# 3

# Probability

Before tackling this chapter, it may be advisable to have a look at Appendix A if you're probability skills need a refresh.

## 3.1 A Pseudo-Random Number Generator

A pseudo-random number generator (PRNG) creates a sequence of numbers that appear to be random. The algorithm starts with a seed number, and if we let $R(t)$ represent the number returned by the PRNG in step $t$, then the seed is $R(0)$. An arithmetic function is applied to the seed to get the first number R(1) in the pseudo-random sequence. To get the next, we apply the function to $R(1)$, producing $R(2)$. This process continues for $R(3)$, $R(4)$, ..., for as long as we need "random" numbers.

One of the simplest algorithms is the *Lehmer PRNG* which employs the following difference equation:

$$R(t) = a \cdot R(t-1) \bmod m$$

where $m$ is a prime number and $a$ an integer between 1 and $m-1$.

Care must be taken when choosing values for $m$ and $a$ otherwise the generated sequence may be very far from random. For example, if we set $m = 13$ and $a = 5$, then the sequence produced is 5, 12, 8, 1, 5, 12, 8, ....

Listing 3.1 (lehmer.py) utilizes $m = 2^{31} - 1$ and $a = 16807$, and includes a simple visual check of the resulting randomness by grouping the generated sequence into 100 histogram buckets which should exhibit a uniform distribution.

```
n = 10000
```

```
MOD = 2**31 - 1    # a prime number
CONST = 16807      # an integer between 1 and MOD-1

def lehmer(r):
  # Compute pseudo-random number using a Lehmer PRNG
  return (CONST * r) % MOD

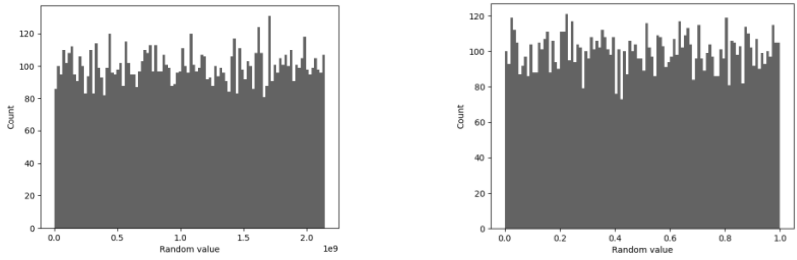
def randomSeq(n, seed):
  r = seed
  rands = []
  for i in range(n):
    # r = lehmer(r)
    r = random.random()
    rands.append(r)
  return rands

seed = datetime.now().timestamp()
print("seed:", seed)
samples = randomSeq(n, seed)
plt.hist(samples, 100)
plt.xlabel("Random value")
plt.ylabel("Count")
plt.show()
```

Listing 3.1. Display a histogram generated by the Lehmer PRNG

Note that a date timestamp is used as the seed. This will be different each time the program is called, which enhances the sequence's randomness.

Typical output is shown in Fig. 3.1a.




(a) The Lehmer PRNG (b) random.random()

Figure 3.1. Random histograms

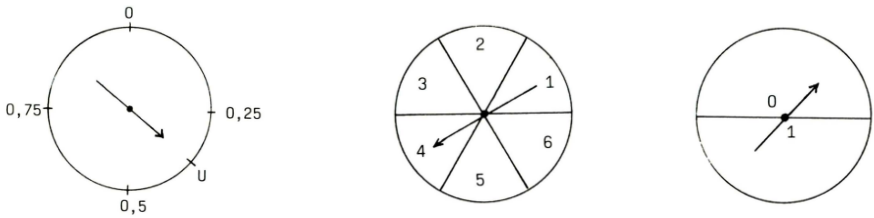
The data is fairly uniformly distributed about 100, which is a result of generating 10000 numbers and putting them into 100 buckets.

It's interesting to compare this output with Python's random(), which only requires the call to lehmer() in Listing 3.1 to be replaced by random.random(). The resulting plot is shown in Fig. 3.1b.

There's no apparent difference between Figs. 3.1b and 3.1a apart from the fact that random() generates values in the range $[0, 1)$ whereas lehmer() returns values in $[0, m-1)$.

In practice, there's really no need for us to write our own PRNG, since Python's random module already contains an excellent range of functions.

## 3.2 Python's PRNGs




(a) Uniform (b) A fair die (c) A fair coin

Figure 3.2. Types of spinners

Python's random module implements PRNGs for various distributions. The most important for us will be

- **random.random()**: produces a random float $n$ uniformly distributed in the *half-open* range $0.0 \leq n < 1.0$ (Fig. 3.2a). *Uniformly distributed* means $n$ falls into the interval $[0, 1)$ with a probability equal to the width of that interval.
- **random.uniform(a,b)**: returns a random float $n$ uniformly distributed in the *closed* range $a \leq n \leq b$.
- **random.randint(a,b)**: returns a random integer $n$ such that $a \leq n \leq b$. For example, random.randint(1,6) mimics the roll of a fair die (Fig. 3.2b).
- **random.choice(seq)**: returns a random element from the supplied sequence. For instance, random.choice([0,1]) replicates the tosses of a fair coin with its faces labeled as 0 and 1 (Fig. 3.2c).

For integers, there's also functions to generate a random permutation of a list, and random sampling without replacement. On the real line, there are functions to compute normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. It's well worth your time to have a quick look through the module's documentation at `https://docs.python.org/3/library/random.html`.

Almost all of the module functions depend on random(), which produces 53-bit precision floats with a period of $2^{19937} - 1$. The underlying implementation in C is of the Mersenne Twister, one of the most extensively tested PRNGs in existence (`https://en.wikipedia.org/wiki/Mersenne_Twister`). However, since it's deterministic, it's unsuitable for cryptographic purposes, for which Python offers the `secrets` module.

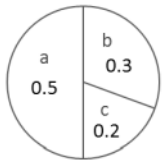



Figure 3.3. Weighted choices

A 'spinner' type that we'll need a few times is *weighted choice*. For example, we wish to choose from the set $\{a, b, c\}$ with the probability distribution $\{0.5, 0.3, 0.2\}$ (see Fig. 3.3).

This can be implemented as in Listing 3.2 (`choices.py`) using a variant of random.choices() which takes a list of weights for driving the selection. The optional keyword argument *k* allows multiple requests to be made at once.

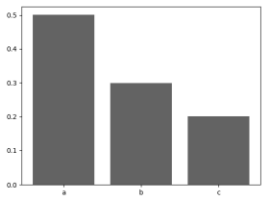


Figure 3.4. Weighted choices plotted

Listing 3.2 also illustrates the use of collections.Counter() for creating a dictionary of counts for each sample (in this case, 'a', 'b', and 'c'). The plot (Fig. 3.4) shows that the samples are distributed with the requested probabilities.

```
pop = ['a', 'b', 'c']
probs = [0.5, 0.3, 0.2]
size = 10**6
samples = random.choices(pop, probs, k=size)
      # get a million samples
countDict = collections.Counter(samples)
plt.bar(countDict.keys(),
       [v/size for v in countDict.values()])
plt.show()
```

Listing 3.2. Plotting weighted choices

## 3.3 Random Walks

The code in the following programs use variants of random.choice() to perform walks on a line, the plane, and in 3D space, starting at the origin.

```
def rWalk1D(n):
  ts = [0]
  y = 0
  locs = [y]
  for i in range(1, n+1):
```

```
    ts.append(i)
    y += random.choice([-1,1])
    locs.append(y)
  return ts, locs
```

Listing 3.3. Random 1D walk

In Listing 3.3 (rWalk1D.py) the choice is between -1 and 1 which forces a step of 1 unit at each iteration. Typical output is in Fig. 3.5.




Figure 3.5. Plotted random 1D walk

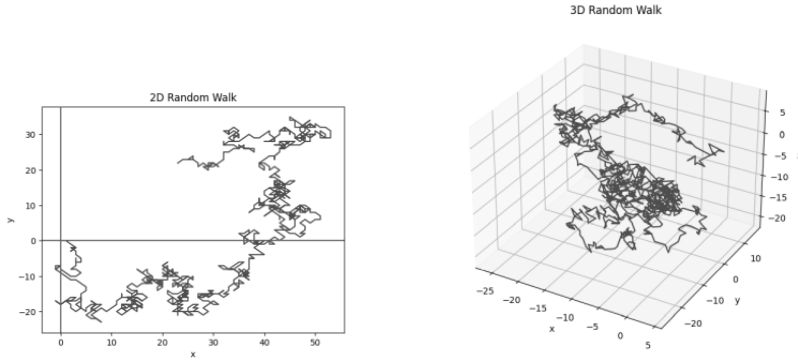
A 2D walk (rWalk2D.py) looks more interesting if the changes to the (x, y) coordinate can be -1, **0**, or 1.

```
def rWalk2D(n):
  x = 0
  xs = [x]
  y = 0
  ys = [y]
  for _ in range(1, n+1):
    x += random.choice([-1,0,1])
    xs.append(x)
    y += random.choice([-1,0,1])
    ys.append(y)
  return xs, ys
```

Listing 3.4. Random 2D walk

Typical output is shown in Fig. 3.6a.

A 3D random walk is analogous, with choice() also modifying a z coordinate. Typical output of rWalk3D.py is shown in Fig. 3.6b.




(a) 2D walk (b) 3D walk

Figure 3.6. Plotted random walks

## 3.4 Problems Involving Random 1D Walking

A 'frog' randomly jumps along the x-axis, starting from the $x = i$. At each time interval, it moves one integer to the right with probability $p$ or one integer to the left with probability $q = 1 - p$ (see Fig. 3.7). The journey stops when it lands on one of the *absorbing* points $x = 0$ or $x = N$.

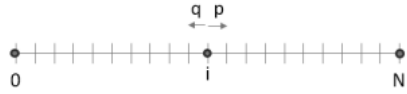



Figure 3.7. Random walk with two absorbing barriers

Let $P_i$ be the probability that starting from $x = i$, the frog reaches $x = N$. It may move either to the right (to $P_{i+1}$) with a probability $p$ or to the left (to $P_{i-1}$) with a probability of $q$. Therefore:

$$P_i = pP_{i+1} + qP_{i-1} \tag{3.1}$$

Also, $P_0 = 0$ (there's no chance of getting to $N$ when you've been absorbed at $x = 0$) and $P_N = 1$.

Since $p + q = 1$, the left-hand side of Equ. (3.1) can be re-written as

$$pP_i + qP_i = pP_{i+1} + qP_{i-1}$$

yielding

$$P_{i+1} - P_i = \frac{q}{p}(P_i - P_{i-1})$$

In particular, $P_2 - P_1 = (q/p)(P_1 - P_0) = (q/p)P_1$ (since $P_0 = 0$), so that $P_3 - P_2 = (q/p)(P_2 - P_1) = (q/p)^2 P_1$, and more generally

$$P_{i+1} - P_i = (\frac{q}{p})^i P_1, \quad 0 < i < N.$$

Thus

$$\begin{aligned} P_{i+1} - P_1 &= \textstyle\sum_{k=1}^{i}(P_{k+1} - P_k) \\ &= \textstyle\sum_{k=1}^{i}(\frac{q}{p})^k P_1. \end{aligned}$$

giving

$$\begin{aligned} P_{i+1} &= P_1 + P_1 \textstyle\sum_{k=1}^{i}(\frac{q}{p})^k = P_1 \sum_{k=0}^{i}(\frac{q}{p})^k \\ &= \begin{cases} P_1 \dfrac{1 - (\frac{q}{p})^{i+1}}{1 - (\frac{q}{p})}, & \text{if } p \neq q; \\ P_1(i+1), & \text{if } p = q = 0.5. \end{cases} \end{aligned} \tag{3.2}$$

(Here we used the geometric series equation $\sum_{n=0}^{i} a^i = \dfrac{1 - a^{i+1}}{1 - a}$, for any number $a$ and any integer $i \geq 1$.)

Choosing $i = N - 1$, and using the fact that $P_N = 1$, gives

$$1 = P_N = \begin{cases} P_1 \dfrac{1 - (\frac{q}{p})^N}{1 - (\frac{q}{p})}, & \text{if } p \neq q; \\ P_1 N, & \text{if } p = q = 0.5, \end{cases}$$

from which we get

$$P_1 = \begin{cases} \dfrac{1 - (\frac{q}{p})}{1 - (\frac{q}{p})^N}, & \text{if } p \neq q; \\ \dfrac{1}{N}, & \text{if } p = q = 0.5, \end{cases}$$

thus obtaining from Equ. (3.2)

$$P_i = \begin{cases} \dfrac{1 - (\frac{q}{p})^i}{1 - (\frac{q}{p})^N}, & \text{if } p \neq q; \\ \dfrac{i}{N}, & \text{if } p = q = 0.5. \end{cases} \tag{3.3}$$

**3.4.1 The Gambler's Ruin Problem.** Consider a gambler who starts with an initial fortune of \$1 and on each successive gamble either wins \$1 or loses \$1 with probabilities $p$ and $q = 1 - p$ respectively. The gambler's objective is to reach a total fortune of $N$, without first being ruined (running out of money). If the gambler succeeds, then he's won the game and stops playing. He also stops when he's ruined, whichever happens first. There's nothing special about starting with \$1; more generally the gambler begins with $i$ dollars where $0 < i < N$.

Let $P_i$ be the probability that the gambler wins (i.e. reaches $N$) when he has $i$ dollars. Thus $P_0 = 0$ (the gambler definitely does not win) and $P_N = 1$ (he wins).

We can rephrase this in terms of random walking. If the gambler starts with $i$ dollars, then he is initially at $x = i$ on the line. The fortune awaits at $x = N$ and ruin at $x = 0$. This means that the outcome of a gamble is the same difference equation as for the random walk in Equ. 3.1:

$$P_i = pP_{i+1} + qP_{i-1}$$

and has the same solution: Equ. 3.3.

**3.4.1.1 Checking the Equation.** John starts with \$2, and $p = 0.6$. What is the probability that he reaches $N = 10$? The following values $i = 2$, $N = 10$ and $q = 1 - p = 0.4$, so $q/p = 2/3$, are plugged into Equ. 3.3:

$$P_2 = \frac{1 - (\frac{2}{3})^2}{1 - (\frac{2}{3})^{10}} = 0.57$$

We check this result experimentally using Listing 3.5 (GamTrials.py).

```
def gamble(p, start, N):
  t = 0; pos = start
  while (pos > 0) and (pos < N):
    r = random.choices(['win', '
        lose'], [p, 1-p])
    t += 1
    if r[0] == 'win': pos += 1
    else: pos -= 1
  return t, (pos == N)

p = float( input("p? "))
i, N = map(int, input("i, N? ").
    split())
ts = []; numWins = 0
for _ in range(NUM_TRIALS):
  time, isWin = gamble(p, i, N)
  if isWin: numWins += 1
  ts.append(time)
print(f"Win Prob: {(numWins/
    NUM_TRIALS):.2f}")
avg = statistics.mean(ts)
std = statistics.stdev(ts)
print(f"Average time: {avg:.1f};
    stdev: {std:.1f}")
```

Listing 3.5. The Gambler's Ruin

The user specifies values for $p$, $i$, and $N$, and the code run NUM_TRIALS (1000) times, recording the fraction of wins and the time taken for the game to finish (either with a win or ruin). For example, we can supply the values for John's game:

```
> python GamTrials.py
p? 0.6
i, N? 2 10
Win Prob: 0.55
Average time: 18.0; stdev: 16.3
```

The estimated winning probability is close to the theoretical value (0.55 vs. 0.57).

A graph is also drawn (see Fig. 3.8), showing the time to win/lose for each of the trials, with the average time as a red-dashed line. This illustrates that there's a very wide spread in the duration for the game, sometimes taken around 100 bets to finish.

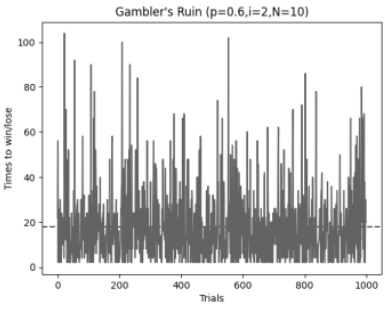



Figure 3.8. John's game; p = 0.6

If John's probability of winning is reduced even slightly below 0.5, the chances of him winning drop significantly. Equ. 3.3 gives $P_2 = 0.17$ when $p = 0.49$, and $i$ and $N$ are unchanged.

When Listing 3.5 is supplied with these inputs, it produces:

```
> python GamTrials.py
p? 0.49
```

```
i, N? 2 10
Win Prob: 0.16
Average time: 15.2; stdev: 18.6
```

The winning probability is again a close match to theory. Fig. 3.9 shows that although the average game duration is not much changed (from 18.0 to 15.2), the spread has become much wider, extending in some cases above 100 bets.

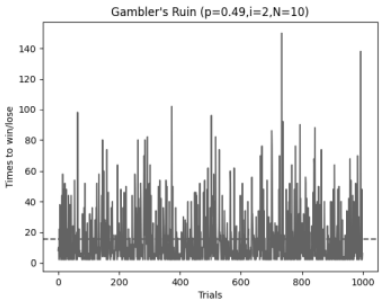



Figure 3.9. John's game; p = 0.49

**3.4.1.2 Bound to Lose.** Why is the gambler so unlikely to make money when the game is only slightly biased against him (i.e. when $p < q$)? We can identify two forces at work. First, the gambler's capital has random upward and downward swings from runs of good and bad luck. Secondly, and more importantly, the gambler's capital exhibits a steady, downward drift, because the negative bias means an average loss of a few cents on each $1 bet. The situation is illustrated in Fig. 3.10.

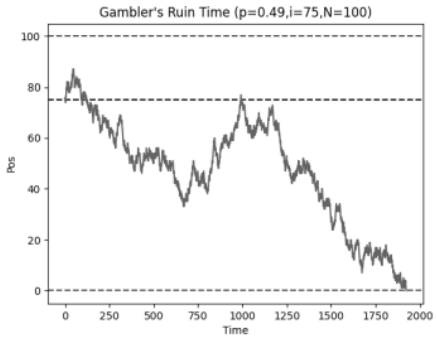



Figure 3.10. The downward drift of a $p < q$ random walk

If the gambler doesn't have a lucky, early, upward swing, then he's doomed since the required size of the upward swing becomes too large. It can be shown that as the size of the required swing grows, the odds that it occurs decrease exponentially.

Fig. 3.10 was generated by Listing 3.6 (GamTime.py).

```
def gambleTime(p, start, N):
  t = 0; ts = [t]
  pos = start; locs = [pos]
  while (pos > 0) and (pos < N):
    r = random.choices(['win', '
        lose'], [p, 1-p])
    t += 1
    ts.append(t)
    if r[0] == 'win':
      pos += 1
    else:
      pos -= 1
    locs.append(pos)
  return ts, locs
```

Listing 3.6. Timing the Gambler's Ruin

It only runs a single game for a specified $p$, $i$, and $N$, but plots the gambler's position changes during the course of that game. In the example above, $p = 0.49$, $i = 75$, and $N = 100$. Even though the player started very close to the winning position, the slightly biased probability was enough to lead him to ruin.

GamSurface.py attempts to visualize the drastic effects that even a slight bias against the gambler can cause, together with the player's starting position.

The code runs multiple games after the user supplies a winning position $N$, by varying the probability $p$ between 0.4 and 0.6, and $i$ in the range $[0..N]$. The winning fraction of games for NUM_TRIALS plays is collected, and plotted as a 3D surface (see Fig. 3.11).

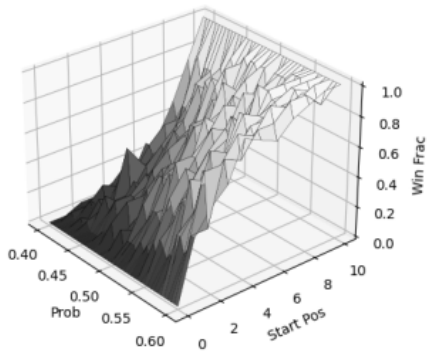
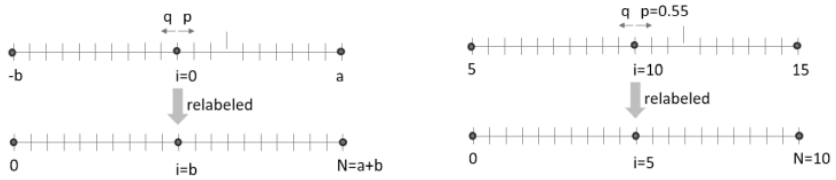



Figure 3.11. Probability and position

A small range of probabilities around 0.5 are enough to change a near-certain win to a near-certain loss, even when the gambler starts very close to the winning position. A look at the surface from different angles makes it clear that the relationship involves a hyperbolic curve, as suggested by our calculations.

**3.4.2 Relabeling the Axis.** Many random walk problems have the 'frog' starting at 0 and moving left to $-b$ or right to $a$. The question is usually phrased as 'what is the probability of reaching $a$ before $b$?'.

This problem, and similar ones, can be relabeled as in Fig. 3.12a to make them more clearly variants of Equ. 3.3.




(a) Random walking from 0 (b) Playing the stock market

Figure 3.12. Number lines

The probability of reaching $a$ before $b$ when starting from $i = 0$ is the same as the probability of reaching $a + b$ before 0 when starting from $i = b$

$$P_b = \begin{cases} \dfrac{1-(\frac{q}{p})^b}{1-(\frac{q}{p})^{a+b}}, & \text{if } p \neq q; \\ \dfrac{b}{a+b}, & \text{if } p = q = 0.5. \end{cases}$$

As an example, imagine that Jane bought a stock share for \$10, and the stock price moves randomly with $p = 0.55$. What's the probability that Jane's stock reaches \$15 before \$5?

We want the probability that the stock goes up by 5 before going down by 5, which can be relabeled as in Fig. 3.12b.

The probability $P_b$, where $b = 5$:

$$P_b = \frac{1-(\frac{q}{p})^b}{1-(\frac{q}{p})^{a+b}} = \frac{1-(0.82)^5}{1-(0.82)^{10}} = 0.73$$

**3.4.3 Gambling Variants.** The main variant of the Gambler's Ruin problem is when there are *two* players competing against each other. There is a probability $p$ that player A wins, and a probability of $1 - p$ that player B wins. Now suppose players A and B have have $n_0$ and $m_0$ dollars, respectively. At each trial, the loser pays the winner \$1. The game ends when one player is broke, so one player ends up with $n_0 + m_0$ dollars, and the other with nothing. This can be rephrased as a one player game (either A or B) who must reach a value $N = n_0 + m_0$ to win.

Another variation is when there is no winning value N, which permits the gambler to win any amount of money. Of course, he can still be ruined, which will stop the game. These changes are reflected in Equ. 3.3 by letting $N \to \infty$. If $p > 0.5$, then $q/p < 1$ and thus

$$\lim_{N\to\infty} P_i = 1 - (\frac{q}{p})^i > 0, \quad p > 0.5 \tag{3.4}$$

If $p \leq 0.5$, then $q/p \geq 1$ and then

$$\lim_{N\to\infty} P_i = 0, \quad p \leq 0.5 \tag{3.5}$$

Equ. (3.4) says that if $p > 0.5$, then there is a positive probability that the gambler will become infinitely rich. Equ. (3.5) says that if $p \leq 0.5$, then the gambler will certainly be ruined.

As an example, reconsider John from an earlier example (he starts with \$2, and $p = 0.6$). Under these new conditions, what's the probability that John will become disgustingly rich?

$$1 - (q/p)^i = 1 - (2/3)^2 = 5/9 = 0.56$$

Alternatively, if John started with \$1, what's the probability that he'll go broke? The probability he becomes infinitely rich is $1-(q/p)^i = 1-(q/p) = 1/3$, so the probability of ruin is 2/3.

Yet another variant is called *the Monkey on the Cliff*. A monkey is standing one step from the edge of a cliff and takes repeated independent steps: backward with probability $p$, or forward with probability $q$. Whereas the gambler moves on a line with two absorbing points (at $i = 0$ and $i = N$), the monkey only has a single end-point (the cliff edge, $i = 0$). This is actually the same as assuming that the gambler can become infinitely rich ($N \to \infty$).

**3.4.4 Expected Duration of the Game.** Let $d_i$ be the expected duration of the standard game from a point when the gambler has $i$ dollars. The boundary conditions are $d_0 = d_N = 0$, because $P_0 = 0$ and $P_N = 1$ means that the game is over.

The expected duration can be expressed as a difference equation for moving from position $i$ to the two neighbors

$$d_i = p(1 + d_{i+1}) + q(1 + d_{i-1}) = 1 + pd_{i+1} + qd_{i-1}$$

The "1+" represents a unit time step which occurs when moving. Rearranging, and including the boundary conditions, we obtain:

$$pd_{i+1} - d_i + qd_{i-1} = -1 \quad \text{where } d_0 = 0, d_N = 0.$$

Since the right-hand side, -1, is nonzero, this is an inhomogeneous linear difference equation, which can be a little tricky to solve. The solution for $0 < i < N$

is:

$$d_i = \begin{cases} \frac{1}{q-p}(i - N\frac{1-(\frac{q}{p})^i}{1-(\frac{q}{p})^N}), & \text{if } p \neq q; \\ i(N-i), & \text{if } p = q = 0.5. \end{cases}$$

We can return to our earlier examples to see if the experimental data (e.g. in Figs. 3.8 and 3.10) tallies with the values produced by the $d_i$ equation.

In the first case, $p = 0.6$, $i = 2$, and $N = 10$ results in $d_2 = 18$. On various runs, Listing 3.5 (GamTrials.py) reported an average time of 15.2, 15.3, and 19.6, which are fairly close, especially considering the wide spread in the game times.

Fig. 3.10 was generated with $p = 0.49$, $i = 75$, and $N = 100$ resulting in $d_{75} = 1970$, while the code reported 1917, so again fairly close agreement.

If we switch to the infinitely rich variant (i.e. $N \to \infty$), with $q > p$, the expected theoretical duration is

$$\lim_{N\to\infty} d_i = \lim_{N\to\infty} \frac{1}{q-p}(i - N\frac{1-(\frac{q}{p})^i}{1-(\frac{q}{p})^N}) = \frac{1}{q-p}(i-0) = \frac{i}{q-p},$$

since $(q/p)^N$ grows much quicker than $N$. This means that the gambler is certainly ruined, taking time $i/(q-p)$, on average.

In the case when $q = p$, we have

$$\lim_{N\to\infty} d_i = \lim_{N\to\infty} i(N-i) = \infty$$

So here, the gambler is still certainly ruined, but it may take forever.

## 3.5 Self Avoiding Random Walks

Fig. 3.13 shows a random path across a 11 x 11 square lattice taken from Donald Knuth's 'Mathematics and Computer Science: Coping with Finiteness' [**Knu96**].

A path begins at $(0, 0)$ (the lower left corner) and ends at $(10, 10)$ (the upper right corner), but is free to wander anywhere in between as long as no previously visited point is revisited.

The number of allowed alternative directions are written next to each point. For example, the 1's at the edges mean that there was only one way to go, since the other way is either already occupied or leads to a dead end. The probability that this particular path is obtained is the product of all the individual probabilities at each choice point, namely

$$\begin{aligned} \tfrac{1}{2}\cdot\tfrac{1}{2}\cdot\tfrac{1}{3}\cdot\tfrac{1}{3}\cdot\ldots\cdot\tfrac{1}{3}\cdot\tfrac{1}{1}\cdot\tfrac{1}{2} &= 2^{-34}2^{-24} \\ &= 1/48521024904413357015 04, \end{aligned}$$

or about one chance in $5 \times 10^{21}$.

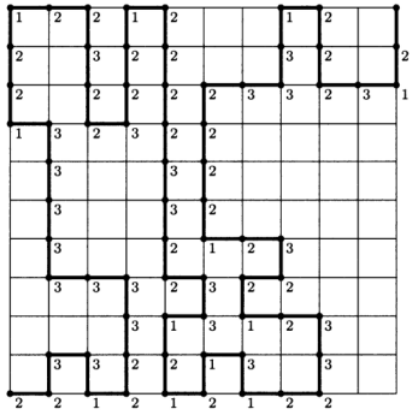


Figure 3.13. A numbered random path

How many paths are there from $(0,0)$ to $(10,10)$ which do not pass through the same point twice? With modern machines, this can be calculated, but real-size lattices quickly become too large to search in a feasible amount of time. However, physicists and chemists would very much like to know the number of self avoiding paths from $(0,0)$ to $(m,n)$ to gain valuable insights into the structure of long molecules.

We can estimate this number by simulation. We start at $(0,0)$ and pick a path by going from node to node, choosing a branch at random at each node, until we reach $(10,10)$. If the number of branches at the $i$-th node is $X_i$, then the probability of obtaining this path is $p = 1/X$, where $X = X_1X_2...X_n$. Thus, the expectation of the random variable X is

$$E(X) = \sum_{\text{All paths}} 1 = \#\text{ of paths through the tree.}$$

This means that counting the number of paths is reduced to finding the average value N, by averaging the product of the number of choices along many paths.

Listing 3.7 (gridPaths.py) utilizes depth-first search (DFS) and three main data structures: a 2D list representing the grid, a list of coordinates for the path chosen between $(0,0)$ and $(10,10)$, and a list of visited points. The path and visited lists are not the same since backtracking may undo part of the path but those dropped points will remain in the visited list.

```
def findPath(showGrid=False):
  global grid, path, visited
  grid = [[' '
    for y in range(GRID_SIZE)]
      for x in range(GRID_SIZE)]
path=[(0,0)]; visited=[(0,0)]
nChoices = dfs(0, 0, 1)
```

```
  if showGrid:
    printPath(path)
    print(f"{len(path)}, {
        nChoices*1.0:.3g}, {1/
        nChoices:3g}")
  return path, nChoices


def dfs(r, c, nChoices):
  if atEnd(r,c):
    return nChoices
  possSteps = nextSteps(r,c)
  nSteps = len(possSteps)
  for (r1,c1) in possSteps:
    path.append((r1,c1))
    visited.append((r1,c1))
    nchs = dfs(r1, c1, nChoices*
        nSteps)
    if nchs != -1:
      return nchs
    else:
      path.remove((r1,c1))
  return -1

def atEnd(r, c):
  return (r == GRID_SIZE-1) and (
      c == GRID_SIZE-1)
```

```
def nextSteps(r, c):
  random.shuffle(DIRS)
  steps = [dirPos(r, c, dir) for
      dir in DIRS]
  return [ (r1,c1) for (r1,c1) in
       steps if isValid(r1,c1)]

def dirPos(r, c, dir):
  if dir == 'N':
    return (r-1, c)
  elif dir == 'E':
    return (r, c+1)
  elif dir == 'S':
    return (r+1, c)
  else:   # assume it's W
    return (r, c-1)

def isValid(r, c):
  if r < 0 or r >= GRID_SIZE:
    return False
  if c < 0 or c >= GRID_SIZE:
    return False
  return not ((r,c) in visited)
```

Listing 3.7. Calculating a path

One way to use the code is by calling findPath() within the Python IDE:

```
>>> from gridPaths import *
>>> findPath(True)
------------------------
| . .       . . . . . . |
| . . . .   .     . . . |
| .   . .   . . . . . . |
| . . . .   . . . . . . |
|       .   .         . |
|       .   x   . . . . |
|   . . .   . . . . . . |
| . . .     . . . . .   |
| . . . . .     .       |
| . . .   . .     . . .   |
| . . .   . .         . . |
------------------------
79, 4.3e+26, 2.32684e-27
([(0, 0), (0, 1), (1, 1), (1, 0), (2, 0), (3, 0), (3, 1),
(3, 2), (2, 2), (1, 2), (1, 3), (2, 3), (3, 3), (4, 3),
(5, 3), (6, 3), (6, 2), (6, 1), (7, 1), (7, 0), (8, 0),
  : # many more tuples, not shown
(4, 10), (5, 10), (6, 10), (6, 9), (5, 9), (5, 8), (5, 7),
```

```
(6, 7), (6, 8), (7, 8), (7, 7), (8, 7), (9, 7), (9, 8),
(9, 9), (10, 9), (10, 10)], 429767699937105647757164544)
>>>
```

Since the grid is implemented as a 2D list, the code treats $(0,0)$ as the *top-left* corner, and $(10,10)$ as the *bottom-right*. Also, the path drawing is a lot simpler than Knuth's Fig. 3.13. The three numbers (79, 4.3e+26, 2.32684e-27) are the length of the path, the number of choices made, and the probability (1/(number of choices)). The 'x' in the grid picture indicates that the path went through the center point (5,5).

Each call to findPath() will usually generate widely different results, such as a path containing only 49 points, with the number of choices reduced from 4.3e26 to 1.2e14.

The other way to use the code is to call the program from the command line, which makes it generate 10000 paths, then print the mean and standard deviations for the collected path lengths and number of choices:

```
> python gridPaths.py
ctr: 46.1%
len. mean = 55.05;  stdev = 16.74
choices. mean = 1.96e+27;  stdev = 3.9e+28
choices. median = 2.37e+18;  mode = 2.74e+15
```

The statistics for the number of choices highlight the enormous amount of variation in the results. It indicates that the mean is biased by the presence of extreme observations, which is why we've also included the median and the mode. A good course of action in a situation like this is to plot the data to see how it skews.

Knuth's results are considerably different from ours. He reports a total number of possible paths of $(1.6 \pm 0.3) \times 10^{24}$, and an average path length of $92 \pm 5$. The chance of hitting the center is $81 \pm 10$ percent of all the paths, whereas our result is around 46 percent.

Perhaps our sample size wasn't large enough, but we think it's more likely that our DFS algorithm differs in some way from Knuth's. For example, are paths that are discarded during backtracking still counted in the number of choices?

The number of paths from corner to corner on an 11 x 11 grid was computed by John Van Rosendale in 1981; the value turns out to be approximately 1.569e24.

**Exercises**

(1) Add plotting code to Listing 3.7 to help you track down the discrepancy between our results and Knuth's.

(2) The number of shortest paths from $(0,0)$ to $(10,10)$ is $\binom{20}{10} = 184756$. Estimate this number by means of Knuth's approach.

(3) The number of shortest lattice paths from $(0,0)$ to $(10,10)$ which may lie below or touch the line $y = x$, but not cross it, is $\frac{2}{11}\binom{20}{10} = 33592$. Estimate this number by means of Knuth's approach.

(4) An 8 x 8 chessboard can be covered by 32 2x1 dominoes in $2^4 901^2 = 12988816$ different ways. Estimate this number by means of Knuth's approach.

## 3.6 Finding Geometric Probabilities by Simulation

Finding exact solutions for problems involving geometric probabilities and expectations is often quite complicated, frequently requiring multiple integrals. Instead we'll repeat a random experiment multiple times to get an estimate of the probability or expectation through *simulation*.

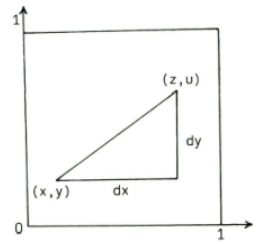



Figure 3.14. Distance between two points in a unit square

**3.6.1 The Distance between Two Points in a Square.** Two points are chosen at random in a unit square, and the expected distance between them is estimated. The coordinates $x, y, z, u$ in Fig. 3.14 are chosen by means of Python's random() so that the points $(x, y)$ and $(z, u)$ are uniformly distributed in the square.

In Listing 3.8 (expdist1.py), we don't assign the $x, y, z, u$ values to variables since we only need them to compute $dx$and $dy$. The sum variable accumulates all the $n$ distances, and the average, sum/n, is a good estimate $E_{est}$ of the expectation $E$.

```
n = int(input("n=? "))
sum = 0
for i in range(n):
  dx = random.random() - random.random()
  dy = random.random() - random.random()
  sum += math.sqrt(dx*dx + dy*dy)
print(f"expected dist == {sum/n:.6f}")
```

Listing 3.8. Points inside a square

When the program was run with $n = 10000$, typical output was $E_{est} = 0.523$.

The exact value calculated by L.A. Santalo [**San04**] is

$$E = \frac{\sqrt{2} + 2 + 5\ln(1+\sqrt{2})}{15} \approx 0.52141.$$

**3.6.2 The Distance between Two Points in an Equilateral Triangle.** This time the boundary values for the two points are a little more complicated. The idea is to choose two coordinates inside a unit square but count them only if they also lie inside the boxed equilateral triangle in Fig. 3.15.

If we translate the triangle so that vertex $V$ moves to the origin $O$ we get $y < -\sqrt{3}|x|$ for the shaded area. Now we translate back to $V = (\frac{1}{2}, \frac{1}{2}\sqrt{3})$ and get the equations for the two diagonal lines

$$y - \sqrt{3}/2 < -\sqrt{3}\,|x - \frac{1}{2}|.$$

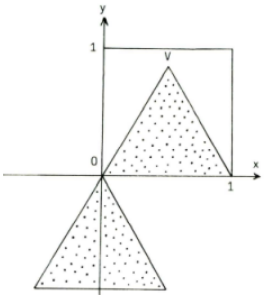



Figure 3.15. Distance inside an equilateral triangle

For $n = 10000$, Listing 3.9 (expdist2.py) gives the estimate $E_{est} = 0.362$.

```
n = int(input("n=? "))
sum = 0; d = math.sqrt(3)
for i in range(n):
  while True:
    x = random.random(); y = random.random()
    if y < d*(0.5 - abs(x - 0.5)):
      break
  while True:
    z = random.random(); u = random.random()
    if u < d*(0.5 - abs(z - 0.5)):
      break
  dist = math.sqrt((x-z)*(x-z) + (y-u)*(y-u))
  sum += dist
print(f"exp. dist. == {sum/n:.6f}")
```

Listing 3.9. Points inside a triangle

Santalo gives the exact value as

$$E = \frac{1}{5} + 3\ln\frac{3}{20} \approx 0.36479.$$

**3.6.3 The Distance between Two Points in a Unit Circle.**

This time it's simpler to use random.uniform(a, b), specifically random.uniform(-1,1). We generate points inside a 2x2 square centered at the origin, but only count those inside the unit circle (Fig. 3.16).

Listing 3.10 (expdist3.py) is based on this idea.

```
n = int(input("n=? "))
sum = 0
for i in range(n):
```

```
  while True:
    x = random.uniform(-1,1)
    y = random.uniform(-1,1)
    if x*x + y*y <= 1:
      break
  while True:
    z = random.uniform(-1,1)
    u = random.uniform(-1,1)
    if z*z + u*u <= 1:
      break
  dist = math.sqrt((x-z)*(x-z) + (y-u)*(y-u))
  sum += dist
print(f"exp. dist. == {sum/n:.6f}")
```

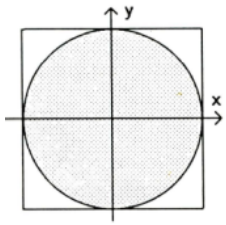



Figure 3.16. Distance inside a unit circle

Listing 3.10. Points inside a circle

For $n = 10000$, Listing 3.10 produces $E_{est} = 0.905$. Santalo gives the exact value as

$$E = \frac{128}{45\pi} \approx 0.90541.$$

**3.6.4 Buffon's Needle.** Buffon's needle is considered the first problem in geometric probability, dating from 1777.

In the plane, consider an infinite set of parallel lines spaced $a$ units apart. Onto the plane we throw "at random" a curve C of arbitrary shape (Fig. 3.17). (Buffon used a needle of length a.)

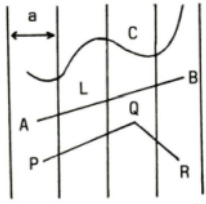



Figure 3.17. Curves and parallel lines

Let $S$ be the number of intersections with the parallels, and we want to find $E(S)$, the expected number of intersections. For a line segment $AB$ of length $L$ we set $E(S) = f(L)$ with an as yet unknown function $f$. Now throw the broken line $PQR$ onto the parallels. If $PQ$ has $S_1$ intersections and $QR$ has $S_2$ intersections, then $S = S_1 + S_2$. In other words, the expectation of the sum of two random variables is the sum of their expectations, even if those variables are not independent. Thus we have $E(S) = E(S_1 + S_2) = E(S_1) + E(S_2)$, or

$$f(L_1 + L_2) = f(L_1) + f(L_2). \tag{3.6}$$

Also, $f$ is additive, and increasing, so must be linear

$$f(L) = L * f(1). \tag{3.7}$$

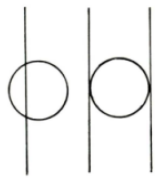
The additive property for expectations tells us that the expected number of intersection with any polygon can be expressed as Equ. (3.7), where $L$ is the length of the polygon. Since any "normal" curve can be approximated by a broken line to any accuracy, then Equ. (3.7) is actually valid for a curve of any shape and length

$L$. So take a circle of diameter $a$ and length $L = \pi a$, which has the useful property that it always has two intersections (Fig. 3.18a) with the parallels, i.e.

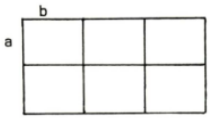
$$f(\pi a) = \pi a f(1) = 2 \quad \rightarrow \quad f(1) = \frac{2}{\pi a}$$

Combining this with Equ. (3.7) produces:

$$f(L) = \frac{2L}{\pi a} \tag{3.8}$$



(a) Circles intersecting lines




(b) Space divided into rectangles

Figure 3.18. Buffon's needles

Now suppose the plane is divided into congruent rectangles with sides $a, b$ (see Fig. 3.18b). Onto this plane we throw a curve of length $L$. If there are $S_1$ intersections with the verticals and $S_2$ intersections with the horizontals, with $S = S_1 + S_2$, then

$$E(S) = E(S_1) + E(S_2) = \frac{2L}{\pi a} + \frac{2L}{\pi b} = \frac{2L}{\pi}(\frac{1}{a} + \frac{1}{b}).$$

For $a = b = L = 1$, we get

$$E(S) = 4/\pi, \text{ or } \pi = \frac{4}{E(S)}$$

If we estimate $E(S)$ by means of

$$E_{\text{est}} = \frac{\text{intersections}}{\text{throws}} = \frac{S}{T}$$

then we get an estimate for $\pi$

$$\pi_{\text{est}} = \frac{4T}{S}.$$

**3.6.4.1 Implementing Buffon's Needle.** Fig. 3.19 illustrates a typical situation after dropping a needle of unit length onto a unit square. We imagine that the center of the needle is at $(x, y)$ relative to the bottom-left corner of the square, and the needle is rotated clockwise from the vertical by $a$ radians.

Throwing this needle "at random" can be coded using:

```
x = random.random()
y = random.random()
angle = math.pi*random.random()
```

The coordinates of the center of the needle are random numbers with a uniform distribution between 0 and 1, and its angle with the northern direction, counted to the right, is randomly between 0 and $\pi$.

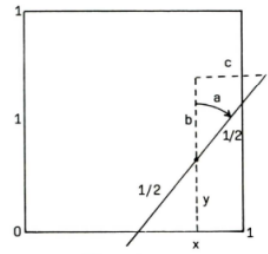



Figure 3.19. A needle intersecting a square

The throwing area can be considered to be an infinite grid of unit squares, but its periodic pattern ensures that the result will be the same if we pick a center with a uniform distribution from a single square.

The generated plot (Fig. 3.20) shows that buffon.py converges (slowly) towards a single probability. It produces a poor estimate for $\pi$ of 3.14398 after one million throws, correct only to 2 decimal places.

The needle's horizontal and vertical intersections are observed by checking if its end points are in the same square or not. This is done by truncating the x- and y- coordinates to integers. One slight complication is that if a needle is off the left side (or bottom) of the square, then its x- (or y-) coordinate should be truncated to -1. This behavior is implemented by truncL().

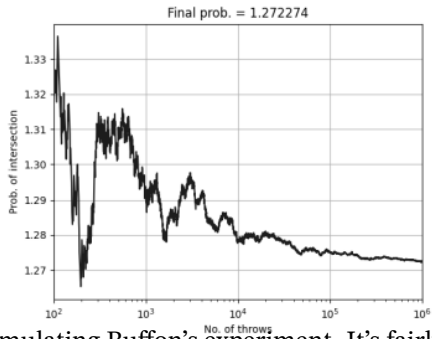



Figure 3.20. Plot of needle intersection probs

buffon.py is really only useful for simulating Buffon's experiment. It's fairly worthless for computing $\pi$ since it has to use $\pi$ in its calculations. Nevertheless, this problem has some interesting applications in stereology, as explained by Solomon [**Sol78**].

A better needle experiment is described in Ex. 7 below.

**Exercises**

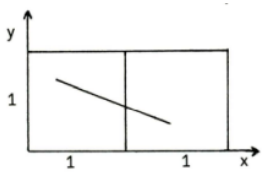
(1) Listing 3.11 (expdist3b.py) shows another way of selecting pairs of points in the unit circle for calculating their average distance apart. Is this method faster than the one in Listing 3.10?

```
n = int(input("n=? "))
sum = 0
for i in range(n):
```

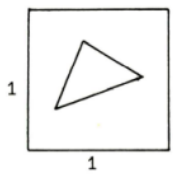
```
  while True:
    x = random.random();
    y = random.random()
    if x*x + y*y <= 1:
      break
  while True:
    z = random.random();
    u = random.random()
    if z*z + u*u <= 1:
      break
  if random.randint(0,1) == 0:
    z = -z
  if random.randint(0,1) == 0:
    u = -u
  dist = math.sqrt(
    (x-z)*(x-z) + (y-u)*(y-u))
  sum += dist
print(f"exp. dist. v2 == {sum/n:.6f}")
```

Listing 3.11. Points inside a circle (2nd approach)

(2) A point is chosen at random in each of the two squares in Fig. 3.21a. Find by simulation the mean distance between the two. (It is possible to calculate $E \approx 1.08814$.)




(a) Distance between two points




(b) Three points forming an obtuse triangle

Figure 3.21. Selecting points

(3) Three points are chosen at random inside the unit square in Fig. 3.21b. Let P be the probability that they form an obtuse-angled triangle, and obtain a value for P by simulation. It can be shown, though with great labor, that $P = 97/150 + \pi/40 \approx 0.725$.

(4) Four points, $P_1, P_2, P_3, P_4$, are chosen at random inside a unit cube. Find by simulation the expected volume of the tetrahedron $P_1P_2P_3P_4$. In this case we don't know the exact value of the expected volume.

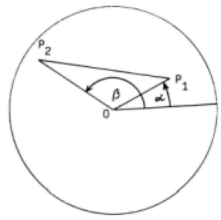
(5) Abby suggests to Carl: Let's choose two points P and Q at random inside a unit circle. If $|PQ| \leq 1$ then I win, otherwise you win. What's Abby's chance of winning? Estimate the answer using either Listing 3.10 or Listing 3.11, but theory predicts that Abby will win with probability $1 - (3\sqrt{3}/4)/\pi \approx 0.5865$.

(6) Pick two random angles, $\alpha$ and $\beta$, then measure out random distances in their directions from the origin to $P_1$ and $P_2$ (see Fig. 3.22a). Find the expected distance between $P_1$ and $P_2$, which will *not* agree with the distance calculated in Ex. 3. Why not?

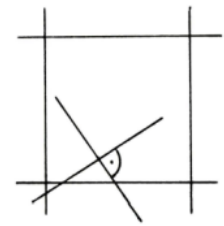
If, instead of averaging the distance

$$|P_1P_2| = \sqrt{r_1^2 + r_2^2 - 2r_1r_2\cos(\beta - \alpha)},$$

we find the average of the quantity $\sqrt{r_1 + r_2 - 2\sqrt{r_1r_2}\cos(\beta - \alpha)}$, we do get the same answer as in Ex. 3. Why?




(a) Distance between two points inside a circle



(b) A cross intersecting a rectangle

Figure 3.22. Buffon exercises

(7) Two needles of unit length are tied together at their centers to form a cross, then thrown onto a chessboard made up of unit squares (see Fig. 3.22b). This will give us a better estimate of $\pi$ than using one needle, which you should check by writing a program similar to `buffon.py`.

(8) Find the mean distance between two points chosen at random in the unit cube.

Theory predicts (see 'E2629', David P. Robbins and Theodore S. Bolis, *American Mathematical Monthly* 1978, p. 277-278, `https://www.jstor.org/stable/i315089`):

$$E = \frac{4 + 17\sqrt{2} - 6\sqrt{3} + 21\ln(1 + \sqrt{2}) + 42\ln(2 + \sqrt{3}) - 7\pi}{105} \approx 0.661707.$$

(9) **Friends Meeting**. Suppose two friends decide to meet between 8 pm and 9 pm. When one of them arrives at the meeting spot, he waits for 10 minutes for the other to show up. For instance, if Friend 1 arrives at 8:40, he'll wait until 8:50 for Friend 2. Friend 1 doesn't know if Friend 2 had shown up earlier and left (and the same is true for Friend 2). However, both friends know that if they arrive at a time near to 9pm, say 8:55, that they only need to wait until 9:00, not 9:05.

What's the probability that the two friends meet?

## 3.7 The Bertrand Paradox

Joseph Bertrand introduced this problem in *Calcul des probabilités* in 1889: draw an equilateral triangle inside a circle, then add a chord to the circle at random. What is the probability that the chord is longer than a side of the triangle?

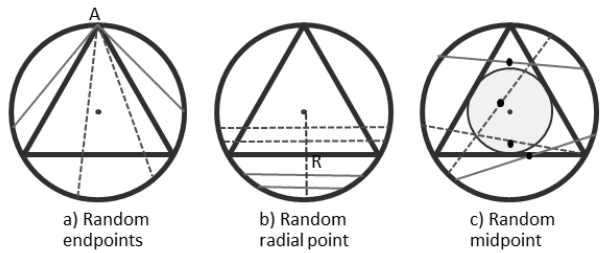



Figure 3.23. Three ways to select chords

The problem statement is vague enough that Bertrand produces three different, valid, answers:

(1) *The random endpoints method*. Choose a random start point for the chord, called $A$, anywhere on the circumference. The other end can be placed anywhere else on the circumference, samples of which are shown in Fig. 3.23a as dashed and solid lines.

The equilateral triangle can be oriented in any direction, so position it to ensure that one vertex coincides with $A$. Now it's clear that only those chords that cut across the triangle are longer than its side (the dashed lines).

The angle of the equilateral triangle at $A$ is 60°, and all the chords lie within a 180° range, so the chances of drawing a chord larger than the side of the triangle must be $\frac{1}{3}$.

(2) *The random radial point method*. Choose a point at random on the radius line $R$ of the circle, and construct a chord through this point, perpendicular to the radius. Samples of possible chords are shown in Fig. 3.23b.

The equilateral triangle can be oriented in any direction, so position it to ensure that one side is parallel to the chords. Now it's clear that a chord will only be longer than this side if it's nearer the center of the circle (the dashed lines) than the side.

The triangle's side bisects the radius line, therefore the probability that a random chord on the line is longer than that side is $\frac{1}{2}$.

(3) *The random midpoint method.* Choose a point anywhere within the circle and construct a chord with that point as its midpoint. Possible chords are shown in Fig. 3.23c as dashed and solid lines.

A chord will only be longer than a triangle side if its midpoint lies within the circle inscribed inside the triangle. This pale-gray circle has a radius 1/2 that of the larger circle, so its area is one fourth the area of the larger one ($\frac{1}{2}^2$). Therefore the probability a chord is longer than a side of the triangle is equal to the chance its midpoint is inside the small circle's area, $\frac{1}{4}$.

Two aspects of the preceding arguments may be a little unclear: why the radius line R is bisected by the triangle side in Fig. 3.23b, and why the inscribed circle in Fig. 3.23c is half the radius of the main circle. Fig. 3.24 shows why these two statements are true.

A standard theorem for circles is that when two angles are subtended by the same chord, then the angle at the center is twice the angle at the circumference. This is the case for chord GH in Fig. 3.24, which subtends angles at C and F. So angle GCH is twice that of GFH, and since GFH is 60°, then GCH is 120°. The triangle GCH is isosceles (the lines GC and HC are the same length), so the angles CGH and CHG must be the same, which implies they are each 30°. Switching to trigonometry, we can calculate the length $CD = CH\sin(30) = \frac{R}{2}$. CE's length is R, which means that GH bisects CE at D.

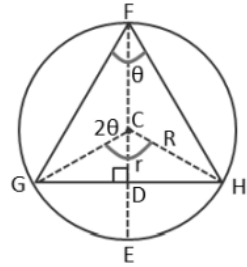



Figure 3.24. An equilateral triangle inside a circle

The inscribed circle in Fig. 3.23c has the equilateral triangle's sides as tangents, which means that its radius is equal to CD in Fig. 3.24, which we have just determined to be $\frac{R}{2}$.

We began by mentioning that Bertrand's problem statement was 'vague enough' to allow three different solutions. The vagueness resides in the meaning of choosing a chord "at random". The three solutions correspond to different selection methods.

Martin Gardner's article on the problem (see Chapter 19 in [**Gar61**]) suggests informative analogies for these methods:

(1) Two spinners are mounted at the center of a circle. They rotate independently. We spin them, mark the two points at which they stop, and connect the points with a straight line.

(2) A large circle is chalked on the sidewalk. We roll a broom handle toward it from some distance away, until the handle stops somewhere on the circle.

(3) We paint a circle with molasses and wait until a fly lands on it; then we draw a chord whose midpoint is the fly.

The difference in the selection methods can be visualized by drawing the generated chords and their midpoints. Listing 3.12 (bertrand.py) produces the six circular diagrams in Fig. 3.25, two for each method. The circles along the top row show the distribution of the chords, while the bottom row highlights the midpoints of those chords.

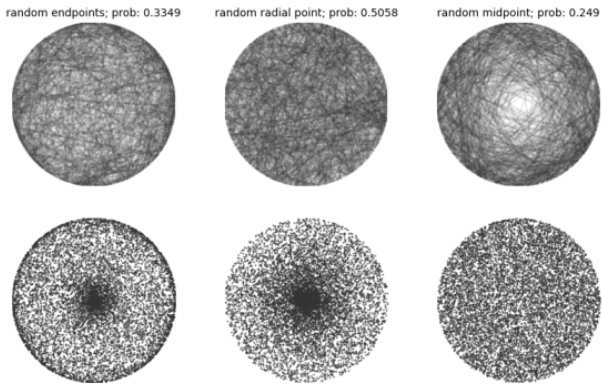



Figure 3.25. Plots of the three Bertrand methods

The diagrams are created using matplotlib's support for drawing circles and lines. We won't explain that code since it's not really relevant to the math problem, but we will show the implementation of the three chord generation methods. This code is based on a numpy solution by Christian Hill at `https://scipython.com/blog/bertrands-paradox/`.

```
def bertrand1():
  chords = []
  midPts = []
  for i in range(NUM_CHORDS):
    angle1 = random.random()*2*math.pi
```

```
    angle2 = random.random()*2*math.pi
    chord = ( R*math.cos(angle1), R*math.sin(angle1),
              R*math.cos(angle2), R*math.sin(angle2) )
    chords.append(chord)
    midPt = ((chord[0]+chord[2])/2, (chord[1]+chord[3])/2)
    midPts.append(midPt)
  return "random endpoints", chords, midPts

def bertrand2():
  angles = [ random.random()*2*math.pi
                              for i in range(NUM_CHORDS)]
  radii =  [ random.random()*R for i in range(NUM_CHORDS)]
  midPts = []
  for i in range(NUM_CHORDS):
    midPt = (radii[i] * math.cos(angles[i]),
             radii[i] * math.sin(angles[i]))
    midPts.append(midPt)
  chords = getChords(midPts)
  return "random radial point", chords, midPts

def getChords(midPts):
  # Return the chords with the provided midPts
  chords = [(0,0,0,0) for i in range(NUM_CHORDS)]
  for i, (x0, y0) in enumerate(midPts):
    m = -x0/y0
    c = y0 + x0**2/y0
    A, B, C = m**2+1, 2*m*c, c**2-R**2
    d = math.sqrt(B**2 - 4*A*C)
    xEnd1 = (-B + d)/ 2 / A
    xEnd2 = (-B - d)/ 2 / A
    yEnd1 = m*xEnd1 + c
    yEnd2 = m*xEnd2 + c
    chords[i] = (xEnd1, yEnd1, xEnd2, yEnd2)
  return chords

def bertrand3():
  angles = [ random.random()*2*math.pi
                             for i in range(NUM_CHORDS)]
  radii =  [ math.sqrt(random.random())*R
                             for i in range(NUM_CHORDS)]
  midPts = []
  for i in range(NUM_CHORDS):
    midPt = (radii[i] * math.cos(angles[i]),
             radii[i] * math.sin(angles[i]))
    midPts.append(midPt)
  chords = getChords(midPts)
  return "random midpoint", chords, midPts
```

Listing 3.12. The Bertrand methods

getChords() is a little complicated since it solves a quadratic equation involving the intersection of a line (representing a chord) and a circle (see Fig. 3.26).

The equation for the line, $y = mx + c$, is squared to give $y^2 = m^2x^2 + c^2 + 2mxc$, then used to replace $y^2$ in the circle's equation $x^2 + y^2 = R^2$. After a little rearranging, the resulting quadratic in $x$ is $(m^2 + 1)x^2 + 2mcx + (c^2 - R^2) = 0$ which is solved in the usual way.

The drawing code keeps a tally of the number of chords whose length exceeds the triangle's side, which is used to calculate a probability for each method. These are included in the titles in Fig. 3.25, and approximate the theoretical results.

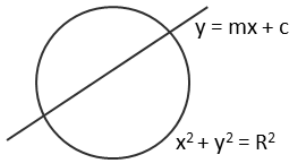



Figure 3.26. Equations of a line and a circle

## 3.8 The Broken Stick Problem

Suppose we randomly break a stick in two places. What's the probability that we can form a triangle using the pieces?

We'll see that there are two different valid answers, depending on the exact meaning of 'randomly break'.

**3.8.1 Independent Breaks.** The first scenario relies on the two breaks being independent. You could imagine this as the breaks being made by two blind-folded people at the same time.

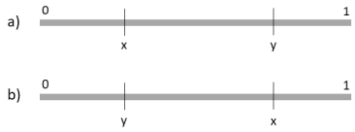



Figure 3.27. Two positions for x and y

We can assume, without loss of generality, that the stick is of unit length. Let the first cut be at position $x$ and the second at $y$, which allows for two interpretations shown in Fig. 3.27.

The *triangle inequality* condition for making a triangle from three lengths $a$, $b$, and $c$ applies to each of the three sides:

$$\begin{aligned} a + b &> c \\ b + c &> a \\ c + a &> b \end{aligned}$$

In Fig. 3.27a), the three lengths are $x$, $y - x$, and $1 - y$. Plugging these into the triangle inequalities produces:

$$\begin{aligned} x + y - x &> 1 - y \\ y - x + 1 - y &> x \\ 1 - y + x &> y - x \end{aligned}$$

These can be simplified to $y > \frac{1}{2}$, $x < \frac{1}{2}$, and $y - x < \frac{1}{2}$, which can be visualized as lines defining regions inside a unit square (see Fig. 3.28a). The unit square comes from the fact that $x$ and $y$ can have any value in the range $[0, 1]$. The region A where all three inequalities are true equals the probability $\frac{1}{8}$.

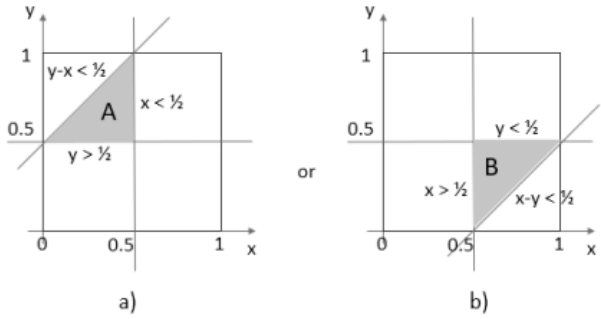



Figure 3.28. Probabilities as areas

We now repeat this process for Fig. 3.27b. The three lengths are $y$, $x - y$, and $1 - x$. Substituted into the triangle inequalities, we get:

$$\begin{aligned} y + x - y &> 1 - x \\ x - y + 1 = x &> y \\ 1 - x + y &> x - y \end{aligned}$$

These can be simplified to $x > \frac{1}{2}$, $y < \frac{1}{2}$, and $x - y < \frac{1}{2}$ which defines a region B in Fig. 3.28b which equals the probability $\frac{1}{8}$.

The total probability of obtaining a triangle is a sum of the two regions, $\frac{1}{4}$.

Listing 3.13 (sticks.py) simulates this process of triangularity testing, with the crucial part being the generation of the $x$ and $y$ lengths, which are independent.

```
def isTriangle(a, b, c):
  return (a + b > c) and (a + c > b) and (b + c > a)

numTris = 0
for i in range(NUM_TRIALS):
  x = random.uniform(0, 1)
  y = random.uniform(0, 1)    # 1st scenario
  # y = random.uniform(x, 1)  # 2nd scenario
  if x > y:
    x, y = y, x
  if isTriangle(x, y-x, 1-y):
    numTris += 1
```

```
print(f"Estimated prob = {(numTris/NUM_TRIALS):0.6f}")
```

Listing 3.13. Making a triangle (scenario 1)

The output confirms the theoretical probability:

```
> python sticks.py
Estimated prob = 0.249740
```

**3.8.2 Dependent Breaks.** Breaking the stick in a dependent manner can be achieved by carrying out the cuts in order: first you break off and take away the left piece, then you break the remaining piece into two. If we call the two break positions $x$ and $y$ again, then we end up with three pieces like those in Fig. 3.27a, but $y$'s position depends on $x$'s.

The lengths are $x$, $y - x$, and $1 - y$, and applying the triangle inequality, we get $x < \frac{1}{2}$, $y > \frac{1}{2}$, and $y - x < \frac{1}{2}$ (the same inequalities as for region A in Fig. 3.28a). The last equation is also rearranged to $y < x + \frac{1}{2}$. The first equation states that $x$ must range between 0 and $\frac{1}{2}$ and the second and third equations imply that $y$ must range between $\frac{1}{2}$ and $x + \frac{1}{2}$.

Therefore, the three pieces can form a triangle if $x$ is in the interval $(0, \frac{1}{2})$ and $y$ is in $(\frac{1}{2}, x + \frac{1}{2})$. Furthermore, $y$ is guaranteed to be in the interval $(x, 1)$, and y's required interval $(\frac{1}{2}, x + \frac{1}{2})$ must be smaller.

Thus, for a given $x$, the *probability* of $y$ being in the required range is the ratio of the length of its required range $(x + \frac{1}{2} - \frac{1}{2} = x)$ to the length of its guaranteed range $(1 - x)$. The resulting probability equation is $\frac{x}{1-x}$.

Since $x$ can vary between 0 and $\frac{1}{2}$. The total probability is the sum of all the possible probabilities, which we can obtain by integrating the probability equation over $x$'s limits.

$$P(\text{triangle}) = \int_0^{\frac{1}{2}} \frac{x}{1-x} dx$$

The integral is

$$[\, 1 - x - \ln|1 - x| \,]_0^{\frac{1}{2}} = \ln 2 - \frac{1}{2} \approx 0.1931$$

Checking out this result with Listing 3.13 requires the replacement of the line of code that generates $y$ by:

```
y = random.uniform(x, 1)
```

This makes it quite explicit that $y$'s value depends on $x$'s. When the modified program is run, it returns a result similar to our calculated probability:

```
> python sticks.py
Estimated prob = 0.191490
```

## 3.9 The Monty Hall Problem

Monty Hall hosts a game show in which a contestant wins whatever is behind a door that is chosen from three closed doors. A car is behind one and a goat is behind each of the other two.

Once the contestant has chosen a door, Monty doesn't open it but instead opens a different door to reveal a goat. Monty then gives the contestant the choice of staying with their original choice or switching to the remaining unopened door. Should the contestant stay or switch to maximize the chance of winning the car?

Most people would think that the chance of winning after the goat is revealed is 50%, and so switching won't make a difference. They're wrong.

There are some rules that determine which door Monty can open after the contestant has made a choice.

- Monty knows which door hides the car.
- Monty doesn't open the door chosen by the contestant.
- Monty always offers the contestant a chance to switch.
- If more than one 'goat' doors could be opened, Monty chooses one at random.

Before we start coding, let's see how the problem can be represented with probabilities.

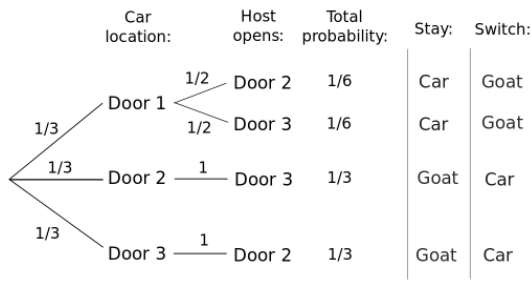



Figure 3.29. The probability of every outcome

The probabilities are summarized by the decision tree in Fig. 3.29. It highlights two places where our intuition may fail us – first there are *four* scenarios to consider, and they do *not* have equal probabilities of occurring.

We now consider the outcome if the contestant switches. The result is positive (win the car) in only the last two cases, but their total probability is 2/3, so it is worthwhile to change.

Listing 3.14 (montyhall.py) simulates a large number of games and keeps track of the probabilities of winning with, and without, switching doors.

```
trials = range(NUM_ITERS)
round1 = range(NUM_DOORS)  # door choices in round 1
stayCount, chgCount = 0, 0
stayProbs = []; chgProbs = []
for i in trials:
  carDoor = random.choice(round1)
  playerPick1 = random.choice(round1)
  montyChoices = [n for n in round1
                  if n!= carDoor and n!= playerPick1]
  goatDoor = random.choice(montyChoices) # what Monty opens
  round2 = [n for n in round1
                  if n!= playerPick1 and n!=goatDoor]
  playerPick2 = random.choice(round2)
  if playerPick1 == carDoor:
    stayCount += 1    # would win by staying
  if playerPick2 == carDoor:
    chgCount += 1     # would win by switching
  p1 = stayCount/(i+1)
  p2 = chgCount/(i+1)
  stayProbs.append(p1)
  chgProbs.append(p2)
```

Listing 3.14. Simulating the Monty Hall problem

Fig. 3.30 shows that the switch and stay probabilities gradually approach 2/3 and 1/3.

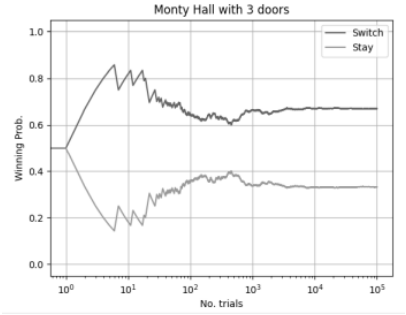



Figure 3.30. Monty Hall simulated

**3.9.0.1 History.** The earliest puzzle related to the Monty Hall problem is Bertrand's box paradox, posed by Joseph Bertrand in 1889 in his *Calcul des probabilités*. There are three boxes: one containing two gold coins, a box with two silver coins, and a box with one of each. After choosing a box at random and withdrawing one coin at random, which happens to be gold, what's the probability that the other coin is gold?

A somewhat better known antecedent is the Three Prisoners problem, first published in Martin Gardner's "Mathematical Games" column in *Scientific American* in 1959 [**Gar61**].

**3.9.0.2 A Generalization.** We finish with a variant of the puzzle suggested in Blom's excellent *Problems and Snapshots from the World of Probability* [**BHS93**]: person A is invited by person B to play a game with an urn containing $w$ white balls and $b$ black balls ($w + b \geq 3$). Person A is blindfolded before beginning, then asked to choose between two strategies:

(1) Player A draws a ball at random. If it's white, then he wins the game, otherwise he loses.

(2) Player A draws a ball and throws it away. Player B removes a black ball. A again draws a ball. If it's white, he wins the game, otherwise he loses.

Which strategy is best for Player A? Clearly, if A chooses Strategy 1, he wins with probability

$$P_1 = \frac{w}{w+b}.$$

If A chooses Strategy 2, then there are two paths to consider – A throws away a white ball or a black ball. In each of these, B always removes a black ball before A tries again. This means that there are now two less balls (i.e. $w + b - 2$), but in case 1 there are $w - 1$ white balls, while in case 2 there are $w$. This can be expressed as:

$$\begin{aligned} P_2 &= \frac{w}{w+b} \cdot \frac{w-1}{w+b-2} + \frac{b}{w+b} \cdot \frac{w}{w+b-2} \\ &= \frac{w}{w+b}\left(1 + \frac{1}{w+b-2}\right) \end{aligned}$$

Since $P_2 > P_1$, Strategy 2 is the best one for A.

In Monty Hall, we have $w = 1$, $b = 2$, and so $P_1 = 1/3$ and $P_2 = 2/3$.

## 3.10 Random Choice of an s-Subset from an n-Set

An $s$-sized sample is chosen at random from an $n$-sized population if each sample of that size has the same probability $1/\binom{n}{s}$ of being chosen. This is an interesting statistical problem, and we'll look at four elementary algorithms that address it.

**3.10.1 Algorithm 1.** Listing 3.15 (`sample1.py`) implements one of the more elegant solutions. For each $i$ from 1 to $n$ it calls randint(0, $n - 1$). If the result is $< s$, then it prints $i$ and decreases $s$ by 1. For each call it also decreases $n$ by 1.

```
s,n = map(int, input("s n=? ").split())
i = 1
while n > 0:
  if random.randint(0, n-1) < s:
    s -= 1
    print(i, end=' ')
  i += 1; n -= 1
```

Listing 3.15. Sampling from a set

Examples:

```
> python sample1.py
s n=? 10 100
5 17 36 49 53 72 77 84 95 100
> python sample1.py
s n=? 5 300
3 43 90 159 206
```

The algorithm has time complexity $O(n)$ since randint() is called $n$ times, and has the advantage that the sample is printed in increasing order.

If $n$ is the number of elements still to be considered and $s$ the number of elements still to be chosen, then the next number will be printed with probability $s/n$. This isn't easy to understand, and it helps to trace the first few iterations of the code's while-loop.

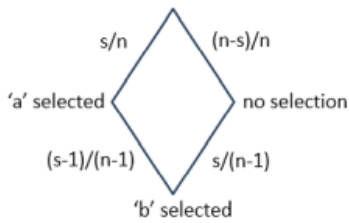



Figure 3.31. Probabilities for reaching the second selection

The first selection is obviously printed with the correct probability $p_1 = s/n$. It requires more thought to see that the next selected value is printed with the same likelihood. Consider the two paths through Fig. 3.31. The left path is the probability for getting $\{a, b\}$ while the right-hand path is for $\{b\}$ (i.e. no selection was made during the first loop). We combine the paths in Fig. 3.31 to get:

$$p_2 = \frac{s}{n} * \frac{s-1}{n-1} + \frac{n-s}{n} * \frac{s}{n-1} = \frac{s}{n(n-1)} * (s-1+n-s) = \frac{s}{n}.$$

**3.10.2 Algorithms 2 and 3.** Listing 3.16 (sample2.py) is more efficient than algorithm 1 if $s$ is a lot smaller than $n$, and it's also perhaps easier to understand.

```
s,n = map(int, input("s n=? ").split())
x = [0]*n
for i in range(s):
  while True:
    r = random.randint(0,n-1)
    if x[r] == 0: break
  print(r, end=' '); x[r] = 1
```

Listing 3.16. Sampling from a set (2/3)

```
> python sample2.py
s n=? 10 100
13 92 67 12 64 1 44 86 97 6
```

It chooses a random number $r$ from 0 to $n-1$, and checks if it's already been chosen. If it has then $x[r] = 1$, which causes the code to repeat the selection. Otherwise $r$ is printed, and $x[r]$ is set to 1 to record that $r$ has been chosen.

One drawback is that the numbers are printed in their selected, random order.

sample3.py implements a one-line change to Listing 3.16 to print the $s$ elements in sorted order:

```
print(*[i for i in range(n) if x[i] == 1])
```

The result:

```
> python sample3.py
s n=? 10 100
4 6 7 11 13 25 33 66 75 84
```

**3.10.3 Algorithm 4.** Listing 3.17 (sample4.py) employs a technique corresponding to drawing an $s$-sized sample from $n$ numbers *without* replacement.

```
s,n = map(int, input("s n=? ").
    split())
x = [i for i in range(n)]
for i in range(s):
  r = i + random.randint(0, n-1-i)
  x[i], x[r] = x[r], x[i]
  print(x[i], end=' ')
```

```
> python sample4.py
s n=? 10 100
78 63 73 94 47 55 61 90 81 13
```

Listing 3.17. Sampling from a set (4)

It chooses a random number $r$ from 0 to $n-1$ and exchanges $x[r]$ with $x[0]$. Then it selects a random number $r$ from **1** to $n-1$ and exchanges $x[r]$ with $x[1]$,..., and continues in this way until it picks a random number $r$ from **s-1** to $n-1$ and exchanges $x[r]$ with $x[s-1]$. The resulting subsequence in $x[0]$ to $x[s-1]$ is out of order.

The standard reference for these algorithms in Vol. 2 of *The Art of Computer Programming*, [**Knu97**], specifically Section 3.4.2.

**3.10.4 Python's sample().** Python's random module contains a sample() function which can return a $k$ length list of elements chosen from a supplied sequence, implementing sampling without replacement in a similar way to Listing 3.17.

To choose a sample from a range of integers, it's especially fast and space efficient to use range(), as in:

```
>>> import random
>>> random.sample(range(10000000), 10)
```

```
[1554429, 5899448, 1217734, 7432044, 1568457, 8183569,
 4173744, 7896302, 6848341, 1777236]
```

**Exercises**

(1) **Generating a random permutation**. $n$ numbers are stored in the list $x$. Write a program which can permute the numbers while ensuring that all the possible permutations have the same probability of occurring. There is a suitable function for doing this in the random module, but you'll benefit from coding a version yourself.

(2) Generate a random permutation of $x$, and record the length $k$ of the longest "saw-tooth" subsequence $x[p+1] < x[p+2] > x[p+3] < ... > x[p+k]$ (with teeth going upwards and the sequence ending with a last downward value).

## 3.11 Coin Tosses in Software

In this and the next two sections we'll examine some unusual methods for generating random digits.

**3.11.1 Generating Random Binary Digits.** We compute $z = \log_b a$ with $1 < a < b$. By definition this is the solution $x$ for

$$b^x = a, \quad 0 < x < 1.$$

Let

$$x = 0.d_1d_2d_3... = d_1/2 + d_2/4 + d_3/8 + ...$$

be the binary representation of $x$. Then

$$b^{d_1/2+d_2/4+d_3/8+...} = a. \tag{3.9}$$

Square both sides and update $a$ to $a * a$:

$$b^{d_1+d_2/2+d_3/4+...} = a.$$

If $a < b$ then $d_1 = 0$. Otherwise $d_1 = 1$ so update $a$ to $a/b$. In both cases we get

$$b^{d_2/2+d_3/4+...} = a. \tag{3.10}$$

This is again Equ. (3.9) with the first digit shaved off. Thus we have a simple algorithm for printing successive binary digits of $x$ which becomes Listing 3.18 (genBin.py).

```
def genBinary(a, b, i):
  while i >= 1:
    a = a*a
    if a < b:
      print("0", end='')
    else:
      a = a/b
      print("1", end='')
    i = i-1
  print()
```

Typical output:

```
> python genBin.py
a b? 2 10
01001101000100000100110101 0
0001001111101111001111111111
    : # many more lines
00111010100010000000111010 11
01111110001110010100011
```

Listing 3.18. Generating random binary digits

It's possible to check this output by remembering that $b^x = a$. So we need to calculate $x$ in $10^x = 2$, or $x = \log_{10} 2$, and convert it to binary. We've included a decimalToBinary() function in decBin.py which translates decimal fractions to binary; it's used like so:

```
>>> import math
>>> from decBin import *
>>> decimalToBinary( math.log10(2), 50)
'.01001101000100000100110101000010011111011110011111'
```

This confirms (at least for the first 50 digits) that Listing 3.18 is producing the correct output.

Instead of 2 and 10 we could use other numbers as the inputs $a$ and $b$ for Listing 3.18, provided they generate a long fractional binary form (so 1 and 2 aren't suitable; try them and see).

genBin2.py is a variant of Listing 3.18. After every 1000 digits of output it prints the total number of "ones" and "zeros" produced so far. This is a simple way to check the quality of the randomness, since the two sums should be about equal.

With $a = 2$ and $b = 10$ and $n = 10000$, we get the following:

```
> python genBin2.py
a b n? 2 10 10000
1s: 504  0s: 496
1s: 997  0s: 1003
1s: 1498  0s: 1502
1s: 1996  0s: 2004
1s: 2517  0s: 2483
1s: 3016  0s: 2984
1s: 3537  0s: 3463
1s: 4036  0s: 3964
1s: 4554  0s: 4446
1s: 5057  0s: 4943
```

**3.11.2 Generating Random Decimal Digits.** In this subsection we'll produce random decimals by calculating the reciprocal of a prime (e.g. 1/17, 1/2063), and extract the repeated decimal sequence in the result. Naturally, not all prime reciprocals are ideal: for example, $1/3 = 0.\overline{3}$ since the repeated decimal is just a single number. $1/7 = 0.\overline{142857}$ isn't much better, only supplying six numbers.

We want to choose a prime $p$ whose reciprocal has a decimal expansion with a *maximal period* $p - 1$, such as 2063, 10463, 20087, 100667, and 2040287. For example, 1/2063 produces a repeated decimal 2062 digits long.

findRepSeq() in Listing 3.19 (repFrac.py) finds the repeating decimal sequence in the rational supplied as a separate numerator and denominator.

```
def findRepSeq(numer, denom):
  divResult = ""
  remPosMap = {}
  rem = numer % denom
  while ((rem != 0) and \
         (rem not in remPosMap)):
    remPosMap[rem] = len(divResult)
    rem = rem * 10
    quotient = rem // denom
    divResult += str(quotient)
    rem = rem % denom
  if (rem == 0):
    return None
  else:
    return divResult[remPosMap[rem]:]

if __name__ == "__main__":
  numer, denom = map(int, input("numer
       denom=? ").split())
  decimal.getcontext().prec = 100
  print("Fraction:",D(numer)/D(denom))
  seq = findRepSeq(numer, denom)
  if (seq == None):
    print("No repeating seq")
  else:
    print("Repeating seq length:", len
        (seq))
    print("Repeating seq is", seq)
    c = collections.Counter(seq)
    cSort = dict(sorted(c.items()))
    print(cSort)
```

Listing 3.19. Finding a fraction's repeated decimal

For example:

```
> python repFrac.py
numer denom=? 2 3
Fraction: 0.666666666666
6666666666666666666666
6666666666666666666666
6666666666666666666666
6666666666666667
Repeating seq length: 1
Repeating seq is 6
{'6': 1}

> python repFrac.py
numer denom=? 1 7
Fraction: 0.142857142857
1428571428571428571428
1428571428571428571428
1428571428571428571428
1428571428571429
Repeating seq length: 6
Repeating seq is 142857
{'1':1, '2':1, '4':1, '5':1,
 '7':1, '8':1}
```

The while-loop generates a quotient and new remainder when the current remainder is divided by the denominator, and the quotient is added to the end of a division result string.

The unusual aspect of the code is the presence of a map which uses the remainders as keys, and the current length of the division result as each key's value. The map is employed to check if the current remainder has appeared previously, in which case the decimal sequence is about to repeat. The loop terminates and

the map value for the remainder is used to return the slice of the divResult string that contains the repeating decimal.

The top-level of the program prints the repeated decimal, and also its length and a tally of the digits used in the sequence.

It's known that for any prime $p$ excepting 2 and 5, the decimal expansion of $1/p$ repeats with a period that divides $p-1$. In other words, the period could be as large as $p-1$, but no larger, and if it's less than $p-1$, then it's a divisor of $p-1$.

Listing 3.20 (primePeriods.py) checks this by generating all the primes below 3000 and uses findRepSeq() to calculate their repeated decimals. The output reports the length of each repeated sequence, and stars those whose lengths are maximal.

```
ps = primes(3000)
i = 0
for p in ps:
  seq = findRepSeq(1, p)  # find seq for 1/p
  if seq == None:
    print(f"{p:4d}:{0:<4d}  ", end = '')
  else:
    if len(seq) == p-1:
      print(f"{p:4d}:{len(seq):<4d}* ", end = '')
    else:
      print(f"{p:4d}:{len(seq):<4d}  ", end = '')
  i += 1
  if i%6 == 0:
    print()
print()
```

Listing 3.20. Prime reciprocals' repeated decimal lengths

Partial output:

```
> python primePeriods.py
   2:0        3:1        5:0        7:6   *   11:2       13:6
  17:16  *   19:18  *   23:22  *   29:28  *   31:15      37:3
  41:5       43:21      47:46  *   53:13      59:58  *   61:60  *
  67:33      71:35      73:8       79:13      83:41      89:44
  97:96  *  101:4      103:34     107:53     109:108 *  113:112 *
 127:42     131:130 *  137:8      139:46     149:148 *  151:75
    :  # many lines not shown
2803:1401  2819:2818* 2833:2832* 2837:709   2843:1421  2851:2850*
2857:408   2861:2860* 2879:1439  2887:2886* 2897:2896* 2903:2902*
2909:2908* 2917:1458  2927:2926* 2939:2938* 2953:984   2957:1478
2963:1481  2969:371   2971:2970* 2999:1499
```

It's possible to check the starred primes since they're sequence A073761 at OEIS (`https://oeis.org/A073761`).

As to why the decimal expansion of $1/p$ repeats with a period that divides $p-1$, consider

$$1/7 = 0.\overline{142857}$$

with a period of 6. Now multiply it by $10^6$:

$$10^6 \times 1/7 = 142857.\overline{142857}$$

Subtract the original

$$(10^6 - 1) \times 1/7 = 142857$$

And so,

$$1/7 = 142857/(10^6 - 1)$$

The denominator (7) is one more than a power of 10 on the right-hand side, and that power is the period of the decimal expansion.

Rearrange the equation:

$$10^6 - 1 = 142857 \times 7$$

More generally,

$$10^n - 1 = a \times d$$

where $a$ is some multiple, $d$ the denominator of the original fraction, and $n$ is the period of the decimal expansion.

Since $a \times d \equiv 0 \pmod{d}$ then

$$10^n - 1 \equiv 0 \pmod{d}$$

or

$$10^n \equiv 1 \pmod{d}$$

Excluding the primes 2 and 5, the decimal expansion of $1/p$ has a repeating part with period $n$ such that

$$10^n \equiv 1 \pmod{p}$$

or

$$p|10^n - 1,$$

This gives us the smallest $n$ such that $10^n - 1$ is divisible by $p$. How do we know there exists any such integer $n$? Fermat's little theorem tells us that

$$p|10^{p-1} - 1$$

and so we could set $n = p - 1$, though this may not be the smallest such $n$. But, not only does $n$ exist, we know that it is at most $p - 1$. The repeating part of $1/p$ is no more than $p - 1$.

We can employ modular arithmetic to determine the length of the period for prime reciprocals. Since $10^n \equiv 1 \pmod{d}$ we can just look for the smallest $n > 0$ satisfying this.

For example, for 1/13, $d = 13$, so try $10^2 \equiv 9 \pmod{13}$, $10^3 \equiv -1 \pmod{13}$, ..., until $10^6 \equiv 1 \pmod{13}$. The repeating part of 1/13 is 6 digits long.

The output from Listing 3.20 listed 2939 as a prime with a maximal period. The details can be obtained by calling `repFrac.py`:

```
> python repFrac.py
numer denom=? 1 2939
Fraction: 0.000340251786321878189860496767608029942157196325280707
772371554950663490983327662470227968696835658387 21
Repeating seq length: 2938
Repeating seq is 00034025178632187818986049676760802 99
     : # many more lines
47056822048315753657706702960190541
{'0': 293, '1': 294, '2': 294, '3': 294, '4': 294,
 '5': 294, '6': 294, '7': 294, '8': 294, '9': 293}
```

The sequence is the correct length, and the distribution of the digits is fairly uniform, which might be useful.

**Exercise**

The output above suggests that the frequencies of the digits in a maximal period differ by at most 1. Prove that if division by $p$ has period $p - 1$ then this will always be so. Similarly, the numbers of occurrences of pairs, triples etc. of digits differ by at most 1. Write programs for checking this for pairs and triples.

## 3.12 Shift Registers: More Coin Tosses

Figs. 3.32 to 3.34 show six shift registers, each filled by a vector of binary digits.




Figure 3.32. Shift registers (a) and (b)

The registers generate a stream of bits as follows: first the modulo 2 sum $s$ of the tapped cells (indicated by arrows leading from these cells) is formed. Then the initial vector moves one place to the left and the resulting empty cell is filled by $s$.

Take for instance the shift register (c) in Fig. 3.33.

The register can assume $2^7$ or 128 states, but 0000000 is a special state that cannot be reached from a non-zero vector. We have tapped the cells so that the vector will pass through all of these before the initial vector is regenerated.




Figure 3.33. Shift registers (c) and (d)




Figure 3.34. Shift registers (e) and (f)

These six registers can also be described by linear difference equations, and so are sometimes called *linear shift registers* (LSRs). The registers in Figs. 3.32 to 3.34 can be defined by

$$x_n = x_{n-1} + x_{n-4}, \quad x_n = x_{n-5} + x_{n-6}, \quad x_n = x_{n-6} + x_{n-7},$$
$$x_n = x_{n-5} + x_{n-9}, \quad x_n = x_{n-7} + x_{n-10}, \quad x_n = x_{n-9} + x_{n-11},$$

respectively. All of them employ addition mod 2.

Listing 3.21 (shift.py) encodes a shift register as a list.

```
def shifts(regs,fn,r1,r2,numShifts):
  initRegs = regs[:]
  print(*regs, sep='', end = ' ')
  for i in range(numShifts):
    if i%7 == 0: print()
    xn = fn(regs[r1], regs[r2])
    regs = regs[1:]+[xn]
    if regs == initRegs:
      print(*regs,sep='',end = '* ')
      print(i+1)
    else:
      print(*regs,sep='',end = ' ')
  print()

if __name__ == "__main__":
  shifts([0, 1, 1, 0, 1, 0, 0],
         xor, -6, -7, 134)
```

Listing 3.21. Implementing a shift register

```
> python shift.py
0110100 1101001 1010010 0100101
1001011 0010111 0101110 1011101
0111011 1110111 1101110 1011100
0111001 1110011 1100110
  : # more lines not shown
1100011 1000110 0001101 0011010
0110100* 127
1101001 1010010 0100101 1001011
0010111 0101110 1011101
```

The top-level call in Listing 3.21 evaluates shift register (c) in Fig. 3.33, and is equivalent to the LSR equation $x_n = x_{n-6} + x_{n-7}$. 134 shifts are printed.

The starred register 0110100 in the output indicates that the initial state has been re-entered, in this case after a period of 127.

The following execute the shift registers (a) and (b) in Fig. 3.32:

```
>>> from shift import *

>>> shifts([1,0,1,0], xor, -1, -4, 20)
1010 0101 1011 0110 1100 1001 0010 0100 1000 0001
0011 0111 1111 1110 1101 1010* 15
0101 1011 0110 1100 1001

>>> shifts([1,0,1,1,1,0], xor, -5, -6, 70)
101110 011101 111011 110110 101100 011001 110011
100110 001101 011010 110101 101010 010101 101011
010111 101111 011111 111111 111110 111100 111000
  : # more lines not shown
001110 011100 111001 110010 100100 001001 010010
100101 001011 010110 101101 011011 110111 101110* 63
011101 111011 110110 101100 011001 110011 100110
```

**Exercises**

(1) Show that all the shift registers in Figs. 3.32 to 3.34 generate sequences with maximal periods.

(2) Show that the binary LSR-sequences are *purely periodic* in the same sense as Fig. 2.3a.

(3) In the sequence 1983113835952... each digit after the fifth is the modulo 10 sum of the preceding four digits. Does the sequence contain the word

a) 1234; b) 3269; c) 5198; d) 1983 a second time; e) 1357?

Parts a) to d) can be answered without the aid of Python, but coding might be useful for (e). Experiment with different initial values. How does the period length depend on the initial values?

(4) The two LSRs in Fig. 3.35 generate streams of binary and base 5 digits with maximal periods. Check this with Python. Initially the cells are loaded with a non-zero vector.

We call the two streams $x$ and $y$, and can use $x$ to transform $y$ into a sequence of decimal digits by means of the following algorithm: if $x_n = 0$, leave $y_n$ unchanged. If $x_n = 1$, set $y_n = y_n + 5$. What's the period of the resulting sequence of decimal digits?

(5) The LSRs defined by the recurrences $x_n = x_{n-14} + x_{n-17} \pmod 2$ and $y_n = 4y_{n-1} + 3y_{n-7} \pmod 5$ generate sequences with periods $2^{17} - 1$ and $5^7 - 1$




(a) Binary LSR: $x_n = x_{n-3} + x_{n-5} \pmod 2$

(b) Base-5 LSR: $y_n = 4y_{n-4} + y_{n-5} \pmod 5$

Figure 3.35. Shift registers for Ex. 4

respectively. If they're combined in the same way as in Ex. 4, what is the period of the resulting sequence?

**Additional Remarks about LSRs.** The quality of the random digits generated by LSRs goes up (i.e. they satisfy more standard tests) as the order of the corresponding difference equations increases.

The recursion given for $n \geq 55$ by $x_n = (x_{n-24} + x_{n-55}) \mod m$, is recommended by Knuth as a very good generator of random digits in an even base (Vol. 2, Sec. 3.3.2, [**Knu97**]) when $x_0, ..., x_{54}$ are not all even initially. For $m = 2^e$ the length of the period is

$$2^f(2^{55} - 1), 0 \leq f < e.$$

Listing 3.22 (randig10.py) implements the $m = 10$ case. It generates $n$ random decimal digits and counts the frequencies of the digits.

```
SIZE = 55
n = int(input("n=? "))
freqs = [0]*10
x = [random.randint(0,9)
       for i in range(SIZE)]
j = 23; k = SIZE-1
for i in range(n):
  d = (x[k]+x[j])%10
  freqs[d] += 1; x[k] = d
  j = (j-1)%SIZE
  k = (k-1)%SIZE
for i in range(10):
  print(i, freqs[i])
```

```
> python randig10.py
n=? 10000
0 978
1 982
2 991
3 1041
4 987
5 1000
6 959
7 998
8 1045
9 1019
```

Listing 3.22. A Knuth generator

We initially fill x[0],..., x[54] with (low quality) random digits, and set $j = 23$ and $k = 54$. The evaluation progresses counterclockwise around the circle

in Fig. 3.36, adding the pairs mod 10 and reducing $j$ and $k$ by 1. When $j$ or $k$ reaches 0, the index jumps to 54. The results of the additions are (high quality) decimal random digits.

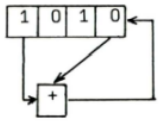



Figure 3.36. The Knuth LSR generator

The digit frequencies reported in the output are spread around 1000, but not too tightly, which is an indicator of the randomness of the generated numbers.

## 3.13 Random Sequence Generation by Cellular Automata

Since randomness is tantamount to total unpredictability, and it's well known that simple quadratic difference equations can lead to unpredictable, and even chaotic behavior, we now turn to nonlinear shift registers (NLSR) or "cellular automata". For more details on this approach, see S. Wolfram, 'Random Sequence Generation', *Advances in Appl. Math.* 7 (1986), pp.123-169.

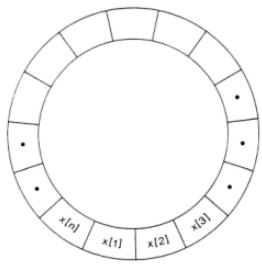



Figure 3.37. A CA buffer

A simple, well studied and very good generator of coin tosses is

$$y_n = x_{n-1} + x_n + x_{n+1} + x_n x_{n+1} \pmod 2$$

This can be written more simply and efficiently as

$$y_n = x_{n-1} \textbf{ xor } (x_n \textbf{ or } x_{n+1}).$$

It transforms an infinite sequence $(x_n)$ into another sequence $(y_n)$, which we'll implement with the help of a circular buffer (see Fig. 3.37).

For $i = 1, 3, ..., n-2$ we use the recurrence

$$y[i] = x[i-1] \textbf{ xor } (x[i] \textbf{ or } x[i+1]),$$

but

$$y[0] = x[n-1] \textbf{ xor } (x[0] \textbf{ or } x[1]),$$
$$y[n-1] = x[n-2] \textbf{ xor } (x[n-1] \textbf{ or } x[0]).$$

Listing 3.23 (`orxor1.py`) is based on these ideas.

```
n,max = map(int, input("n max=? ").
    split())
x = [random.choice([True, False]) for
    i in range(n)]
y = [False]*n
for _ in range(max):
  y[0] = x[n-1] ^ (x[0] or x[1])
  y[n-1] = x[n-2] ^ (x[n-1] or x[0])
  for i in range(1,n-2):
    y[i] = x[i-1] ^ (x[i] or x[i+1])
  for i in range(n):
    print(int(y[i]), end=' ')
  print()
  for i in range(n):
    x[i] = y[i]
```

Listing 3.23. A CA generator

Typical output:

```
> python orxor1.py
n max=? 15 10
0 0 0 1 1 1 0 1 1 0 0 0 1 0 0
0 0 1 1 0 0 0 1 0 1 0 1 1 0 0
0 1 1 0 1 0 1 1 0 1 0 1 0 0 0
1 1 0 0 1 0 1 0 0 1 0 1 1 0 0
1 0 1 1 1 0 1 1 1 1 0 1 0 0 1
0 0 1 0 0 0 1 0 0 0 0 1 1 0 1
1 1 1 1 0 1 1 1 0 0 1 1 0 0 1
0 0 0 0 0 1 0 0 1 1 1 0 1 0 1
1 0 0 0 1 1 1 1 1 0 0 0 1 0 1
0 1 0 1 1 0 0 0 0 1 0 1 1 0 1
```

The list $x$ is filled by $n$ random booleans. Then the sequence $y$ is computed from $x$, and copied back into $x$.

**Exercise**

Explore the similar rule $y_i = x_{i-1}$ **xor** $(x_i$ **or** $(1 - x_{i+1}))$, or

$$y_i = (1 + x_{i-1} + x_{i+1} + x_i x_{i+1}) \pmod 2.$$

## 3.14 The Period Finding Problem

Given a function $f$ which maps a finite domain $D$ of integers into itself, and an arbitrary starting point $x \in D$, the sequence $x_0 = z$, $x_1 = f(x_0)$, $x_2 = f(x_1)$, $x_3 = f(x_2)$, ... is ultimately periodic. That is, for some $t$ and $c$ we have $t + c$ distinct values $x_0, x_1, ..., x_{t+c-1}$, but $x_{t+c} = x_t$. This implies that $x_{i+c} = x_i$ for all $i > t$. The integer $c$ is the *cycle length* and $t$ is the *tail length*.

This situation is illustrated by the graph in Fig. 3.38. The first $t$ elements (6 in this case) are unique, and comprise the *tail*, while the next $c$ elements (12) form the *cycle*. For example $x_{18} = x_6$, $x_{19} = x_7$, and so on.

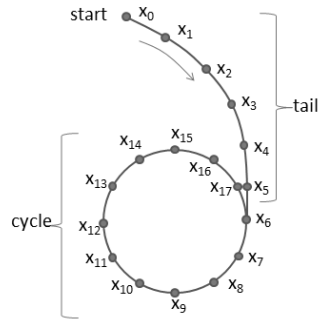



Figure 3.38. A tail and cycle

The problem of finding the unique pair (t, c) is called the *cycle problem*, and arises when analyzing random number generators that produce a new number by applying a function to the preceding value in the sequence, as is the case for the Lehmer PRNG in Sec. 3.1.

One simple solution for the cycle problem is due to R. W. Floyd (see `https://en.wikipedia.org/wiki/Cycle_detection`). We work with two variables, called `slow` and `fast`, which traverse the nodes of the graph at different speeds (see Listing 3.24 (fcycle.py)).

```
def findCycle(fn, start):
  slow = start; fast = start
  cycleLen = 0; tailLen = 0
  # let fast catch up to slow
  while True:
    slow = fn(slow)
    fast = fn(fn(fast))
    cycleLen += 1
    if slow == fast:
      break

  if slow == start:  # no tail
    print("pure cycle")
  else:  # slow goes round cycle
    cycleLen = 0
    while True:
      slow = fn(slow)
      cycleLen += 1
      if slow == fast:
        break
    # calculated cycle len

    # set fast back to start to
        calculate tail
    fast = start
    tailLen = 0
    while True:
      fast = fn(fast)
      slow = fn(slow)
      tailLen += 1
      if fast == slow:
        break
  return cycleLen, tailLen
```

Listing 3.24. Finding a cycle

fcycle() has three main parts. In part 1, `fast` proceeds twice as fast as `slow`, and when they catch up to each other it must be somewhere in the cycle.

In part 2, if they've met at the start node then this graph has no tail, and represents a *pure cycle*. Otherwise, `slow` is sent once more around the cycle in order to measure its length.

Part 3 has `fast` measure the length of the tail. It is sent back to the start node, and now both `fast` and `slow` move at the same unit speed. When they meet again it must be at the first cycle node after the end of the tail (e.g. at $x_6$ in Fig. 3.38). This is the tricky part of the algorithm, but can be proved by considering all the possible combinations of odd and even lengths for $t$ and $c$.

All of the node moves are done by calling fn() which returns the next node in the graph. For example, the graph in Fig. 3.38 is defined using:

```
def dLoop(x):
  if x < 17:
    return x+1
  else:
    return (x%17)+6
```

Listing 3.25. The dLoop() function

The top-level call to findCycle() in this case is:

```
c, t = findCycle( dLoop, 0)
print("dLoop. Cycle len:", c, " tail len:", t)
```

The result is:

```
dLoop. Cycle len: 12  tail len: 6
```

## 3.15 Shrinking Squares

We label the four successive vertices of a square with four non-negative real numbers $a, b, c, d$ (see the first square in Fig. 3.39).

Then each midpoint is labeled with the absolute value of the difference between the labels of its neighbors, and this process is repeated until we get to (0,0,0,0).

Ignoring the pretty nested squares, we are repeatedly applying the transformation

$$T : (a, b, c, d) \mapsto$$
$$(|a - b|, |b - c|, |e - d|, |d - a|).$$

This is implemented in Listing 3.26 (square.py), which prints the successive values of the tuple, and the number of steps until the process stops.

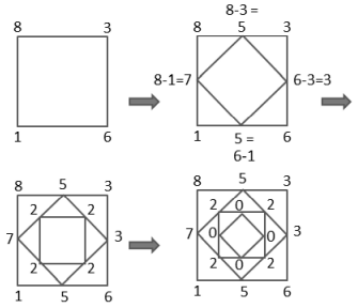



Figure 3.39. Labeling midpoint vertices

```
a,b,c,d = map(float, input("a b c
    d=? ").split())
count = 0
while (a+b+c+d) != 0:
  a1 = abs(a-b); b1 = abs(b-c)
  c1 = abs(c-d); d1 = abs(d-a)
  a = a1; b = b1; c = c1; d = d1
  count += 1
  print(f"{a:10.5f} {b:10.5f} {c
      :10.5f} {d:10.5f}")
print("count =", count)
```

Listing 3.26. Reducing a 4-tuple

```
> python square.py
a b c d=? 1 6 3 8
5.00000 3.00000 5.00000 7.00000
2.00000 2.00000 2.00000 2.00000
0.00000 0.00000 0.00000 0.00000
count = 3
```

After playing with Listing 3.26 for some time, the impression is that we always reach $(0, 0, 0, 0)$ after not too many steps. But there is an exception.

The differences in consecutive 4-tuples form a geometric sequence, which can be made explicit by labeling the values as $\{1, \quad t, \quad t^2, \quad t^3\}$ where $t$ is some number $> 1$. Note that the first three differences, $t-1, t^2-t$ and $t^3-t^2$, advance by the same $t-1$ factor:

$$1(t-1), \quad t(t-1), \quad t^2(t-1)$$

To make the fourth difference advance in the same way, we require that

$$\begin{aligned} t^3 - 1 &= (t-1)t^3 \\ (t-1)(t^2+t-1) &= (t-1)t^3 \\ t^2+t-1 &= t^3 \\ t^3 - t^2 - t - 1 = 0 \end{aligned}$$

Now the absolute differences reproduces the original configuration, multiplied by $t-1$:

$$t-1, \quad (t-1)(t+1), \quad (t-1)t^2, \quad (t-1)t^3$$

which means that the algorithm will never reach $(0, 0, 0, 0)$ and never stop.

It turns out that the only initial values for which the algorithm does not stop are those obtained from the above by an affine mapping $y = ax + b$ applied to each component of the 4-tuple, or by reversing the order in which the numbers are written.

This problem can be solved using matrices and eigenvectors, as detailed in Problem 29 'Hustle Off to Buffalo' in M. Gardner, *Riddles of the Sphinx* [**Gar88**]. See also p.80 of R. Honsberger, *Ingenuity in Mathematics* [**Hon70**]. We'll instead find the roots for the equation $t^3 - t^2 - t - 1 = 0$ using our bisection code from Listing 1.25.

```
>>> from bis import *
>>> def fn(t):
  return t*t*t - t*t - t - 1

>>> fn(1)
-2
>>> fn(2)
1
>>> fn(3)
14
>>> bis(fn, 1, 2)
1.8392867553047836
```

For visual confirmation, we can plot the equation using plotEq.py:

```
> python plotEq.py
x equation? x*x*x - x*x - x -1
a b=? 1 2
```

The plot is shown in Fig. 3.40.

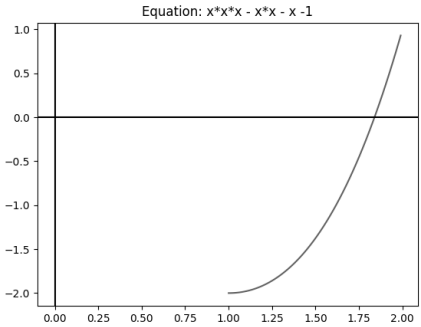



Figure 3.40. Plot of $x^3 - x^2 - x - 1$

**3.15.1 Transforming n-tuples.** Let's consider the more general mapping of an n-tuple containing rationals $x_0, x_1, ..., x_{n-1}$.

$$T : (x_0, x_1, \cdots, x_{n-1}) \rightarrowtail (|x_0 - x_1|, |x_1 - x_2|, \cdots, |x_{n-1} - x_0|).$$

After the first step the n-tuple will only contain positive numbers. Also for any factor $c > 0$, $(x_0, x_1, ..., x_{n-1})$ and $(cx_0, cx_1, ..., cx_{n-1})$ have the same life expectancy or cycling behavior. This allows us to replace the tuple's rationals by positive integers by multiplying them by a suitable $c$ to create integers which are pairwise co-prime.

Listing 3.27 (polygon.py) starts with $n$ random integers from $\{0, 1, ..., 99\}$ and repeatedly applies the $T$ mapping until the $n$-tuple contains only zeros.

```
n = int(input("n=? "))
xs = [random.randint(1,99)
          for i in range(n)]
count = 0
while sum(xs) != 0:
  count += 1
  t = xs[0]
  for i in range(n-1):
    xs[i] = abs(xs[i] - xs[i+1])
  xs[n-1] = abs(xs[n-1] - t)
  for i in range(n):
    print(f"{xs[i]:4}", end ='')
  print();

print("count =" ,count)
```

Listing 3.27. Reducing polygon values to 0s

```
> python polygon.py
n=? 8
  52  66  21  36  62  51  33   1
  14  45  15  26  11  18  32  51
  31  30  11  15   7  14  19  37
   1  19   4   8   7   5  18   6
  18  15   4   1   2  13  12   5
   3  11   3   1  11   1   7  13
   8   8   2  10  10   6   6  10
   0   6   8   0   4   0   4   2
   6   2   8   4   4   4   2   2
   4   6   4   0   0   2   0   4
   2   2   4   0   2   2   4   0
   0   2   4   2   0   2   4   2
   2   2   2   2   2   2   2   2
   0   0   0   0   0   0   0   0
count = 14
```

By experimenting with this program we made the following discoveries:

a) The algorithm always stops if $n$ is a power of 2, as in the example above.

b) For $n \neq 2^r$ (e.g. $n = 7$) the algorithm almost never stops as it enters a cycle.

c) In a cycle, eventually only two numbers survive: 0 and some positive integer.

d) A consequence of c) is that the initial n-tuple may as well just use some combination of 0s and 1s. Indeed, this would allow us to replace the absolute difference operation by modulo arithmetic since

$$|a - b| = (a + b) \bmod 2 = a \text{ xor } b.$$

The expression $(a + b) \bmod 2$ is useful in algebraic work, since addition mod 2 has the algebraic properties of ordinary addition, whereas expressions involving absolute values can seldom be simplified. For example, we can use the additivity of $T \bmod 2$, in the following

$$\begin{aligned} T(x_0 + y_0, x_1 + y_1, ..., x_{n-1} + y_{n-1}) = \\ T(x_0, x_1, ..., x_{n-1}) + T(y_0, y_1, ..., y_{n-1}) \end{aligned}$$

to state that if $c$ is the cycle length of both $n$-tuples on the right, then it's also the cycle length, or a multiple of the cycle length, of the left-hand tuple.

e) If $n$ is odd, then $(1, 1, 0, ..., 0)$ always appears in the cycle.

We can use d) and e) to efficiently determine the maximal cycle length $c(n)$ for odd $n$. For any $n$-tuple $x$, the elements of $T_x$ have the sum to 0 mod 2. Hence the $n$-tuples of a cycle must all sum to 0 mod 2. Every such n-tuple is the element-wise sum modulo 2 of n-tuples with two neighboring 1's and 0's in all other places. This, and the remark we made after d), imply that the cycle length of $(1, 1, 0, ..., 0)$ is $c(n)$.

Listing 3.28 (cycle1.py) finds $c(n)$ for odd $n$ by applying $T$ until the $n$-tuple reappears.

```
n = int(input("n=? "))
xs = [0]*n; xs[0] = 1; xs[1] = 1
print(*xs); c = 0
while True:
  while True:
    c += 1; t = xs[0]
    for i in range(n-1):
      xs[i] = xs[i] ^ xs[i+1]
    xs[n-1] = xs[n-1] ^ t; print(*xs)
    if (xs[0] + xs[1]) == 2: break
  if sum(xs[2:n]) == 0: break
print("cycle length c =", c)
```

Example:

```
> python cycle1.py
n=? 7
1 1 0 0 0 0 0
0 1 0 0 0 0 1
1 1 0 0 0 1 1
0 1 0 0 1 0 0
1 1 0 1 1 0 0
0 1 1 0 1 0 1
1 0 1 1 1 1 1
1 1 0 0 0 0 0
cycle length c = 7
```

Listing 3.28. Finding cycles for odd $n$ sized tuples

Various cycle lengths for different $n$ are shown in Tables 3.1 and 3.2.

| n | 3 | 5 | 7 | 9 | 11 | 13 | 15 | 17 | 19 | 21 | 23 | 25 | 27 | 29 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c(n) | 3 | 15 | 7 | 63 | 341 | 819 | 15 | 255 | 9709 | 63 | 2047 | 25575 | 13797 | 475107 | 31 |

Table 3.1. Cycle lengths for $n$

| n | 33 | 35 | 37 | 39 | 41 | 43 | 45 | 47 | 49 | 51 |
|---|---|---|---|---|---|---|---|---|---|---|
| c(n) | 1023 | 4095 | 3233097 | 4095 | 41943 | 5461 | 4095 | 8388607 | 2097151 | 255 |

Table 3.2. Cycle lengths for $n$ (continued).

f) Our tables suggests that $n|c(n)$.

cycle2.py finds cycles for even $n$ by adapting the "hare-and-turtle" algorithm that we employed in Sec. 3.14.

Examples:

```
> python cycle2.py
n=? 80
Catchup: 240 ; Cycle len: 240 ; Tail len: 13

> python cycle2.py
n=? 160
Catchup: 480 ; Cycle len: 480 ; Tail len: 31

> python cycle2.py
n=? 31
pure cycle
Catchup: 31 ; Cycle len: 31 ; Tail len: 0
```

With cycle2.py, we make the following discoveries:

g) $c(2n) = 2c(n)$ (e.g. compare the first two examples from above),

h) The number of catch-up steps is always equal to the cycle length.

i) The tail length can vary from 0 (which defines a pure cycle) to $2^r$, where $r$ is the maximum exponent, such that $2^r|n$.

j) $c(n) = n$ for $n = 2^r - 1$ (e.g. see the last example above).

For odd $n$ we have $c(2^r n) = 2^r \cdot c(n) = 2^r n \cdot q(n)$, but we need more information about $q(n)$. Very often we have

$$q(n) = 2^m - 1 \quad \text{and} \quad c(n) = n(2^m - 1). \tag{3.11}$$

The first two exceptions are $n = 23$ and $n = 35$:

```
> python cycle2.py
n=? 23
pure cycle
Catchup: 2047 ; Cycle len: 2047 ; Tail len: 0

> python cycle2.py
n=? 35
pure cycle
Catchup: 4095 ; Cycle len: 4095 ; Tail len: 0
```

$c(23) = 2^{11} - 1$ and $c(35) = 2^{12} - 1$. Thus $q(23)|(2^{11} - 1)$ and $q(35)|(2^{12} - 1)$.

What about $m$ in Equ. (3.11)? Its value appears to be an order of 2 mod $c(n)$, i.e. the smallest $m$ such that $2^m \equiv 1 \bmod c(n)$. That is

$$c(n)|(2^m - 1).$$

Finding proofs for most of our discoveries is quite difficult.

**Exercises**

(1) Show that we always arrive at an all-zero $n$-tuple for $n = 4$ and $n = 2^r$.

(2) Show that within a cycle just two numbers survive: 0 and some other number $a > 0$.

(3) Show that for odd $n$ the $n$-tuple $(1, 1, 0, ..., 0)$ always appears in a cycle, and $(1, 0, ..., 0)$ never does.

(4) Show that $c(2n) = 2c(n)$.

(5) Show that $c(n) = n$ for $n = 2^r - 1$.

(6) Show that $n|c(n)$ for all $n$.

(7) Suppose $t = 1.8392867553047836$. How many steps do you need to reduce $(1, t, t^2, t^3)$ to $(0, ..., 0)$?

(8) *Bulgarian Solitaire.* a) Start with a triangular number $t = n(n+1)/2$ of cards distributed into stacks of any number $k$. A move consists in taking one card from each stack and forming a new stack with those $k$ cards. One-card stacks will disappear during this move. You win the game as soon as the table has $n$ stacks containing $1, 2, 3, ..., n$ cards respectively.

Write a program which plays the game. Count the number of moves, and try to find empirically the maximum number of moves until you win.

b) Investigate the case when $t$ is not a triangular number, and a win is no longer possible. Develop some theorems about the cyclic structure of the game.

# Additional Exercises for Chapters 1 to 3

(1) Write a function to represent the graph in Fig. 3.41.

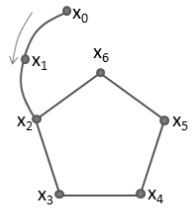



Figure 3.41. Another tail and cycle

Calculate its cycle and tail lengths using Listing 3.24 (fcycle.py).

(2) Incorporate the Lehmer PRNG from Listing 3.1 into Listing 3.24 to calculate its cycle length $c$ and tail length $t$. Note that its default values for MOD and CONST will cause Listing 3.24 to take a long time to finish, so you might want to start with smaller numbers for those constants. What do the reported $c$ and $t$ values mean?

(3) We define an infinite binary sequence as follows: start with 0 and repeatedly replace each 0 by 001 and each 1 by 0.

a) Write a program that prints this sequence.

b) What is the 10000-th bit of the sequence?

c) What is the proportion of zeros among the first 10000 bits of the sequence?

d) Can you show that the sequence is not periodic?

e) Develop a formula for the place numbers of the 1's that is, for the sequence 3, 6, 10, 13, 17,.... Also find a formula for the place numbers of the 0's.

(4) Start with $a[0], a[1], ..., a[n]$ and extend it using $a[n+1] = \max_{0 \le i \le n}(a[i] + a[n-i])$. You will find that the differences ultimately become periodic. For instance, start with $a[i] =$ random.randint(10) for $i = 1, 2, ..., 9$.

(5) *An unsolved problem due to L.E. Dickson.* Given $k$ integers $a[0] < a[1] < ... < a[k-1]$ define $a[n]$ for $n > k-1$ as the least integer greater than $a[n-1]$ which is not of the form $a[i] + a[j]$, $i, j < n$.

Is the sequence of differences $a[n] - a[n-1]$ eventually periodic? Take $k = 2, a[0] = 1, a[1] = 6$. Can you detect a periodicity in the sequence of differences? What about the set $\{1, 4, 9, 16, 25\}$? Is our cycle finding algorithm of any use here?

(6) Start with the digit 1 and repeatedly apply the replacement rule $T$:

$$0 \to 0000, \quad 1 \to 1321, \quad 2 \to 0021, \quad 3 \to 1300.$$

We get $T(1) = 1321$, $T^2(1) = T(1321) = T(1)T(3)T(2)T(1) = 1321130000211321$, ..., so that the sequence $T^{n+1}(1)$ has $T^n(1)$ as its initial segment. Thus, if we start with 1 and apply $T$ repeatedly, we get longer and longer initial segments of an infinite sequence.

a) Show that this sequence does not change when $T$ is applied.

b) Check if there is a period in the first 30,000 digits.

c) Show that this sequence is not periodic.

**Remark**. By placing a dot in the infinite sequence, e.g. $t = 1.321130000211321...$, we get the infinite expansion in some base $\ge 4$. Then one can prove by advanced techniques that $t$ is either rational (if periodic) or transcendental. We cannot get an algebraic irrational like $\sqrt{2}$ by fixed replacement rules, which means that $t$ must be transcendental.

(7) *Numbers with the equisum property.* Write a program which generates random words with $2n$ decimal digits and checks if the sum of the first $n$ digits is equal to the sum of the last $n$ digits. Estimate the corresponding probability $p(n)$, and also $\sqrt{n}p(n)$. Guess a good approximation for $p(n)$.

(8) For the last time, we return to the sequence defined by

$$a_0 = 3, a_1 = 0, a_2 = 2,$$
$$a_n = a_{n-2} + a_{n-3}, n > 2.$$

One can show that $n$ prime $\to n | a_n$ is always true. Show that $n | a_n$ for $n = 271441 = 521^2$. This may be the smallest counterexample to the conjecture $n | a_n \to n$ prime. There is no counterexample up to $n = 140000$. With matrices we can speed up the program from $O(n^2)$ to $O(n \log n)$, resulting in a huge saving in computation time.

(9) Write a program to find the smallest $n$ with the following property: in the binary representation of $1/n$ the bit patterns of the binary representations of all the numbers from 1 to 1990 occur somewhere. $n$ can also be found by Olympiad-level hard thinking plus some number theory.

(10) A die is rolled repeatedly until the sum $X_1+X_2+...+X_N$ of the points surpasses 100. Find by simulation

a) the most probable value of $N$

b) the most probable sum.

(11) *The Newton–Pepys problem.* In 1693 Samuel Pepys and Isaac Newton corresponded over the following problem: which of the following three propositions has the greatest chance of success?

(a) Six fair dice are tossed independently and at least one 6 appears.
(b) Twelve fair dice are tossed independently and at least two 6s appear.
(c) Eighteen fair dice are tossed independently and at least three 6s appear.

Pepys initially thought that outcome (c) had the highest probability of occurring, but Newton correctly concluded that outcome (a) is actually more likely. Write a program to confirm Newton's answer.

(12) *Sicherman dice.* Suppose that you have two six-sided dice, one with faces labeled 1, 3, 4, 5, 6, and 8 and the other with faces labeled 1, 2, 2, 3, 3, and 4. Compare the probabilities of occurrence of each of the values of the sum of the dice with those for a standard pair of dice.

Sicherman dice produce sums with the same frequency as regular dice (2 with probability 1/36, 3 with probability 2/36, and so on).

(13) *Three-sum analysis.* Calculate the probability that no triple among $n$ random 32-bit integers sums to 0, and give an approximate estimate for $n$ equal to 1000, 2000, and 4000. Hint: we utilized random.getrandbits(32) in our solution.

Give an approximate formula for the expected number of such triples (as a function of $n$), and run experiments to validate your estimate.

More information can be found at `https://en.wikipedia.org/wiki/3SUM`.

(14) Estimate the probability that a random triangle is obtuse. Approach this as a geometric problem, where the probability is approximated by testing many randomly generated triangles.

For a triangle to be obtuse, you'll recall from high school geometry that it must have an interior angle greater than 90°.

The crucial formula is the cosine law. If we denote the three interior angles of the triangle by $A$, $B$, and $C$ and the lengths of the sides opposite those angles by $a$, $b$, and $c$, respectively, then we have

$$a^2 = b^2 + c^2 - 2bc\cos(A)$$

In other words:

$$\cos(A) = \frac{b^2 + c^2 - a^2}{2bc}$$

The final part of the puzzle is to recall that the cosine of an acute angle, i.e. an angle in the interval $(0, 90°)$, is positive, while the cosine of an angle in the interval $(90°, 180°)$ is negative.

This problem comes from *Digital Dice* by Paul Nahin [**Nah13**].