \quad (B \leftrightarrow C), ...$$

Essentially, this technique first makes the move that avoids pole C, then the move that avoids pole B, then the move that avoids pole A, and so on, cycling the excluded pole in the direction $C \to B \to A \to \cdots$.

The Python implementation of this method is in `hanoiSGS.py`, and reuses the poles dictionary from `hanoiShow.py`.

The output when four disks are moved:

```
> python hanoiSGS.py
n? 4
                   A          B          C
                   [ 4 3 2 1  [          [
Disk 1: A --> B  : [ 4 3 2    [ 1        [
Disk 2: A --> C  : [ 4 3      [ 1        [ 2
Disk 1: B --> C  : [ 4 3      [          [ 2 1
Disk 3: A --> B  : [ 4        [ 3        [ 2 1
Disk 1: C --> A  : [ 4 1      [ 3        [ 2
Disk 2: C --> B  : [ 4 1      [ 3 2      [
Disk 1: A --> B  : [ 4        [ 3 2 1    [
Disk 4: A --> C  : [          [ 3 2 1    [ 4
Disk 1: B --> C  : [          [ 3 2      [ 4 1
Disk 2: B --> A  : [ 2        [ 3        [ 4 1
Disk 1: C --> A  : [ 2 1      [ 3        [ 4
Disk 3: B --> C  : [ 2 1      [          [ 4 3
Disk 1: A --> B  : [ 2        [ 1        [ 4 3
Disk 2: A --> C  : [          [ 1        [ 4 3 2
Disk 1: B --> C  : [          [          [ 4 3 2 1
```

1.4 Derangements

A *derangement* is a permutation of objects that leaves no object in its original position. For example, 21453 is a derangement, but 21543 is not because 4 has not moved.

The problem of counting derangements was first considered by Pierre Raymond de Montmort in his *Essay d'analyse sur les jeux de hazard* in 1708.

The number of derangements of n objects (OEIS sequence A000166 (`https://oeis.org/A000166`)) for small n is given in Table 1.2.

n	0	1	2	3	4	5	6	7	8	9	10
D_n	1	0	1	2	9	44	265	1,854	14,833	133,496	1,334,961

Table 1.2. The number of derangements of n objects

For instance, $D_3 = 2$, because the only derangements of a sequence of length 3, such as 123, are 231 and 312.

Another version of the problem arises when we ask for the number of ways n letters, each addressed to a different person, can be placed in n pre-addressed envelopes so that no letter is assigned the correctly addressed envelope.

Another popular variant is the *hat-check problem*, in which we count the number of ways in which n hats (let's call them h_1 through h_n) can be returned to n people (P_1 through P_n) such that no hat makes it back to its owner. In other words, we want to calculate D_n. Each person may receive any one of the $n-1$ hats that are not their own. Call the hat that the person P_1 receives h_i and then consider h_i's owner. P_i may receive either P_1's hat, h_1, or some other, so the problem splits into two cases:

(1) P_i does *not* get hat h_1, which means this case is equivalent to solving the problem with $n-1$ people and $n-1$ hats (P_1 received h_i, so the number of hats and people is reduced by 1). We can write this as D_{n-1}.

(2) P_i does get h_1. In this case the problem reduces to $n-2$ people and $n-2$ hats. (P_1 and P_i received h_i and h_1, so the number of hats and people reduces by 2). We can write this as D_{n-2}.

To summarize, the number of actions possible for P_i is the sum of the counts for the two cases, which is $D_{n-1} + D_{n-2}$. However, this must be multiplied by $(n-1)$ because any one of the $(n-1)$ people could have been chosen to be P_i.

Thus, the number of derangements D_n is

$$D_n = (n-1)(D_{n-1} + D_{n-2}) \text{ for } n \geq 2, \tag{1.1}$$

where $D_0 = 1$ and $D_1 = 0$.

There's a direct translation to Python shown in Listing 1.18 (derange.py).

```
def countDer(n):
  if n == 0:
    return 1
  elif n == 1:
    return 0
  else:
    return (n-1) * \
       (countDer(n-1) + countDer(n-2))
```

Listing 1.18. A recursive derangement counter

This code works correctly, but becomes prohibitively slow for larger values of n. More efficient encodings are needed.

1.4.1 Derivation of a Closed Form. Subtracting nD_{n-1} from both sides of Equ. (1.1) yields

$$D_n - nD_{n-1} = -(D_{n-1} - (n-1)D_{n-2})$$

Let $A_n = D_n - nD_{n-1}$, and the recurrence relation becomes

$$A_n = -A_{n-1}.$$

with $A_0 = 1$.

Since $A_0 = 1$, we get $A_1 = -A_0 = -1, A_2 = -A_1 = 1, A_3 = -A_2 = -1$, and so on. In general, $A_n = (-1)^n$.

Now we can write

$$D_n - nD_{n-1} = A_n, = (-1)^n. \tag{1.2}$$

This can be expressed as the recurrence

$$D_n = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot D_{n-1} + (-1)^n & \text{if } n > 0. \end{cases}$$

Although this is still a recursive definition, the fact that it only uses a single recursive call rather than the two calls in the original definition, suggests that it can be implemented more efficiently. Also, in a system supporting tail-recursive optimization (which is not part of Python), this code would be compiled to use iteration. In any case, this form makes it easier to carry out a manual translation (see the next subsection). The single-recursive function is shown in Listing 1.19 (derange.py).

```
def countDer2(n):
  if n == 0:
    return 1
  else:
    return n*countDer2(n-1) + (-1)**n
```

Listing 1.19. A single-recursive derangement counter

This version is not much slower than the iterative code (as we'll see below), but it may still crash if it exceeds Python's recursion depth limit, and also uses a lot of memory for stack frames.

Back to the math. Divide both sides of Equ. (1.2) by $n!$ to obtain

$$\frac{D_n}{n!} - \frac{D_{n-1}}{(n-1)!} = \frac{(-1)^n}{n!}$$

Let $B_n = \frac{D_n}{n!}$, so we have

$$B_n = B_{n-1} + \frac{(-1)^n}{n!}$$

with $B_0 = 1$. Solving this recurrence, we get

$$\frac{D_n}{n!} = B_n = \sum_{k=0}^{n} \frac{(-1)^k}{k!}$$

Multiplying both sides by $n!$ yields the derangement formula:

$$D_n = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!} \tag{1.3}$$

The sum is complex, but we can utilize the identity:

$$\lim_{x\to\infty} \sum_{k=0}^{n} \frac{x^k}{k!} = e^x$$

So when $x = -1$, we have

$$D_n \approx n! \lim_{x\to\infty} e^x = n!\, e^{-1}$$

More precisely:

$$D_n = \left[\frac{n!}{e}\right] = \left\lfloor \frac{n!}{e} + \frac{1}{2} \right\rfloor \text{ for } n \geq 1$$

where [x] is the nearest integer function and $\lfloor x \rfloor$ is the floor function.

We'll implement this with Python's factorial() and math.e, resulting in the one-liner in Listing 1.20 (derange.py):

```
def countDerApprox(n):
  return int(math.factorial(n)/math.e + 0.5)
```

Listing 1.20. An approximate derangement counter

1.4.1.1 Using Iteration. The original recursive definition is quite similar to our Fibonacci code from Listing 1.2, and so can utilize the same sort of iterative conversion. The idea is to employ two additional variables to hold the current and previous derangement values, which are used to calculate the next count. The code is in Listing 1.21 (derange.py).

```
def countDerIter(n):
  if n == 0:
    return 1
  elif n == 1:
    return 0
  prev = 0
  dr = 1
  for i in range(3, n+1):
    curr = (i-1)*(prev+dr)
    prev = dr
    dr = curr
  return dr
```

Listing 1.21. An iterative derangement counter

The translation of the single-recursive version is slightly complicated by the need to add $(-1)^n$, For efficiency's sake, this should *not* be implemented using powers; instead an additional variable, `mult`, maintains the ongoing power, which is multiplied by -1 on each iteration (see Listing 1.22; derange.py).

```
def countDerIter2(n):
  if n == 0:
    return 1
  mult = -1
  dr = 1
  for i in range(1, n+1):
    dr = i*dr + mult
    mult *= -1
  return dr
```

Listing 1.22. A faster iterative derangement counter

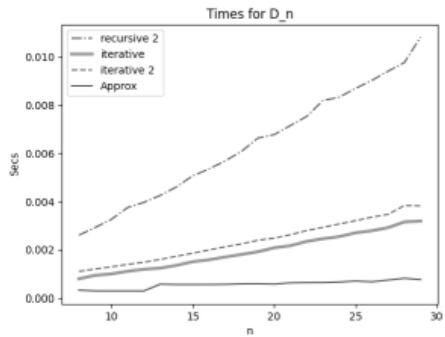


Figure 1.18. Running times for derangement

We can measure and plot the times for the five derangement functions using the timeit module that we saw back in Listing 1.1. Actually, the original recursive function (1.18) is so slow that we commented out its call, resulting in the timing plots shown in Fig. 1.18.

The timings are based on the average of 1000 calls for calculating D_8 to D_{30}.

The single-recursive function (Fig. 1.19) is the slowest but with very similar running times to the others in absolute terms. There's not much to choose between the two iterative versions, but the approximation function (Listing 1.20) is the clear winner. Incidentally, the approximation is fairly accurate, as demonstrated below:

```
>>> from derange import *
>>> countDerApprox(3)
2
>>> countDerIter(3)
2
>>> countDerApprox(25)
57062552826334675195658 24
>>> countDerIter(25)
57062552826334667623572 24
```

1.5 The Bisection Method

The following sequence was first described by Édouard Lucas (who later devised 'The Tower of Hanoi' puzzle in Sec. 1.3) and his ideas were developed further by Raoul Perrin in 1899:

$$v(0) = 3, \quad v(1) = 0, \quad v(2) = 2$$
$$v(n) = v(n-2) + v(n-3) \text{ for } n \geq 3.$$

Two recursive functions (Listings 1.23 (linRec.py) and 1.24 (linRec1.py)) print the ratio $q(n) = v(n)/v(n-1)$. The first stops as soon as $v(n-1) > 10^8$ holds, the second prints 50 terms of the sequence q(n), starting with n = 3. We'll return to Perrin numbers (`https://en.wikipedia.org/wiki/Perrin_number`) again and again in future sections.

```
def v(n0, n1, n2):
  print(f"{n1/n0:.10f}")
  if n0 <= 1E+08:
    v(n1, n2, n0+n1)  # shift left

v(2, 3, 2)
```

Listing 1.23. Perrin numbers to 1e8

```
def v(n0, n1, n2, col):
  if col > 0:
    print(f"{n1/n0:14.10f}", end ='')
    if (col-1)%5 == 0:
      print()
    v(n1, n2, n0+n1, col-1)  # shift left

v(2, 3, 2, 50)
```

Listing 1.24. 50 Perrin numbers

Output from Listing 1.24:

```
 1.5000000000  0.6666666667  2.5000000000  1.0000000000
 1.4000000000  1.4285714286  1.2000000000  1.4166666667
 1.2941176471  1.3181818182  1.3448275862  1.3076923077
```

```
1.3333333333  1.3235294118  1.3222222222  1.3277310924
1.3227848101  1.3253588517  1.3249097473  1.3242506812
1.3251028807  1.3245341615  1.3247362251  1.3247787611
1.3246492986  1.3247604639  1.3247049867  1.3247126437
1.3247288503  1.3247093499  1.3247218789  1.3247177382
1.3247164894  1.3247195193  1.3247170266  1.3247182160
1.3247180989  1.3247177043  1.3247181492  1.3247178740
1.3247179579  1.3247179924  1.3247179218  1.3247179776
1.3247179522  1.3247179536  1.3247179631  1.3247179530
1.3247179590  1.3247179573
```

If $q(n)$ converges, the limit is a root of the equation:

$$x^3 - x - 1 = 0 \tag{1.4}$$

This can be obtained by rearranging the equation,

$$v(n) = v(n-2) + v(n-3) \Rightarrow \frac{v(n)}{v(n-1)} = \frac{v(n-2)}{v(n-1)} + \frac{v(n-3)}{v(n-2)} * \frac{v(n-2)}{v(n-1)}$$

Let x be the limit of $q(n)$. Then:

$$x = \frac{1}{x} + \frac{1}{x^2}, \quad \text{or} \quad x^3 = x + 1$$

So $q(n)$ converges to a root of Equ. (1.4). For $f(x) = x^3 - x - 1$ we have $f(1) = -1$, $f(2) = 5$, so there must be a root between 1 and 2. We can visually check this in Fig. 1.19, which is generated by `plotEq.py` after supplying it with the equation of the curve, and start and end x values:

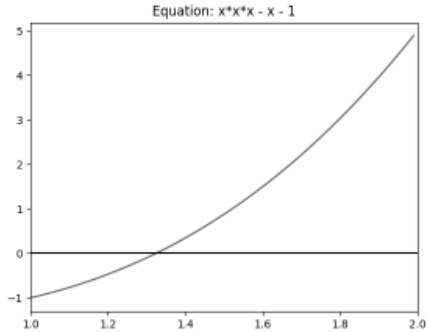


Figure 1.19. Plot of $x^3 - x - 1$ between $1 \leq x \leq 2$

```
> python plotEq.py
x equation? x*x*x - x - 1
xStart xEnd=? 1 2
```

The equation string is converted to a function by Python's eval().

We'll pass the interval [1, 2] to the recursive bisection method in Listing 1.25 (bis.py).

```
def perrin(x):  # root between 1 and 2
  return x*x*x - x - 1

def bis(fn, a, b):
  if sign(fn(a)) == sign(fn(b)):
    print(f"{a} and {b} do not span a root")
    return None, 0
  else:
    return biRec(fn, a, b, 0)

def biRec(fn, a, b, iters):
  mid = (a + b)/2
  if iters > MAX_ITERS:
    print(f"Iterations exceeded", MAX_ITERS)
    return mid
  if abs(fn(mid)) < EPS:
    return mid
  elif sign(fn(a)) == sign(fn(mid)):
    # mid is an improvement on a
    return biRec(fn, mid, b, iters+1)
  elif sign(fn(b)) == sign(fn(mid)):
    # mid is an improvement on b
    return biRec(fn, a, mid, iters+1)

if __name__ == "__main__":
  a,b = map(float, input("a b? ").split())
  root = bis(perrin, a, b)
  # root = bis(ef, a, b)
  print(f"Root = {root:.10f}")
```

Listing 1.25. Recursive bisection

```
> python bis.py
a b? 1 2
Root = 1.3247179571
```

The bisection method works because a continuous function with values of opposite signs in an interval must have a root within that interval. biRec() narrows in on the root by calculating the midpoint $mid = \frac{a+b}{2}$. This gives us two smaller intervals: (a, mid) and (mid, b). The sign of fn(mid) compared to fn(a) and fn(b) determines which of these contains the root, and biRec() is recursively called on that sub-interval.

Each call reduces the interval by half, making bisection a *linear convergence* technique that's guaranteed to find a root (if it was supplied with a starting interval that brackets the root).

Slow convergence is the main drawback compared to other root-finding approaches, so an iteration counter is used to check if the function is taking too long. The other convergence problem is if there are multiple roots within the interval, which may cause the method to move away from the root we intended to find, and converge on another one.

Since we're dealing with floats, we stop the recursion when fn(mid) is less than EPS rather than 0 at the root, which is unlikely to occur.

It's fairly simple to rewrite bis() to be iterative since the recursive calls in biRec() are at the end of the function. This *tail recursion* is translated into a loop which 'jumps' back to the top of the function rather than recursing. Exercise 2 makes use of this approach.

Bisection is one of the big three root-finding techniques, the other two being Newton-Raphson and the secant methods. Newton-Raphson is great for speed (it can converge quadratically), but requires the function's derivative, and does not guarantee that the method will converge. The secant method gets around Newton-Raphson's derivative issue by estimating it using a secant line (see Fig. 1.20), but has the same convergence problem.

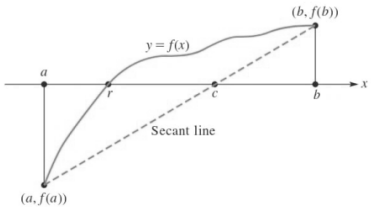


Figure 1.20. Secant method

1.6 Gambling Boldly

My current fortune is $x < 1$ (perhaps 1 stands for \$1 million), and I decide to start gambling *boldly* with the aim of increasing x to 1.

If $0 < x \leq 0.5$ then I stake x, all I have. If I win, my fortune will jump to $2x$, although a loss will wipe me out. If $0.5 < x < 1$, then I bet $1 - x$. If I win, then my fortune increases to $x + (1 - x) = 1$, and I retire happy. Also, if I lose, I still have $x - (1 - x) = 2x - 1$ left.

Suppose in one round my chance of winning is p, my chance of losing q, with $p + q = 1$. Let $f(x)$ be the probability of eventual success under these bold gambles starting with x. Then we have:

$$f(0) = 0, f(1) = 1, \quad f(x) = \begin{cases} p \cdot f(2x) & 0 < x \leq \frac{1}{2}; \\ p + q \cdot f(2x - 1) & \frac{1}{2} < x < 1. \end{cases} \tag{1.5}$$

The equation is nonlinear, which can be expressed separately as a function $d()$ defined on $[0,1]$:

$$d(x) = 2x - \lfloor x \rfloor = \begin{cases} 2x & : \ x \in \left[0, \frac{1}{2}\right) \\ 2x-1 & : \ x \in \left[\frac{1}{2}, 1\right) \end{cases}$$

d() is sometimes called the *doubling function*.

Another way to think about $f(x)$ is to treat x as a binary number. The function left-shifts the number by one bit then drops the integer part if the result is > 1.

Our program based on Equ. (1.5) is in Listing 1.26 (bold.py). The function is plotted for several values of p (see Fig. 1.21), with x ranging between 0 and 1.

```
''
def f(x,p):
  if x <= 0:
    return 0
  elif x >= 1:
    return 1
  elif x <= 0.5:
    return p*f(2*x,p)
  else:
    return p + (1-p)*f(2*x-1,p)
```

Listing 1.26. Gambling boldly by doubling

It can be shown that gambling boldly is optimal for $p < 0.5$ (i.e. in an unfair game). For $p = 0.5$ the gambling strategy has no influence on $f(x)$. For $p > 0.5$ bold gambles are bad; you should play as timidly as the rules of the game permit.

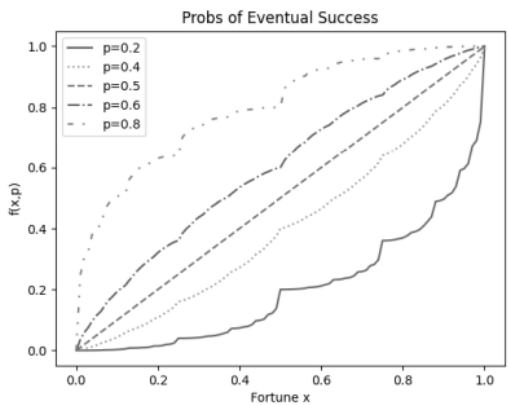


Figure 1.21. Probabilities of success for bold gambles for various p's

The mathematical analysis is explained in detail in the classic text by Lester E. Dubins and Leonard J. Savage [**DS76**]. Less technical sources are P. Billingsley: "The Singular Function of Bold Play", *Am. Scientist*, 71(1983), 392-397 and Rényi [**RÉ87**]. An online explanation begins at `https://stats.libretexts.org/Bookshelves/Probability_Theory/Probabil`

ity_Mathematical_Statistics_and_Stochastic_Processes_(Siegrist)/13%3A_Games_of_Chance/13.08%3A_The_Red_and_Black_Game. It is often called a *red and black* game, after the color bets in roulette.

1.7 The Josephus Problem

During the Jewish rebellion against Rome (A.D. 70), forty men were trapped in a cave, and to avoid a life of slavery they agreed upon mutual destruction. They would stand in a circle numbering themselves from 1 to 40. Every seventh person was then to be killed until only one was left, who would then commit suicide. We know the story from the only survivor, the historian Flavius Josephus, who obviously didn't carry out that last step.

This story is the source for the **Josephus problem**: n people are arranged in a circle, numbered 1 to n. Every k-th person is removed, with the circle closing up after each removal. What is the number $f(n)$ of the last survivor? What's the number of the s-th removed person?

The problem is fairly simple for $k = 2$ since we can express $f(2n)$ and $f(2n+1)$ in terms of $f(n)$. In Fig. 1.22a, with $2n$ people around the circle, we eliminate numbers $2, 4, ..., 2n$, and are left with $1, 3, ..., 2n-1$, which are then renumbered as 1 to n.

In Fig. 1.22b, with 2n+1 people, we eliminate numbers $2, 4, ..., 2n, 1$, and are left with $3, 5, ..., 2n+1$ who are renumbered to $1, 2, ..., n$.

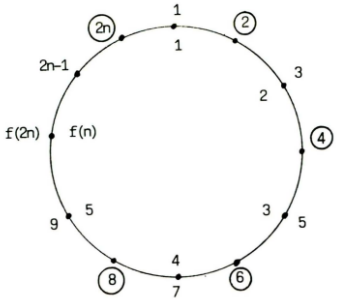
Since $f(n)$ denotes the last survivor in the inner circle, we see that his original number (on the outer circle) is:

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

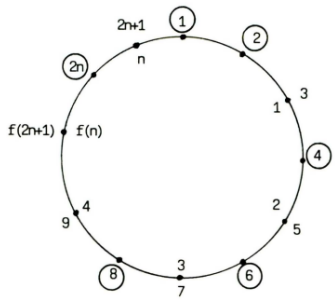
In addition we have $f(1) = 1$.

These three $f()$ equations translate into the recursive function in Listing 1.27 (jos.py). The use of // ensures that any decimal part of the division is omitted.

```
def f(n):
  if n < 3:
    return 1
  elif (n % 2 == 1): # odd
    return 2*f(n//2)+1
  else:
    return 2*f(n//2)-1

if __name__ == "__main__":
  print("Josephus Problem for n=2..40 and k = 2")
  print(" n: position of survivor at end")
  print("         ", end ='')
  for i in range(2, 41):
    print(f"{i:2d}: {f(i):2d}; ", end = ' ')
    if i%5 == 0:
```



(a) Removals using $f(2n) = 2f(n) - 1$



(b) Removals using $f(2n+1) = 2f(n) + 1$

Figure 1.22. Josephus circles

```
    print()
print()
```

Listing 1.27. Recursive Josephus

Fig. 1.23 suggests that the recursion tree depth is $O(\log_2 n)$ since each f() call roughly halves the number of people left until a single person remains.

The wikipedia page for the problem (`https://en.wikipedia.org/wiki/Josephus_problem`) includes a useful table for various group sizes, n, and step sizes, k. Looking at the values in the $k = 2$ column confirms that the following output is correct:

```
> python jos.py
Josephus Problem for n=2..40 and k = 2
 n: position of survivor at end
          2:  1;   3:  3;   4:  1;   5:  3;
 6:  5;   7:  7;   8:  1;   9:  3;  10:  5;
11:  7;  12:  9;  13: 11;  14: 13;  15: 15;
16:  1;  17:  3;  18:  5;  19:  7;  20:  9;
21: 11;  22: 13;  23: 15;  24: 17;  25: 19;
26: 21;  27: 23;  28: 25;  29: 27;  30: 29;
31: 31;  32:  1;  33:  3;  34:  5;  35:  7;
36:  9;  37: 11;  38: 13;  39: 15;  40: 17;
```

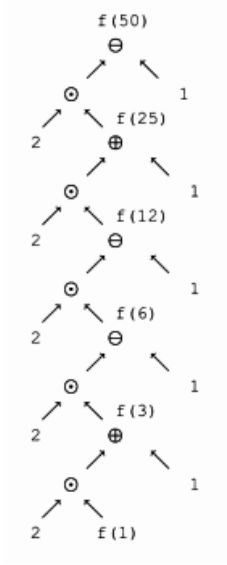


Figure 1.23. Execution of f(50)

Note that the survivor is the first person in the circle (i.e. $f(n) = 1$) whenever $n = 2^m$.

For $k = 2$, there's a simple expression for the number $f(n)$ of the survivor. For arbitrary n, let m be the largest

integer such that $2^m \leq n$. We can then write $n = 2^m + (n-2^m)$. Now remove the people numbered $2, 4, 6, ..., 2(n-2^m)$, leaving 2^m people in the circle. By the above result, the first one of these 2^m people will survive. The place number of this person is $2(n-2^m)+1$. Hence, for $k = 2$, the number of the last survivor is:

$$f(n) = 2(n-2^m)+1,$$

where 2^m is the largest power of $2 \leq n$. For example, for $n = 1988 = 1024 + 964$, then $f(n) = 2 * 964 + 1 = 1929$, which we can confirm with Python's IDE:

```
>>> from jos import *
>>> f(1988)
1929
```

If n is encoded in binary, then $f(n)$ can be computed by transferring the first digit of n to the end:

$$n = 1b_1b_2...b_m = 2^m + (n-2^m) \Rightarrow b_1b_2...b_m1 = 2(n-2^m)+1 = f(n).$$

An excellent, detailed discussion of the Josephus problem can be found in section 1.3 of Graham [**GKP94**], while the most extensive account is by Ahrens [**Ahr01**]. The surprising depth of the Josephus permutation is treated in Herstein and Kaplansky [**HK78**].

1.7.0.1 Finding the s-th Eliminated Person. For arbitrary k, the number x of the s-th eliminated person can be determined by means of Listing 1.28 (joseph1.py), which comes from problem 1.3.3-33 in Knuth, Vol. 1 [**Knu97**].

```
n,k,s = map(int, input("n k s=? ").split())
x = k*s
while x > n:
  x = int((k*(x-n)-1)/(k-1))
print(f"The {s}th eliminated person is labeled #{x}")
```

Listing 1.28. Calculating elimination position

For $k = 2$, $k/(k-1) = 2$, and the statement in the while-loop becomes $x = 2 * (x-n) - 1$. An example (which is simple enough to check manually):

```
> python joseph1.py
n k s=? 50 2 5
The 5th eliminated person is labeled #10
```

Consider a person P whose initial number in the Josephus circle is x_1, and whose number after the i-th round of eliminations is x_i. Assume the original circle is of size n, and the elimination step is k. We know x_i and want to determine x_1.

If P wasn't eliminated during the i-th cycle, then:

$$x_{i+1} = x_i + n - \left\lfloor \frac{x_i}{k} \right\rfloor. \tag{1.6}$$

The number of people eliminated during the x_i round was $\lfloor x_i/k \rfloor$, and there remains $n - \lfloor x_i/k \rfloor$ lucky survivors. After counting off these people we get back to P.

Now we form the sequence $x_1, x_2, ...$, until we reach an x_j which is divisible by k; that person will become the (x_j/k)-th eliminated. Thus the last number of the s-th eliminated person is ks. We must solve Equ. (1.6) for x_i, so we can work backwards to find his original number x_1. Now:

$$\frac{x_i}{k} = \left\lfloor \frac{x_i}{k} \right\rfloor + e, \quad \text{with } \frac{1}{k} \le e \le \frac{k-1}{k}; \tag{1.7}$$

$e \neq 0$, else x_i would be a multiple of k, and the person would have already been eliminated in the i-th round. From Equs. (1.6) and (1.7) we get:

$$x_i = \frac{(x_{i+1} - n)k}{k-1} - \frac{k}{k-1}e = \frac{(x_{i+1} - n)k - 1}{k-1} - \frac{k}{k-1}(e - \frac{1}{k})$$

By Equ. (1.7), the term subtracted on the right is between 0 and $\frac{k-2}{k-1}$. Since x_i is an integer, it is the larger integer part of the first term:

$$x_i = \left\lfloor \frac{(x_{i+1} - n)k - 1}{k-1} \right\rfloor = x_{i+1} - n + \left\lfloor \frac{x_{i+1} - n - 1}{k-1} \right\rfloor. \tag{1.8}$$

To find the original number of the s-th eliminated person, set $x_{i+1} = ks$ and apply Equ. (1.8) repeatedly until we get a number $\le n$; this is the number we are looking for.

The reader should try out this procedure by forming the sequence $x_1, x_2, ...$ and obtain the inverse sequence by means of Equ. (1.8). Suggestion: use $n = 18, k = 3, x_1 = 7$. Listing 1.28 reports:

```
> python joseph1.py
n k s=? 18 3 7
The 7th eliminated person is labeled #4
```

1.7.0.2 Josephus Iteratively. The cyclic nature of the Josephus problem means that iteration is probably a better match for coding it than recursion. Listing 1.29 (josIter.py) is more general than Listing 1.27 since it can process different step sizes, and it also prints a list of all the eliminated numbers. The 'last man standing' is the final number in the list.

```
def josIter(n, k):
  people = list(range(1, n+1))
  elims = []      # ID numbers of
      eliminated
  pos = 0
  while people:
    pos = (pos + k - 1) % len(
        people)
    elims.append(people.pop(pos))
  return elims
```

Listing 1.29. Iterative Josephus

```
> python josIter.py
n k? 18 3
[3, 6, 9, 12, 15, 18, 4, 8, 13,
17, 5, 11, 1, 10, 2, 16, 7, 14]

> python josIter.py
n k? 16 7
[7, 14, 5, 13, 6, 16, 10, 4, 2,
1, 3, 9, 15, 8, 11, 12]
```

1.7.0.3 Josephus Visualized. An understanding of the Josephus problem is greatly enhanced by having access to a visualization tool, which you can find in josVis.py. As an example, consider the problem of removing every 3rd person in a group of 18. What's the number of the seventh person to be eliminated? Listing 1.28 reports:

```
> python joseph1.py
n k s=? 18 3 7
The 7th eliminated person is labeled #4
```

This can be pictured by running `josVis.py` with $n = 18$ and $k = 3$. The animation progresses a cycle at a time, waiting for the user to press 'n' to continue. The seventh removal occurs in the second round, to person #4:

```
> python josVis.py
n k=? 18 3
Starting at ID 1
Removed: 3 6 9 12 15
At ID 16
Removed: 18 4 8 13 17
At ID 1
```

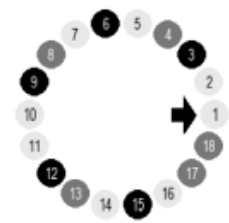


Figure 1.24. Josephus visualization

This is depicted in Fig. 1.24. Light gray circles represent people who are still alive, black circles are for people eliminated in earlier cycles, and dark gray circles are those removed during the cycle that has just finished.

We won't explain josVis.py here since the animation techniques using Pygame (`https://www.pygame.org/docs/`) aren't relevant to this problem.

Exercises for Sections 1.1 to 1.7

(1) The sequence $f(n)$ is defined by $f(1) = f(2) = f(3) = f(4) = 1$ and $f(n) = 4f(n-1) + 3f(n-2) + 2f(n-3) + f(n-4)$ for $n > 4$.

a) Write a program which prints f(20)/f(19). Hint: use the mapping $(a, b, c, d) = (b, c, d, a + 2b + 3c + 4d)$.

b) Using Listing 1.25 find the maximal root of $x^4 - 4x^3 - 3x^2 - 2x - 1 = 0$, and compare with f(20)/ f(19).

(2) Rewrite Listing 1.25 so that it works for functions increasing *or* decreasing from a to b. Also, try to remove the recursion.

(3) Find by means of Listing 1.25 the smallest positive solution of a) $x - \cos x = 0$; b) $x \ln x = 1$; c) $xe^x = 1$; d) $\tan x = x$.

(4) The equation $x^3 - 3x + 1 = 0$ has three real roots. Find them using Listing 1.25.

(5) For this and the following problems, fib(n) = f(n) and define the Lucas sequence as $g(0) = 2, g(1) = 1, g(n) = g(n-1) + g(n-2), n \geq 2$.

a) Write a program which for input n finds $g(n) - 5f^2(n)$.

b) Conjecture and prove, for instance by induction.

(6) Write a program which for input n tests if the following formulas are true:

a) $g(n) = f(n-1) + f(n+1)$;

b) $f^2(n+1) - f(n)f(n+2) = (-1)^n$.

(7) Let $f(x)$ be the probability of eventual success in the bold gamble, starting with fortune x. Find exactly

a) $f(1/3)$; b) $f(2/5)$; c) $f(1/1984)$ for $p = q = 1/2$.

(8) Here's another solution of the Josephus problem for any k by Domoryad [**Dom63**].

The number t of the s-th eliminated person can be found by means of the following algorithm: set $x = 1 + k(n-s)$ and $q = k/(k-1)$. Then generate the "integer" geometric sequence $x(1) = \lceil x \rceil, x(2) = \lceil qx(1) \rceil, x(3) = \lceil qx(2) \rceil, ...$, where $\lceil y \rceil$ denotes the ceiling of y, i.e. the smallest integer $\geq y$. If a is the largest term $\leq nk$ then $t = 1 + kn - a$.

Translate this algorithm into a program.

(9) The number $f(n)$ in the Josephus problem for $k = 2$ can also be found as follows: write n in binary, replace each 0 by -1 and you get $f(n)$. For instance, $n = 50 = 110010$ implies $f(n) = 11-1-11-1 = -1+2-4-8+16+32 = 37$. Show this.

(10) Show that for $k = 2$ in the Josephus problem the last remaining person has the number $f(n) = 1 + 2n - 2^k$ with $k = \lfloor \log_2 n \rfloor + 1$.

(11) *Duplication Formula for the Fibonacci Sequence.* Let $u(1) = 0, u(2) = 1, u(n) = u(n-1) + u(n-2)$ for $n \geq 3$. That is, u(n) = fib(n-1). Prove by induction:

$$\begin{array}{lrcll} (i) & u(2n) & = & u^2(n) + u^2(n+1), & n > 1 \\ (ii) & u(2n+1) & = & 2u(n) \cdot u(n+1) + u^2(n+1), & n \geq 1. \end{array}$$

Construct a fast recursive function based on (i) and (ii). It requires only $O(\log n)$ steps instead of $O(n)$ in fibIter() and $O(2^n)$ in fibRec(). Time the speeds of fibIter() and this function.

Remark: By means of matrices one can show that similar duplication formulas exist for all linear difference equations.

(12) Write a program which prints the sequence $J(n, k)$ of eliminations in the Josephus problem. For instance, $J(8, 3) = (3, 6, 1, 5, 2, 8, 4, 7)$. This sequence is called the *Josephus permutation*.

(13) Let $H_n = 1 + 1/2 + 1/3 + ... + 1/n$ be the harmonic numbers.

a) Find H_{10000} by adding terms from left to right.

b) Find the same sum by adding terms from right to left.

c) Which sum is more accurate and why?

(14) Find by experimentation the sum

```
round(n/2) + round(n/4) + round(n/8) + ...
```

for positive integers n. Try to prove your observations.

(15) a) Find all pairs (x, y) of positive integers from 1...100, which satisfy

$$|x^2 - xy - y^2| = 1.$$

b) How might you generate all the solutions.

(16) How many of the 6-digit blocks 000000 to 999999 have the property that the sum of the first three digits equals the sum of the last three digits. (The idea here is to reduce computation as much as possible.)

(17) The *stutter* function. Translate the functional equation

$$f(0) = 0,\ f(1) = 1,\ f(x) = \begin{cases} f(2x)/4 & \text{for } 0 < x < \frac{1}{2} \\ \frac{3}{4} + f(2x-1)/4 & \text{for } \frac{1}{2} \leq x < 1. \end{cases}$$

into a recursive program. (Theoretically, the program should stop only if x is a finite binary fraction, although the equation determines $f(x)$ uniquely for

all rational x. It can not determine $f(x)$ for irrational values unless we specify the additional condition that f be monotonic. After studying Sec. 2.1 you will be able to translate x and $f(x)$ into binary, and you'll see the reason for the function's name.

(18) Investigate how the recurrence relation:

$$x_n = x_{n-1} - \frac{2}{9}x_{n-2}$$

behaves with various starting numbers for x_0 and x_1. Verify that the ratio x_n/x_{n-1} gradually approaches 2/3 unless the initial values are in the ratio 1/3, in which case it stays at that value. Try starting with values very nearly in the ratio 1/3 to see what happens.

(19) *Factorial Sum List*. Given a number, calculate the sum of the factorials of its digits. Repeat with the new number. Stop when a repetition occurs. For example: $25 \to 122 \to 5 \to 120 \to 4 \to 24 \to 26 \to 722 \to 5044 \to 169 \to 3636001 \to 1454 \to 169$. For more information on *factorions*, see `https://en.wikipedia.org/wiki/Factorion`.

(20) Write a function that returns a count of trailing zeros in $n!$. For example, 5! ends with 1 zero, 20! has 4 zeros, and 100! finishes with 24.

(21) The Ackermann function takes two arguments, each of which can be assigned any nonnegative integer. Its definition:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

Find A(1,2) and A(1,3).

The function exhibits a phenomenal rate of growth. While the early values aren't that impressive, you'll be hard-pressed to obtain A(4, 1). Our direct translation to recursive code crashed due to a lack of recursion depth, and an iterative version took 20 minutes to finish.

Nevertheless, the Ackermann function is *well defined* in the sense that on a machine with enough memory it will always return a result. That is not the case in the next exercise.

(22) The following recursive definition, despite appearances, is not a well defined function:

$$G(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + G(\frac{n}{2}) & \text{if } n\%2 = 0 \\ G(3n-1) & \text{if } n\%2 = 1 \text{ and } n > 1 \end{cases}$$

Translate the definition into a function, and try running it on a range of positive integers. Resilient code will need to make use of Python exception handling.

Execute G(5) by hand and keep simplifying the expression. You'll end up with $G(5) = 3 + G(5)$, which is suggesting that $0 = 3$. Since the assumption that G is a function has led to a false statement, it follows that G is not a function.

(23) *The Catalan numbers* occur in various counting problems, often involving recursively defined objects. They are named after the French-Belgian mathematician Eugène Charles Catalan (1814-1894) even though they had appeared in the work of Leonhard Euler 70 years before, and were known to Chinese mathematicians before that.

The first Catalan numbers for $n = 0, 1, 2, 3, 4, , 5, ...$ are $1, 1, 2, 5, 14, 42, ...$

The recurrence relation:

$$C_0 = 1 \quad \text{and} \quad C_n = \sum_{i=1}^{n} C_{i-1} C_{n-1} \quad \text{for } n > 0$$

A non-recursive expression:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

An asymptotic approximation:

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

Implement theses definitions and compare their results.

Catalan numbers turn up in a surprisingly variety of problems, including counting the number of binary bracketings of n parentheses, the number of states possible in an n-flexagon, the number of different diagonals possible in a frieze pattern with n+1 rows, polygon triangulation, the number of rooted binary trees with n internal nodes, the number of mountains which can be drawn with n upstrokes and n downstrokes, and the number of non-crossing handshakes possible across a round table between n pairs of people. *Enumerative Combinatorics*, Vol. 1 by Richard P. Stanley [**Sta12**] describes 66 different interpretations, and a few are explained at `https://en.wikipedia.org/wiki/Catalan_number`.

(24) According to Zeckendorf's theorem (`https://en.wikipedia.org/wiki/Zeckendorf%27s_theorem`), any positive integer can be expressed as a sum of distinct non-consecutive Fibonacci numbers. Write a function that tests this theorem, as shown in the examples in Table 1.3.

n	Sum
53	[34, 13, 5, 1]
25	[21, 3, 1]
31	[21, 8, 2]
3009	[2584, 377, 34, 13, 1]

Table 1.3. Testing Zeckendorf

(25) *The Tower of Hanoi with an adjacency requirement.* Suppose that the overworked Brahmins are restricted to moving a disk only to an *adjacent* pole. Assume poles A and C are at the two ends of the row and pole B is in the middle.

Work out the number of moves required for 1, 2, 3, and 4 disks, and use this information to help you write a recurrence relation. Translate this relation into a function.